UNIVERSITY OF MISKOLC

Faculty of Mechanical Engineering and Informatics

PhD Dissertation

Author:

Jawad Ahmad Qasem Alshboul

MSc in Data Science, MSc in Computer Science

József Hatvany Doctoral School of

Information Science, Engineering, and Technology

Title of the Dissertation

Automatic Generation and Evaluation of Programming Questions from Source Code

Research Area

Applied Computer Science

Research Group

Data and Knowledge Bases, Knowledge Intensive Systems

Head of Doctoral School:

Prof. Dr. László Kovács

Academic Supervisor:

Dr. Erika Baksáné Varga

Miskolc, Hungary 2025

Declaration of Authorship

The author hereby declares that this thesis has not been submitted, either in the same or in a different form, to this or any other university to obtain a PhD degree.

The author confirms that the submitted work is his own, and the appropriate credit has been given where reference has been made to the work of others.

Author's declaration

I, the undersigned, Jawad Ahmad Qasem Alshboul, declare that I have prepared this doctoral dissertation and have used only the sources provided.

All parts that I have taken from another source, either directly or in the same content but paraphrased, are clearly marked with the source.

Miskolc, June 2025.

Jawad Ahmad Qasem Alshboul

Acknowledgment

All praise and thanks are due to Allah, the Most Merciful, whose blessings and guidance have allowed me to reach this milestone in my academic journey. Without His mercy and support, this accomplishment would not have been possible.

I would like to express my deepest gratitude to my supervisor, Dr. Erika Baksáné Varga, for her invaluable guidance, encouragement, and patience throughout my doctoral studies. Her insightful feedback and continuous support have greatly shaped this research and helped me grow both academically and personally.

My sincere appreciation also goes to Prof. Dr. László Kovács, Head of the Doctoral School, for his support, facilitation, and the constructive environment provided during my research period. His leadership and encouragement have been instrumental in enabling me to complete this work.

To my parents, whose endless prayers, love, and sacrifices have been the foundation of every achievement in my life, I owe more than words can express. I am forever grateful for their unwavering belief in me. I am also thankful to my brothers and sisters for their support, understanding, and constant encouragement throughout this journey, even during the most challenging days.

Jawad Ahmad Qasem Alshboul

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
ANOVA	Analysis of Variance
API	Application Programming Interface
AQG	Automatic Question Generation
AST	Abstract Syntax Tree
AWS	Amazon Web Services
BERT	Bidirectional Encoder Representations from Transformers
CFG	Control Flow Graph
DF	Degrees of Freedom
DFG	Data Flow Graph
ELO	Elo Rating System (relative model ranking)
F1-Score	Harmonic Mean of Precision and Recall
GPT	Generative Pre-trained Transformer
JSON	JavaScript Object Notation
LLM	Large Language Model
LMS	Learning Management System
MCQ	Multiple-Choice Question
N	Number of Samples
NLP	Natural Language Processing
OWL	Web Ontology Language
PDG	Program Dependence Graph
p-value	Probability Value in Statistical Testing
QG	Question Generation
SPARQL	SPARQL Protocol and RDF Query Language
TIOBE	The Importance of Being Earnest (Programming Languages Popularity Index)

Chapter 1 Introduction

1.1 Background

Automatic Question Generation (AQG) is the process of creating meaningful and relevant questions automatically from various types of input, including text, structured data, images, or videos, using computational methods. In simple terms, it involves designing systems that can understand content, identify key information or patterns, and generate clear, contextually appropriate questions to support learning, comprehension assessment, conversational systems, or data exploration without requiring manual question crafting for each instance.

The evolution of programming education necessitates a profound reflection on how assessment has been designed, delivered, and evaluated. Given that coding has become necessary across academic disciplines and industries, educational institutions increasingly need to develop robust and scalable ways to assess their students' programming knowledge and problem-solving skills. Learners today often study multiple programming languages, including Python, Java, C++, and C, each with unique syntactic and conceptual nuances, making standardized assessment even more challenging.

While most recent studies in AQG have focused on generating questions from natural language texts and, to a lesser extent, visual data, AQG from source code remains underexplored despite its potential to transform programming education. Academic computer programming textbooks consist of materials such as text, images, and code examples. Most recent research on AQG systems focused primarily on natural language processing (NLP) techniques applied to generating questions from text, while some research focuses on generating questions from visuals or images [1], [P3]. The review paper [P2] suggests adopting question-generation methods that would create questions on programming-related topics and evaluation criteria for these questions.

Traditional methods of question design in programming courses have struggled to keep pace with this growth. As noted in previous studies, manually crafted questions are time-consuming to produce, difficult to standardize across diverse learners and languages, and often fall short of covering the full spectrum of cognitive skills outlined in Bloom’s Taxonomy. Moreover, they tend to lack scalability, particularly in large or multi-language educational settings where hundreds of students may require tailored assessment materials.

These challenges have driven a growing interest in AQG from source code. Rather than relying on static repositories of questions, AQG approaches analyze code directly, extracting structure, semantics, and logic to generate assessment items that dynamically align with the learner’s context. This dissertation responds to that demand by presenting a unified exploration of five distinct but complementary approaches: ontology-driven content generation, hybrid semantic-to-question modeling, template-based multi-language question generation via static code analysis, evaluation of large language models for code-based question generation, and a comprehensive Control Flow Graphs (CFG) and Program Dependence Graphs (PDG)-powered multi-language assessment system.

Each approach contributes to a shared objective: to automate programming question generation in a pedagogically grounded, cognitively stratified, and linguistically inclusive way. The background and motivation for this work emerge directly from the collective recognition within these studies of the limitations in existing systems and the urgent need for more intelligent, adaptable, and scalable solutions in programming education assessment.

1.2 Research Motivation

Despite significant advancements in artificial intelligence-driven educational technologies, several critical gaps persist in the domain of automatic question generation for programming education.

The first significant challenge is the lack of scalable systems capable of generating high-quality, diverse, and cognitively stratified questions directly from source code. As identified in the literature, current template-based approaches often rely on rigid pattern matching and are limited in flexibility and adaptability across code types and educational objectives. Previous studies show that manually created questions are time-consuming and struggle to maintain cognitive coverage across large-scale deployments, reinforcing the necessity for automation that accommodates a range of programming logic and learner profiles.

A second limitation is the insufficient support for multi-language question generation across most existing tools. Template and static analysis-based methods typically underperform when handling multi-language syntax and semantics, making them less effective for inclusive educational environments. Additionally, few frameworks integrate pedagogical models such as Bloom's Taxonomy in a systematic way, resulting in assessment items that are either too shallow or mismatched in cognitive depth.

Finally, current evaluation practices for code-based question generation lack standardization and pedagogical alignment. Although large language models (LLMs) like GPT-4 can generate syntactically fluent questions, their outputs vary in relevance, clarity, and educational value. Previous efforts rarely incorporate expert-validated, multi-dimensional evaluation frameworks, further limiting instructional reliability.

These limitations underscore the need for a principled and pedagogically grounded approach to automated question generation from source code. By integrating semantic modeling, cognitive stratification, and rigorous evaluation practices, such an approach can support scalable and equitable learning assessments in programming education.

1.3 Problem Statement

The global expansion of computer science education has intensified the need for scalable, high-quality assessment tools that can effectively serve diverse learners across various programming languages. Traditionally, the manual development of programming assessment questions has been labor-intensive, inconsistent, and insufficient to meet the rising demand for pedagogically sound, comprehensive evaluation materials in programming education.

Automatic Question Generation (AQG) has emerged as a promising approach for scalable assessment across educational contexts. However, the current research landscape in AQG reveals a pronounced imbalance in focus and development across different input modalities. The field has been dominated by text-based question generation, benefiting from extensive datasets, mature neural models, and a clear trajectory from rule-based systems to large pre-trained transformers and LLMs. Similarly, visual question generation has seen growing attention, particularly for generating questions from images and, more recently, educational diagrams, leveraging advancements in multimodal learning. These areas have established robust evaluation practices and benchmarks, fueling rapid progress and adoption.

In contrast, code-based question generation remains significantly underrepresented despite its critical potential in programming education. Generating meaningful and pedagogically aligned questions directly from source code presents unique challenges, including understanding code semantics, aligning questions with relevant programming concepts, and ensuring cognitive coverage across difficulty levels. The lack of standardized datasets and well-defined evaluation metrics further impedes systematic advancements in this domain. Most existing AQG research overlooks this niche, and only a few recent studies have begun exploring it, often in isolated or single-language contexts, leaving a substantial gap in the scalable assessment needs of programming education.

Addressing this gap is essential to ensure equitable, effective, and scalable programming assessment tools that align with modern pedagogical frameworks and can adapt across multiple programming languages. Advancing code-based AQG requires not only robust generation methods that capture the semantics of source code but also the development of principled evaluation frameworks tailored to the unique requirements of programming education.

This dissertation aims to address these gaps to advance scalable, high-quality, and pedagogically aligned AQG systems that support equitable programming education worldwide.

1.4 Research Aims

This dissertation aims to advance programming education by designing, implementing, and evaluating automated systems that generate and assess programming questions directly from source code in a pedagogically grounded, linguistically inclusive, and cognitively diverse manner. This research seeks to bridge the gap between code-level semantic understanding and educational assessment, using various techniques including ontologies, template-based static analysis, and large language models.

A central aim is to alleviate the manual workload of educators while improving assessment quality and scalability across multiple programming languages.

Another aim is to systematically align generated questions with established cognitive learning models, especially Bloom's Taxonomy, to ensure relevance across difficulty levels and educational contexts.

This dissertation also aims to contribute robust evaluation methodologies combining automatic scoring and expert review, improving the reliability and instructional alignment of automatically generated content.

Ultimately, the research aspires to provide an integrated, technically rigorous, and pedagogically valid foundation for future systems in programming assessment, especially in multi-language and large-scale learning environments.

These aims collectively shape the trajectory and cohesion of the dissertation's contributions, reflecting the interdisciplinary intersection of code analysis, natural language generation, and educational measurement.

1.5 Research Objectives

This dissertation sets out to address the limitations outlined in the previous section by pursuing the following core objectives:

1. To develop models and systems that generate programming questions automatically from source code.
2. To ensure alignment with cognitive frameworks such as Bloom's Taxonomy.
3. To support multiple programming languages (Python, Java, C++, C) in a multi-language context.
4. To evaluate both the technical quality and the educational value of the generated questions.

Collectively, these objectives form the foundation of this dissertation's contribution to programming education assessment through artificial intelligence-enhanced, source code-driven question generation and evaluation.

1.6 Scope and Limitations

This dissertation is bounded by the following scope and limitations, each drawn directly from the operational designs and stated constraints of the conducted studies:

1. Focus on source code as input, not textbook content or problem descriptions.
2. Covers four programming languages: Python, Java, C++, and C.

3. Question types generated include multiple-choice questions (MCQs), open-ended questions, Boolean (yes/no) questions, short answer questions, code tracing questions, fill-in-the-blank questions, error identification (debugging) questions, and creative coding questions.
4. Evaluation consists of both automated scoring metrics and human expert review.
5. The study does not include real-time feedback, adaptive learning mechanisms, or student modeling.

These scope boundaries ensure a focused contribution to automated programming question generation from source code while acknowledging the limitations of generalizability. This allows future research to expand toward personalized or interactive assessment systems.

1.7 Significance of the Study

This dissertation makes several key contributions to the field of programming education assessment through automated question generation:

1. It reduces the workload of educators by automating the creation of pedagogically aligned programming questions.
2. It enhances inclusivity by supporting multi-language question generation and cognitively diverse assessment items.
3. It introduces rigorous evaluation pipelines that combine automated metrics and expert review, contributing to the trustworthiness of educational artificial intelligence (AI).
4. It contributes to the growing intersection of natural language processing (NLP), machine learning, and programming pedagogy by applying structured and AI methods to real-world educational tasks.

Together, these contributions affirm the dissertation's relevance not only as a technological endeavor but also as a meaningful advancement in equitable, scalable, and cognitively aligned programming education.

1.8 Dissertation Structure

This dissertation is organized to reflect the systematic development, analysis, and evaluation of five distinct yet interrelated approaches to automatic question generation and assessment from source code.

Chapter 2: Literature Review. This chapter surveys the existing body of research in programming assessment, question generation, semantic code analysis, template design, and the application of large language models. The review contextualizes the core studies conducted in this dissertation.

Chapters 3–7: Research Studies. The following five chapters present a standalone study, structured with an introduction, methodology, results, discussion, conclusion, and summary.

Chapter 3: Ontology-Based Automatic Generation of Learning Materials for Python Programming.

Chapter 4: Hybrid Approach for Automatic Question Generation from Program Codes.

Chapter 5: Evaluating Large Language Models for Code-Based Question Generation in Programming Education.

Chapter 6: Template-Based Question Generation from Code Using Static Code Analysis.

Chapter 7: Multi-Language Static-Analysis System for Automatic Code-Based Question Generation.

Chapter 8: Conclusion. This chapter synthesizes the findings of the dissertation and presents:

8.1 Contributions: Summarizing the theoretical and practical contributions across the five studies.

8.2 Future Work: Outlining potential extensions and improvements for research and system deployment.

8.3 Author’s Publications: listing publications resulting from this research.

While each study stands independently, together they offer a layered, cohesive exploration of the dissertation’s overarching aim. This structure reflects both the linear development of the dissertation and the layered complexity of its contributions across computational, pedagogical, and linguistic dimensions.

Chapter 2 Literature Review

2.1 Introduction

Automatic question generation (AQG) from source code is situated at the intersection of educational assessment, programming pedagogy, static program analysis, and artificial intelligence. As programming becomes a fundamental skill in education and industry, the demand for scalable, cognitively diverse, and pedagogically sound assessment frameworks has intensified. This chapter synthesizes the foundational literature across these intersecting domains, organizing contributions and identifying gaps thematically across ontology-driven instructional content, graph-based static analysis, template-based question systems, large language models (LLMs), multi-language question generation, and the application of Bloom's Taxonomy in automated assessment frameworks.

2.2 Ontology-Based Instructional Content Generation

Effective instruction in programming education requires comprehensive and adaptive learning materials. These materials include textual and visual content, interactive exercises, tutorials, real-world examples, assessment tools, and personalized pathways that reinforce hands-on practice and real-world applicability. Textual content delivers explanations, code examples, and problem sets, while interactive exercises and tutorials facilitate active learning and progressive skill development. Real-world examples bridge theory with practice, and assessment tools measure student progress and understanding. The overarching aim is to provide accessible, engaging, and personalized resources that support varied learning preferences.

Programming languages are a central area of study in computer science and software development. Developing effective methods for teaching programming concepts is essential. Interest in question generation techniques for programming languages has grown as a means of creating scalable practice opportunities, reinforcing learning, and enabling ongoing assessment [P2]. The paper [P] applied ontology to develop a question-generation approach for programming concepts.

Several studies have investigated the possibility of automatic generation of learning materials and their positive impact on enhancing student engagement and learning outcomes. Vergara et al. [2] found that AI-generated personalized learning materials boosted students' motivation and performance in mathematics courses. At the same time, Liu et al. [3] highlighted how AI-powered tools assist educators by automating quiz and worksheet creation, reducing manual workload while maintaining instructional quality. Lin et al. [4] examined the relationship between student engagement and outcomes in a cyber-flipped course, finding a positive correlation between active participation and academic performance, thereby underscoring the value of dynamic course materials in blended learning environments.

Over the years, numerous researchers have explored the use of ontologies in education to automatically create and structure learning materials, enhancing personalization and interoperability within learning management systems [5]. For example, [6] proposed an intelligent ontology-based system to automate tasks such as course scheduling and academic advising, demonstrating improvements in efficiency and student experience through structured domain knowledge. William and Joselin [5] discussed how ontologies enhance personalized learning, advocating for their use in shifting away from one-size-fits-all models to adaptive, student-centered instruction.

In [7], a method for constructing structured knowledge graphs using word embeddings and NLP techniques was introduced, enabling automated semantic extraction and relationship mapping from educational content. This structured approach facilitates reference definition (prerequisites, hierarchy, relatedness), supporting the creation of dynamic, interconnected learning resources. Similarly, Stephen [8] explored the use of LLMs like GPT-3 to generate CS learning materials across topics, evaluating quality, relevance, and coherence to propose innovative methods for scalable CS education. Flanagan et al. [9] proposed leveraging NLP and machine learning (ML) to structure educational content extracted from various sources, aligning it with learning objectives to improve digital learning environments. Meanwhile, [10] detailed the construction and practical application of a knowledge graph within Australian school science curricula, focusing on personalized learning and adaptive tutoring system integration.

Despite the growth of ontology-driven learning material generation, significant limitations remain: insufficient knowledge representation structures, limited flexibility and context awareness, challenges in reusability, and the lack of deep, adaptive personalization. Current systems often require human oversight, lack the interactivity and nuanced feedback of human instruction, and fall short in fostering critical problem-solving skills. Continued AI advancements in contextual understanding and adaptability are necessary to overcome these limitations. Table 2.1 compares traditional methods with ontology-based approaches, highlighting the latter's strengths in semantic structuring, flexibility, scalability, and personalization, which are essential for modern, learner-centered programming education. The complexity of question generation requires expertise, deep content knowledge, and substantial time investment, especially in online learning contexts since the emergence of syntax-based and semantic-based question generation models in 2014 [11], ontologies have proven effective for standardizing knowledge representation across domains, including e-learning, facilitating personalized and efficient learning [P9].

Table 2.1 Comparison between the traditional approaches and ontology-based approaches

Feature/Aspect	Traditional Approaches	Ontology-based Approaches	References
Knowledge Structure	linear and hierarchical	semantic and interconnected	[6], [P1]
Flexibility	limited adaptability to new topics	highly adaptable to new knowledge and domains	[7], [12]
Context Awareness	minimal context consideration	rich context understanding through relationships	[13], [P8]
Content Reusability	low reusability of materials	high reusability due to modular components	[P3], [P9]
Personalization	basic customization, often static	dynamic personalization based on learner profiles	[5], [14]
Scalability	difficult to scale with growing content	easily scalable with ontological frameworks	[15], [16]
Interoperability	often siloed systems	enhanced interoperability across platforms	[2], [17]
Knowledge Representation	simple data structures (e.g., text, images)	rich semantic representation using classes, properties, and relationships	[18], [P13]
Maintenance	time-consuming updates and revisions	more accessible updates due to modular ontology design	[19], [20]
Collaboration Support	limited collaboration features	facilitates collaboration through shared ontologies	[8], [P9]
Learning Pathways	predefined and rigid learning paths	dynamic learning pathways based on learner needs	[2], [3]
Assessment Tools	basic quizzes and tests	adaptive assessments based on learner progress	[21], [22]
Feedback Mechanism	limited feedback based on performance	contextual feedback based on semantic analysis	[9], [23]

Domain knowledge models, particularly those implemented with Python and Owlready2, offer flexible and integrable representations for e-learning systems [P8]. They enable adaptive learning systems capable of tailoring experiences to individual learners, reinforcing efficient knowledge transfer.

Although question generation in programming education holds transformative potential, implementation remains partial in modern contexts. Programming languages, central to CS education, demand effective teaching methods, with question-generation approaches enabling scalable practice and assessment opportunities [P3].

To support learning, Urazova [24] developed a system for automatic UML database design question generation and response evaluation using AI and NLP, providing students with practical, self-assessment tools. Russell [25] explored automated code-tracing exercises in CS1 courses, demonstrating their utility in reinforcing control flow and problem-solving skills, while acknowledging challenges in replacing traditional teaching approaches.

Large language models (LLMs) have recently been applied to generate programming tasks and explanations, offering scalable solutions for instructors [26]. However, challenges remain, including potential bias, dependence on large-scale models, computational demands, and difficulties in generating high-quality training data, all of which must be addressed when implementing these technologies in educational contexts [27].

2.3 Static Code Analysis and Graph-Based Representations

Static code analysis is employed across various domains, particularly in compiler design and security [28]. Static code analysis is used to automate checking student programming assignments. It verifies the correctness of student programming assignments concerning assignment instructions [29]. Many static analysis techniques are based on code representation, and it is critical in performing other tasks that involve drawing deductions about semantic relationships between program statements [30]. A proper code representation procedure allows deriving meaningful source code features that capture different aspects of the source code structure and behavior. Graph-based structures have mainly been employed in recent innovations in code representation to capture both the syntactic and the semantic details embedded in the code. The Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG) are the most commonly used forms of representation. The definitions of AST, CFG, and DFG are as follows:

Definition 1: Abstract Syntax Tree (AST)

An Abstract Syntax Tree for the function f_i in a program $P = \{f_1, f_2, \ldots, f_n\}$ is represented as a graph $G_A^i = (V_A^i, E_A^i)$ where V_A^i is the set of leaf nodes and E_A^i is the set of directed edges, where each edge connects a parent node to its corresponding child node.

Definition 2: Control Flow Graph (CFG)

The Control Flow Graph for the function f_i is defined as a graph $G_C^i = (V_C^i, E_C^i)$ where V_C^i is a set of nodes and E_C^i is a set of directed edges representing the control flow between the nodes.

Definition 3: Data Flow Graph (DFG)

A Data Flow Graph for the function f_i is defined as a graph $G_D^i = (V_D^i, E_D^i)$ where V_D^i is a set of nodes and E_D^i is a set of directed edges capturing variable accesses and modifications during the execution.

The following is a simple example of a small function and shows how its AST, CFG, and DFG would look in a basic form. This will give the reader a clear idea of how each graph is constructed and what it represents. A simple Python function illustrates these structures:

```
# Example Function
```

```
def add(x, y):
    z = x + y
    return z
```

1. Abstract Syntax Tree (AST): The **AST** represents the syntactic structure of the code. It focuses on how the source code is structured, not how it executes or flows.

AST Nodes (simplified):

- FunctionDef
    - Name: add
    - Parameters: x, y
    - Body:
        - Assignment: z = x + y
            - Expression: x + y
        - Return: z

AST Edges:

- Each node connects to its child syntax elements. For example:
    - FunctionDef → Assignment
    - Assignment → Expression
    - Expression → x, Expression → y
    - FunctionDef → Return

2. Control Flow Graph (CFG): The **CFG** shows the control flow from one instruction to another.

CFG Nodes:

1. Start
2. z = x + y
3. return z
4. End

CFG Edges:

- Start → Assignment
- Assignment → Return
- Return → End

Note: Since there is no branching (like if or loop), the CFG is linear.

3. Data Flow Graph (DFG): The **DFG** captures how data (variables) are used and modified.

DFG Nodes (variables): x, y, and z.

DFG Edges:

- x → z (z is computed from x)
- y → z
- z → return (z is used in return)

This tells us that z depends on x and y and is then used in the return statement. Table 2.2 shows a summary of AST, CFG, and PDG.

Table 2.2 AST, CFG, and PDG summary table

Graph Type	What It Shows	Example Focus
AST	Code structure	z = x + y is an assignment with an addition expression
CFG	Execution order	Start → Compute → Return
DFG	Variable flow	x, y → z → return

2.3.1 Automatic Question Generation

Automatic question generation (AQG) has developed as a considerable scholarly subject in learning technology, and it has been used in many fields, such as programming education. Early research in this area tended to target the case of generating natural language questions based on natural language text and not as much about generating program questions based on program code [P3]. The combination of CFG and PDG analyzers and question generation systems can considerably improve the quality and relevance of automatically created questions. A combination of the CFGs to provide program control flow information with PDGs to provide data dependency information can give a more comprehensive view of the program's behavior. In education, artificial intelligence (AI) presents not only challenges but also opportunities, especially in its application to gauge student understanding. The rise of AI-generated code necessitates rethinking assessment practices to accurately measure student understanding and effort [31]. Systems using AST, CFG, and DDG have been developed for grading programming skills [32], demonstrating the potential of structured code analysis for automated evaluation.

2.3.2 Program Analysis

The problem of code analysis in programming languages has been discussed in several settings, but little has been said about a particular case of question generation. The combination of CFG and PDG, analysis done when performing code comprehension, has been examined under various settings. The paper [33] has shown that graph-based neural networks can well be applied to the problem of code understanding by combining information in abstract syntax trees and in data flow graphs. In the same way, the authors in [34] employed graph-based forms to enhance bug detection and code completion. These strategies point to the possibility of using graph-based code analyses to build a better understanding of code at a deeper level, though they have not been used directly to answer questions. More relevant to the present work, the authors [35] built a natural language generator that takes a Python code snippet and generates a natural language description of that code. Their strategy involved a language-specific parser coupled with standard, intermediate representations, just like the current work. Nevertheless, they were concerned with code summarization and giving feedback, and did not discuss the difficulties of achieving balance in coverage of algorithms and cognitive levels.

2.3.3 Control Flow Graph Analyzers

Control Flow Graphs are especially useful for program analysis abstractions and indicate all potential paths of execution in a program. A graph is a model of a program in that each node corresponds to a basic block of code, and edges indicate the flow of control between blocks. The CFG analyzers exploit this format to obtain information about the structure of the program, to find out whether or not there are possible loops and conditional transitions, and to identify unreachable code blocks [30]. Such information can be used invaluably in the generation of questions so that one can then be asked questions that determine how the programmer understands the mechanics of control flows, including loop invariants, branch conditions, and exception handling. Modeling and analysis of the execution flow of a program is paramount in its correctness, reliability, and security [36]. It is possible to extract syntax and semantic information of source code using control flow graphs, which allows a more detailed analysis of the behavior of programs [30].

Questions about the order of statement execution, the circumstances under which different blocks of code are entered, and the possibility of entering an infinite loop or dead code are answerable by studying the control flow. Suggesting a student to concentrate on the control flow, such questions may examine his/her grasp of the logic of the program. Also, CFGs can be used to determine important areas of code that can be looked at in more detail, e.g., performance bottlenecks or error-prone areas.

2.3.4 Program Dependence Graph Analyzers

Program Dependence Graphs are a contrasting view in that they explicitly specify the data and control dependency between distinct program statements. Nodes in a PDG are the individual statements, whereas the edges show whether the value computed by one statement is referenced by another (data dependence) or whether the evaluation of one statement is conditional on the result of another (control dependence) [30]. This representation provides an analysis of critical data dependencies, potential data races, and possibilities of code optimization with PDG analyzers. They give a structure to how questions can be generated, which tests the understanding of the programmer on issues like the flow of data, side effects, and effects of changes in a particular variable or statement. All vulnerable cases of buffer overflows are spatial mistakes, which can be diagnosed with the assistance of spatial information in a data flow graph [37]. Buffer overflow can be discovered with the aid of static data flow analysis.

The PDGs are also capable of determining the inputs that influence or determine specific outputs, which is an important aspect of numerous security vulnerabilities. Data flow presents an analysis of how data is directed through a program and what is done to the data [38]. Data flow is a dependency relationship among variables, with nodes representing variables and edges denoting what caused the value of a variable [39]. Data flow analysis may discover a variety of bugs and is among the most frequently used approaches in practice [40]. Following the interdependency of variables allows determining the possible vulnerabilities, including a buffer overflow or a format string, to be identified.

2.3.5 Hybrid CFG-PDG Analysis

Combining CFG and PDG analyzers provides an effective method to generate questions and thus allows the generation of questions requiring insight into control flow and data dependencies. This combination enables the creation of questions that are more complicated and subtle and tests the reasoning of a learner about the interaction of various program components. The integration of data and control that has been implemented in applications is more intriguing when designing a custom architecture [41]. For example, one may pose questions like whether a modification in a specific variable will affect the execution course of the program or what conditions could cause a particular data dependency to produce a run-time error. This would allow for coming up with more difficult and pertinent actual programming situations. Furthermore, CFG-PDG combinations can also be used to discover the most critical control-sensitive and data-dependent code sections to generate questions that pinpoint the most important parts of program behavior. Combining these techniques improves the capability of defining questions that can assess single pieces of code and code interactions between control flow and data dependency. Beyond question generation, the synergy between CFG and PDG provides broader benefits for comprehensive software understanding and analysis, as discussed next.

2.3.6 Synergistic Use of CFG and PDG

Studies that expand AST-based code representations to cover paths in CFG and PDG have demonstrated dramatic performance benefits to software engineering activities like method naming, classification, and clone detection [42]. This combination of CFG and PDG analyzers provides a more comprehensive picture of the program behavior. It allows us to generate questions that will focus on control flow and data dependencies. The study of the interaction between these two representations can enable the production of questions that demand deeper knowledge regarding the functionality of the program in general and the possible interactions between the various sections of the code. Such integration allows the formulation of questions that are more rigorous and insightful. It results in a more elegant measure of the fairness of assessing the competency of a programmer. Such a combination presents stronger questions, and the programmer understands the code better.

The combination of PDGs and CFGs presents a synergistic effect and is useful when it comes to finding vulnerabilities in code. When control flow and data dependency information are combined, this capability emerges to discover fine-grained defects that may remain elusive to either of the techniques individually [43].

2.3.7 Question Generation Strategies

Designing effective question-generation strategies is critically important in the design of assessments that not only measure the knowledge a programmer has about code, but also measure it accurately. Such strategies must apply to the characteristics of CFGs and PDGs and utilize the strong points of these subjects to outline thought-provoking and relevant questions. Among these approaches are identifying high-priority sections of code, including loops, conditional statements, or function calls, and creating questions about their behavior. The other way is following data dependencies with the PDG, forming questions about the information flow in the program. The assessment should be on relevant issues.

2.4 Template-Based and Question Generation Strategies

Template-based approaches have been widely used in AQG across various domains. The paper [44] provided a comprehensive survey of template-based question-generation techniques, highlighting their effectiveness in ensuring question quality and relevance. It mentioned that the template library is a major component of question-generation systems. The paper [45] addressed educators' challenges in creating exam questions, particularly in remote learning environments. To tackle these challenges, the authors proposed a new approach that combines generative software development principles with feature-oriented product line engineering. This approach was designed to automate the creation of exam questions, specifically single-choice questions, using written templates. The proposed generator allows educators to create families of questions that vary based on specific features and parameters. However, existing template-based AQG methods often fall short in supporting multi-language contexts, balanced algorithm coverage, and strategic difficulty alignment.

This dissertation builds on these foundations while addressing these limitations, ensuring multi-language support and cognitive diversity in question generation.

2.5 Bloom’s Taxonomy and Cognitive Alignment

Bloom's Taxonomy is a starting point from which a set of questions can be classified according to the complexity of thinking skills [46]. Bloom's Taxonomy is a foundational framework for categorizing questions based on cognitive complexity [46]. It includes remembering, understanding, applying, analyzing, evaluating, and creating [46], [47]. In [48], the authors have performed a thorough review of factors that complicate introductory programming tasks and have established several major factors that make questions more or less challenging. Their result offers valuable information in preparing questions of adequate difficulty based on varying programming languages. The tactical use of programming languages' difficulty level has been argued on different educational fronts. These learning theories guide us in generating questions, especially in providing proper cognitive demand, difficulty levels, and language-specific issues.

Integrating 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 [49]. It encompasses remembering, understanding, applying, analyzing, evaluating, and creating [46], [47], [50]. 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. It involves recognizing and recalling relevant knowledge stored in memory.
2. Understanding: Learners demonstrate comprehension by explaining ideas or concepts, summarizing information, and interpreting meaning.
3. Applying: It involves using knowledge in new situations. Learners can apply what they have learned 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.

2.6 Question Types in Programming Education

Programming instructors use a variety of question formats to assess and enhance student understanding, often leveraging automatic question generation from source code. Each question type serves different learning objectives and challenges. The following are the question types in programming education:

1- Multiple-Choice Questions (MCQs): MCQs are a popular assessment tool in programming courses. MCQs can be an effective and motivating way for students to test their understanding of programming concepts [51].

2- Open-Ended Questions: Open-ended questions in programming education require students to provide an unstructured response, such as explaining code or writing their own solution [52].

3- Boolean (Yes/No) Questions: Yes/No or True/False questions are a simple form of assessment where students judge the correctness of a statement. In programming education, these judgment questions are considered a type of closed-ended exercise alongside MCQs and fill-in-the-blanks [53].

4- Short Answer Questions: Short answer questions require a brief textual or numeric response rather than selecting from given options. In programming, this format is often seen in questions like "What is the output of the following code?" or "Give the Big-O time complexity of this algorithm." These questions compel students to recall or deduce an answer without cues. They can assess understanding more directly than MCQs, and recent systems have begun to automatically grade such answers [54].

5- Code Tracing Questions: Code tracing questions present a piece of code and ask students to simulate its execution to determine the outcome or state. A typical prompt might be: "Given this code, what will be the output?" or "What values do the variables hold after execution?" This question type is well-established in programming education as a way to test understanding of control flow and state changes [55].

6- Fill-in-the-Blank Questions: Fill-in-the-blank questions in programming provide a code snippet or sentence with certain parts removed, and students must supply the missing piece. This format is often used to focus attention on specific syntax or concepts [56].

7- Error Identification (Debugging) Questions: Error identification questions, also known as debugging tasks, present students with faulty code and ask them to find and/or fix the error. These questions target a student's ability to read code critically and understand common bugs. For instance, a prompt may say: "This code is supposed to do X but it does not. What is the error and how would you fix it?" [57].

8- Creative Coding Questions: Creative coding tasks are open-ended prompts that require students to write original code to achieve some goal, often with room for creative expression or multiple correct solutions. Unlike the strictly defined answers of the above formats, these questions might ask students to "Design a program that meets scenario X" or "Create a graphic using code that accomplishes Y." The emphasis is on problem-solving, design, and creativity in programming [58].

2.7 Large Language Models in Programming Question Generation

Advances in NLP have led to the emergence of large language models (LLMs). These language models have proven their potential in different NLP applications, including question generation and evaluation [59]. This section reviews the related works that laid the foundations for developing and evaluating LLMs in generative artificial intelligence.

2.7.1 Background On Language Models in NLP

The development of LLMs has been influential [60]. In the past decade, the emergence of LLMs has driven a paradigm shift in NLP [61]. These models are characterized by their immense size, often containing billions of parameters. They are pre-trained on vast amounts of data, which enables them to learn patterns, syntax, and semantics of natural language. Pre-training is followed by fine-tuning specific tasks, making them adaptable to various applications.

Other methods of question generation involve building specialized ontologies and integrating them with artificial intelligence models, such as the research conducted by Alshboul and Baksa-Varga [P3]. The authors adopt a hybrid ontology and artificial intelligence approach to build an automatic question-generation model. It lacks automatic evaluation criteria.

OpenAI's GPT models have continuously improved language generation capabilities, starting with GPT-1 and advancing to GPT-2, GPT-3, and beyond [62]. GPT-3.5, for example, delivered human-level performance on different language tasks, from translation to question-answering.

LLMs have proved their adaptability in NLP tasks. They perform well in text generation, summarization, translation, sentiment analysis, and various other tasks. The capacity to understand and generate text in multiple languages and domains causes such adaptability [62]. While LLMs are powerful tools, they are not without their challenges. Their massive size demands substantial computational resources, making them inaccessible to many researchers and organizations. These models have been criticized for keeping biases in their training data [63]. Research efforts to mitigate these biases and make LLMs fairer have gained attention.

One of LLMs' strengths is their adaptability through fine-tuning [64]. Researchers and practitioners can customize these models for domain-specific tasks, allowing them to perform well in specialized domains. The fine-tuning process involves training the model on task-specific data, enhancing performance and relevance to specific tasks. The growth of LLMs has raised ethical and societal concerns. The ability of these models to generate coherent, human-like responses also means they might be used for malicious activities such as misinformation and deepfakes. Discussions on responsible artificial intelligence and ethical use are ongoing. LLMs have become the focus of many studies, ranging from model architecture and training techniques to healthcare, finance, and education applications. Researchers are exploring ways to harness LLMs' power to benefit society while mitigating potential harms [65].

2.7.2 Question Generation with Large Language Models

Integrating LLMs into language processing has significantly advanced question-generation capabilities. Because of their extensive pre-training on vast text corpora, LLMs have transformed how questions are generated. This section explores the evolution and impact of LLMs on question generation, emphasizing their contributions to the field of NLP [66].

1) From rule-based to data-driven approach: Before the era of LLMs, question generation primarily relied on templates and rule-based methods. These techniques effectively generated simple questions but were inadequate in generating relevant and diverse questions. LLMs have adopted a data-driven approach. Their ability to learn complex language patterns and semantics has led to the generation of questions customized to the specific content from which they are derived [67].

2) Contextual understanding and coherence: LLMs can contextualize the input text to generate coherent and relevant outputs, unlike rule-based methods, which often produce disconnected or irrelevant questions. Contextual understanding is critical when generating questions from documents with complex structures, technical language, or nuanced information [68].

3) Fine-tuning for question generation: Fine-tuning involves adapting pre-trained models to specific tasks by training them on question-generation datasets [69]. It allows LLMs to learn the patterns for various contexts, which improves their performance.

4) Challenges and opportunities: LLMs offer great potential in question generation, but challenges exist. Generating clear and concise questions with different levels of complexity and coverage remains an ongoing research challenge [70]. Our research addresses these challenges by introducing evaluation criteria such as clarity, conciseness, and coverage to comprehensively evaluate LLMs in question generation.

2.7.3 Evaluation Metrics for NLP

Evaluating language processing models is critical to NLP research and application development. Effective evaluation metrics allow researchers and practitioners to assess models' performance in various tasks quantitatively and qualitatively [71].

1) The need for evaluation metrics: Evaluation metrics judge how the performance of NLP models is measured. NLP tasks have different aspects and often involve generating or processing human language, making it challenging to assess models’ performance objectively. Metrics provide a structured framework for evaluating models’ output, identifying strengths and weaknesses, tracking progress, and guiding model development [72].

2) NLP evaluation metrics: For NLP evaluation, several widely accepted evaluation metrics have been developed to assess different aspects of model performance. These include clarity, which measures the similarity between generated and reference text, and ROUGE (Recall-Oriented Understudy for Gisting Evaluation) for text summarization tasks [73]. These metrics evaluate the generated text’s specific linguistic qualities.

3) Objective evaluation: Objective metrics can be used to assess the capability of NLP models. For example, clarity provides quantitative scores indicating the clarity between the generated and reference text. Combining metrics like relevance, coherence, and conciseness offers a more

comprehensive understanding of model performance [74]. Our research adopts this set of criteria to assess LLMs' performance in generating questions from program codes.

4) Ethical considerations in metrics: Using evaluation metrics raises ethical concerns. Metrics should be carefully chosen to avoid reinforcing biases or undesirable behaviors in NLP models [75]. Responsible artificial intelligence practices involve developing metrics that encourage fairness and ethical behavior in NLP models. The approach proposed in the current research addresses these ethical concerns while evaluating LLMs' performance and considering issues related to relevance and clarity in question generation. As LLMs become more powerful, ethical considerations have become important. Developing responsible artificial intelligence and mitigating biases in LLMs are critical [76].

2.7.4 State-of-the-art LLMS

Various models have emerged, each showing considerable performance across language processing tasks [65].

1) GPT-4: Building on the success of its predecessors, GPT-4 is known for its language generation ability [77]. GPT-4 exhibits contextual understanding due to its larger model size, improved training techniques, and increased parameters [78]. GPT-40314 has a smaller context capacity than GPT-4-0613. GPT-4 has set a high benchmark for other models in question generation.

2) GPT-3.5: It is the updated version of GPT-3; a later version is 3.5-turbo. It supports 4096 tokens, is free on the web interface, and has a paid API. The capabilities of GPT3.5-turbo-0613 result in better output than GPT-3 for text processing tasks [79].

3) Llama-2: Llama-2 specializes in chat-based interactions and is designed to generate human-like responses [80]. This specialization makes Llama2 a strong candidate for dialogue-based question generation.

4) H2OGPT Variants: The H2OGPT series features fine-tuned variants for specific domains. H2OGPTgm-oasst1-en-2048-falcon-40b and H2OGPT-oasst1-falcon40b offer promising performance for domain-specific applications [81]. These models are customized to generate questions from technical content, which aligns with our research's focus on code-based question generation. Several versions with different parameter sizes are available; all are open-source and can be optimized for specific domains. Each falcon has a distinct parameter capacity or token size [80]. The following is a brief description of each model:

- H2OGPT-gm-oasst1-en-2048-falcon-40b-v1: It has the largest parameter size in open-source models, reaching 40 billion parameters, and the precision of text generation and understanding of NLP is high [82].
- H2OGPT-gm-oasst1-en-2048-falcon-40b-v2: This version is similar to the previous version, as they both trained on the same dataset; however, different personalization settings were added. Additionally, both versions support 2048 tokens [82].
- Falcon-40b-sft-top1-560: This model supports up to 2048 tokens and performs very well in text generation. It was trained on the OSSAT dataset [82].
- H2OGPT-oasst1-falcon-40b: This version is the initial release with 40 billion parameters and supports 2048 tokens. However, the other versions have more refined training data than the initial version [82].
- H2OGPT-gm-oasst1-en-2048-falcon-7b-v3: This model is significantly smaller than the other Falcon models; however, it is also trained on the OSSAT data set, and supports the context length of 2048 [82].

- Falcon-40b-instruct: This model is the newer version of Falcon and uses the same dataset as the previous ones. However, this version is tuned specifically to perform tasks and follow instructions precisely. This version performs better on the required tasks than the previous ones [82].

5) Vicuna-33b: Vicuna-33b focuses on specialized applications [83]. Its model size of 33 billion parameters combines scalability with domain expertise. Vicuna-33b's potential for generating questions in specific technical domains might provide valuable insights into the feasibility of using such models for specialized tasks.

6) Claude: The Claude model is from Google, and it has a huge input token limit that reaches up to 100K user input. Claude performs well on multiple-choice tasks [84]. However, when preparing the paper, this model is only available in the USA and the UK, which is considered an access limitation [85]. The parameter size for this model reaches 130 billion parameters. Furthermore, for text generation, it is stated that it outperforms GPT-3.5, but GPT-4 remains better at prompt understanding and coding [86].

2.8 Evaluation Metrics for Code-Based Generated Questions

Evaluating automatically generated questions is still a problematic issue, and multiple metrics and methods are suggested in the literature. Authors in [87] have proposed a framework to measure the quality of multiple-choice questions that are produced automatically in terms of relevance, clarity, and educational worth. The paper [51] proposes some evaluation measures to gauge the quality and effectiveness of the generated multiple-choice questions (MCQs). These parameters make questions relevant, varied, and appropriate for educational programs. The primary measurement criteria include question relevance score, diversity index, and difficulty alignment accuracy. In another paper [88], the authors mentioned that large language models automatically generate multiple-choice questions in curricula CS0 and CS1. The course outline of both CS0 and CS1 is the core input data into the EduCS system. The paper includes a list of evaluation metrics that will help to evaluate the quality of MCQs provided by the EduCS system. The most relevant aspects of these assessment measures were clarity, relevance, and difficulty level. As a knowledge representation technique [P13], ontology has been used to build semantic models for the Python language [P8], [P9]. The paper [P1] used automatic evaluation measures, BERT-based semantic accuracy, to assess the content. The paper [P3] does not cover automatic evaluation but proposes a hybrid model with human expert evaluations focused on code difficulty and generated question validity. Overall, assessing the quality of machine-generated code-based questions calls for robust metrics beyond conventional automated scoring methods.

2.9 Summary

This chapter examined the intersection of automatic question generation (AQG) and programming education, emphasizing how ontology-driven methods, graph-based code analysis, and large language models (LLMs) contribute to scalable, high-quality assessment systems. The chapter reviewed ontology-based instructional content generation, highlighting its role in structuring and personalizing learning materials for programming education while enhancing content reuse and consistency.

It also explored how static code analysis techniques, particularly Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Program Dependence Graphs (PDGs), provide a structured foundation for analyzing code semantics to inform AQG. The integration of these graph-based representations supports the development of targeted, cognitively diverse programming questions that align with Bloom's Taxonomy, ensuring assessments measure varying levels of cognitive skills.

The chapter further discussed template-based approaches and LLMs like GPT-4 and LLaMA-2, demonstrating their potential to generate coherent, contextually relevant programming questions while acknowledging challenges such as bias, scalability, and the need for robust evaluation frameworks. It highlighted the importance of clear evaluation metrics, including semantic accuracy, relevance, and cognitive alignment, to assess the quality of automatically generated questions effectively.

Overall, the chapter established a comprehensive theoretical foundation for the dissertation, identifying the limitations of current AQG methods in programming education and underscoring the potential of ontology-based and graph-based systems to advance scalable, adaptive, and pedagogically sound assessment frameworks in programming learning environments..

Chapter 3 Ontology-Based Automatic Generation of Learning Materials for Python Programming

3.1 Introduction

Recently, knowledge graphs (KGs), as structured forms of knowledge representation, have gained substantial research interests across academia and industry from modern ontology views. Integrating educational technologies with KGs has an impressive influence on teaching and learning activities, especially in programming with Python. E-learning platforms provide students with tools to easily engage and receive ongoing feedback during the e-learning sessions [8].

KGs are crucial in optimizing the automation of ontology-based learning material generation. They support the organization, interrelation, and knowledge utilization in a particular field [89]. In Python programming, KGs can delineate the existing knowledge, relations, and entities [89]. Additionally, ontology-driven systems support more effective comprehension of the context and relations of various concepts, thus enabling more precise and thorough learning materials generation [89]. Adding KGs to the ontology-based automatic generation of educational materials improves content relevance, personalization, interoperability, content reuse, and efficient knowledge capture [90]. KGs can efficiently organize and manage the structural knowledge of Python programming [90].

In the information age, one's programming capability is required in many professions, as accentuated by the availability of resources aimed at teaching and training in programming [3]. Designing high-quality learning materials for programming languages is difficult and requires substantial resources because of fragmentation in educational programming design, instructional programming expertise, and difficulty in adaptive personalization [5]. Ontology-based automatic learning materials generation (ALMG) leverages advanced educational technologies to streamline this process [12]. This technology will assist educators in saving time and costs by generating particular and appealing materials for students [12]. Calmon et al. [15] describe an automated curriculum selection system that tailors educational content to student needs using machine learning and data analytics, improving learning effectiveness and institutional delivery. Similarly, Xia et al. [21] propose adaptive networked learning material delivery, demonstrating how machine learning can manage learning processes and enhance student outcomes in networking education.

One of the methods to represent domain models is through ontology-based representation [P13]. Semantic understanding and knowledge representation enable ontology-based automatic learning materials generation for Python programming that produces resources like tutorials, code examples, exercises, and assessments. The development of an ontology for capturing Python programming concepts, relationships, and properties is used in this approach. It attempts to create learning materials based on the pedagogical requirements and learning objectives. The ontology-based approach further enables continuously updating and refining the learning materials to sync with Python programming environment changes [91]. Ontology-based automatic learning materials generation for Python programming is a highly efficient and scalable approach using structured knowledge presentation for automating educational content creation [5]. With this method, its learning materials remain consistent, high quality, and personalized, all while allowing for the efficient creation of various resources. Likewise, the existence of the ontologies makes the routines adaptable to changes in Python programming [92], i.e., updating the ontologies and automatically regenerating learning materials. Ontologies' automation saves educators and content creators time and effort and improves a deep semantic understanding of the Python programming domain for a better generation of learning materials [7].

Manual creation of Python programming learning materials remains time-consuming and often fails to keep pace with the ecosystem's rapid evolution [P3]. An ontology-driven automated approach can address these challenges, improving learners’ access to high-quality, adaptive, and contextually relevant resources.

The automatic generation of Python learning materials is critical for ensuring scalability, adaptability, consistency, and accessibility while facilitating innovation in educational technology and programming pedagogy [22]. It enables diverse, personalized learning experiences aligned with learners’ needs and learning styles, supporting educational quality while reducing instructor workload.

This chapter aims to develop a comprehensive ontology for Python programming and design an ontology-based ALMG system tailored to Python education. It outlines the system’s design and implementation while exploring potential enhancements and the implications of such a system in educational contexts.

This chapter details the technologies and methodologies underlying ontology-based ALMG, emphasizing how ontologies capture domain knowledge and facilitate the automated generation of educational content. It discusses the educational and practical implications of ontology-based ALMG, illustrating its potential to enhance Python programming instruction. The objectives of this chapter are to:

1. Design an ontology-based framework that models Python programming concepts and their interconnections.
2. Develop a system for automatically generating Python programming learning materials (specifically quizzes) that align with the modeled concepts and relationships.

The structure of this chapter is as follows: Section 3.2 describes the methodology, outlining the ontology-based approach, domain-specific knowledge modeling, and implementation details, including validation and evaluation of the proposed model. Sections 3.3 and 3.4 present the results and discussion, respectively, while Section 3.5 concludes the chapter, highlighting practical implications. A brief summary follows at the end.

3.2 Methodology

3.2.1 Ontology-Based Approach for Learning Materials Generation

Formal knowledge representation is used in an ontology-based approach that captures domain-specific concepts, relations, and properties and uses such information to generate learning materials. The method involves an ontology for the target domain's concepts, relationships, and properties, such as programming languages. Semantic understanding is captured through ontology, meaning it results in inferring relationships and categorizing concepts. Learners' needs and preferences are analyzed based on educational objectives and learner profiles. The ontology is used to generate content that is coherent and contextually relevant. The materials are presented using natural language processing techniques to make the explanation as clear and understandable as possible. Because it is based on ontology, it allows for continuous updating and refinement as the domain knowledge changes. The benefits include scalability, adaptability, personalization, consistency, efficiency, and accessibility. The ontology-based approach can create adaptive, personalized, high-quality educational content for various domains, such as programming education.

The ontology-based approach for generating learning materials involves structured knowledge representations on a domain to automatically create the learning materials. Ontologies are leveraged in this process to map the relationships between different concepts in the subject of a knowledge domain, providing generated materials that are pedagogically sound and contextually relevant. The primary process of generating learning materials using an ontology-based approach can be demonstrated in several steps as follows:

1. Ontology development, which includes domain analysis, is to identify the key concepts, relationships, and rules within the subject area, and ontology construction to define the concepts (classes), properties (relationships), and instances (individuals) within the domain, and validation and refinement ensure that the ontology accurately represents the domain knowledge through validation and iterative refinement.

2. Knowledge representation involves formalizing the ontology. This formal language provides precise semantics for the concepts and relationships, axioms, and rules to define axioms and inference rules to capture the logical constraints and derivations within the domain.

3. Learning materials generation, which contains the content extraction for identifying relevant content from the ontology based on the learning objectives, content structuring to organize the extracted content into a coherent structure, following educational best practices (e.g., Bloom's taxonomy), and template application to apply predefined templates to format the content into various types of learning materials (e.g., textbooks, task assessments, interactive modules).

4. Automated generation algorithms include the input processing to accept inputs such as learning objectives, target audience, and preferred content format; ontology querying, which uses description logic (DL) queries to retrieve relevant concepts, relationships, and instances from the ontology, material assembly to assemble the retrieved information into structured learning materials using the defined templates, and output generation for producing the final learning materials in the desired format (e.g., HTML, e-learning platform).

Automatically generating learning materials involves a complex pipeline integrating NLP, machine learning, and educational technology. The following is an algorithmic approach to automatically generating learning materials from an ontology. Automatically generating learning materials in the programming domain involves several tailored steps. The following is a proposed general algorithm for automatic learning material generation in the programming domain:

Inputs:

- Programming Language: The specific language (Python).
- Learning Objectives: Skills or concepts to be covered (e.g., syntax, data structures, algorithms).
- Content Sources: Online tutorials, documentation, code repositories.
- Format Preferences: Code snippets, quizzes, text explanation.
- Target Audience: Beginner, intermediate, or advanced learners.

Steps:

1. Content Retrieval:

- Query content sources using APIs or web scraping to gather relevant programming resources.
- Use NLP techniques to filter and categorize content based on relevance and complexity.

2. Content Analysis:

- Analyze the retrieved content for key programming concepts, syntax rules, common pitfalls, and best practices.
- Identify gaps in the content that need to be addressed to fulfill the learning objectives.

3. Content Structuring:

• Organize the content into a logical flow, such as:

• Introduction to the language

• Basic syntax and constructs

• Control structures (loops, conditionals)

• Data structures (arrays, lists, dictionaries)

• Functions and modules

• Advanced topics (e.g., OOP, frameworks)

• Create outlines or flowcharts to visualize the structure.

4. Material Creation:

• Generate text explanations for each section using NLP techniques.

• Create code examples and snippets that illustrate each concept.

• Develop quizzes or coding challenges based on the key concepts identified.

• Design multimedia elements (like screencasts or infographics) if applicable.

5. Customization:

• Tailor the generated materials to fit the target audience's skill level.

• Adjust complexity by simplifying explanations or introducing advanced topics as needed.

6. Interactive Elements:

• Integrate coding environments (like Jupyter Notebooks or online IDEs) where learners can practice coding directly within the material.

• Include live coding demonstrations or interactive simulations.

7. Feedback Loop:

• Incorporate user feedback mechanisms (like quizzes and surveys) to evaluate understanding and engagement.

• Use machine learning to refine content generation based on user performance data.

8. Output Generation:

• Compile all materials into a cohesive format (e.g., HTML pages, PDF documents, online course modules).

• Ensure accessibility standards are met (e.g., code readability, alt text for images).

9. Review and Iteration:

• Implement a review process where educators or experienced programmers can evaluate the generated materials.

• Iterate on the content based on feedback and updates in programming language features or best practices.

Outputs:

• Comprehensive learning materials tailored to programming topics and audiences.

• Code snippets and examples for hands-on practice.

• Quizzes and coding challenges to reinforce learning.

Considerations:

• Ethics and Copyright: Ensure all content respects copyright laws and ethical guidelines.

• Diversity and Inclusion: Include diverse perspectives and examples in the programming context.

• Technology Integration: Consider integrating learning management systems (LMS) or coding platforms for easy distribution and tracking.

Example Use Case:

1. Input:

• Programming Language: "Python"

• Learning Objectives: Understand basic syntax, functions, and data structures.

• Format Preferences: Text explanations, code examples, quizzes.

• Target Audience: Beginners.

2. Output:

• A structured document explaining Python basics with annotated code snippets.

• A set of quizzes covering key points about Python syntax and functions.

• Links to interactive coding environments for practice.

Algorithm 3.1 automatically generates multiple-choice quizzes (MCQs) aligned with Python programming concepts using a domain-specific ontology. It aims to deliver personalized, adaptive, and contextually accurate assessments while ensuring semantic alignment with reference materials through BERT-based similarity checks (implemented and deployed on a Flask App). The process begins by building a domain ontology for Python programming. This ontology formalizes concepts such as data types, control structures, functions, and OOP, capturing relationships and properties necessary for the semantic structuring of learning materials. For each domain concept template, the system uses a template-based generation approach to create relevant MCQs, systematically organizing these questions into a structured MCQ bank. This bank is then saved in a CSV format for efficient retrieval and further processing. When a learner requests a quiz, the system loads the MCQ dataset, filters questions based on the desired difficulty level, randomly selects the required number of questions, computes semantic similarity using BERT embeddings to compare the learner's domain with reference materials, ensuring that the questions are contextually aligned and relevant, and returns the personalized quiz alongside similarity metrics for evaluation and adaptive learning path refinement. This approach enables scalable, automated generation of high-quality, semantically accurate quizzes in programming education, reducing manual effort while enhancing learning personalization and alignment with learning objectives.

3.2.2 Proposed Knowledge Model for The Domain-Specific Concepts

The domain-specific concept is the system's knowledge module, organizing the domain knowledge structure, including its central concepts and their relationships. This model facilitates the automatic generation of learning materials for the educational process. It focuses on constructing and organizing domain-specific concepts and their interrelations [20]. A knowledge module consists of guidelines to identify all vocabulary concepts to illustrate or solve problems. It is purely declarative and does not provide instructions on how learners can utilize it to address practical issues [93].

Algorithm 3.1: Ontology-Based MCQ Generation

```
Input: Domain, Difficulty, Number_of_Questions
Output: Random_MCQ_Quiz, Similarity_Score
PROCEDURE BUILD_PYTHON_ONTOLOGY()
    ontology ← ONTOLOGY_STRUCTURE()
    RETURN ontology
END PROCEDURE
PROCEDURE GENERATE_MCQ_DATASET()
    mcq_bank ← Ø
    for each domain_template do
        questions ← TEMPLATE_BASED_GENERATION(domain_template)
        mcq_bank.ADD(domain, questions)
    end for
    SAVE_TO_CSV(mcq_bank, "mcq_dataset.csv")
END PROCEDURE
PROCEDURE SERVE_QUIZ(domain, difficulty, num_questions)
    questions ← LOAD_FROM_CSV("mcq_dataset.csv")
    filtered ← FILTER_BY_DIFFICULTY(questions[domain], difficulty)
    selected ← RANDOM_SAMPLE(filtered, num_questions)
    similarity ← BERT_SIMILARITY(ontology_material[domain], domain)
    RETURN FLASK_RESPONSE(selected, similarity)
END PROCEDURE
```

Two categories of ontology modules have been developed based on the characteristics of the learning materials: general domain-specific concepts ontology and specific domain-specific concepts knowledge module ontology. These modules represent the knowledge concepts needed for learning, provide input to the knowledge module, offer particular feedback, select problems, create learning materials, and support the student model. A domain-specific concepts knowledge module has been proposed based on current research, as illustrated in Figure 3.1. This model is fundamentally based on domain concepts, properties, task assessments, material resources, learning objectives, learning rules, learning levels, and their interrelationships. To generate learning materials and reuse the knowledge module in the learning process, ontologies organize and represent the domain-specific concepts in the knowledge module. The advantage of this model is its ability to personalize and automatically generate learning materials for learners. Based on the general domain-specific concepts ontology shown in Figure 3.1, domain concepts, domain properties, task assessments, material resources, learning objectives, learning rules, and learning levels terminologies refer to the following:

• Domain concepts present domain-specific knowledge or a comprehensive learning material or course overview.

• Domain properties represent learning material or domain-specific properties within a domain knowledge model.

• Task assessments explain how the application system can assess or measure the required learner activities within a specific period.

• Material resources are physical or digital items used in educational settings to support and facilitate learning. They include textbooks, web resources, software, multimedia tools, and laboratory equipment.

• Learning objectives are clear, measurable goals that outline students' expected learning outcomes. They guide teachers in planning instruction, designing assessments, and evaluating progress. Aligned with curriculum and instructional standards, they provide a framework for effective teaching and assessment.

• Learning rules are principles or guidelines that describe how learning occurs and how new information is acquired and processed. These rules help educators understand student learning and inform instructional strategies while helping students become more effective learners by optimizing their learning processes.

• Learning levels are the stages of proficiency and understanding that individuals progress through as they acquire new knowledge, skills, and competencies. They are crucial in education and instructional design, as they help educators tailor teaching methods and materials to support students at different stages of their learning journey.

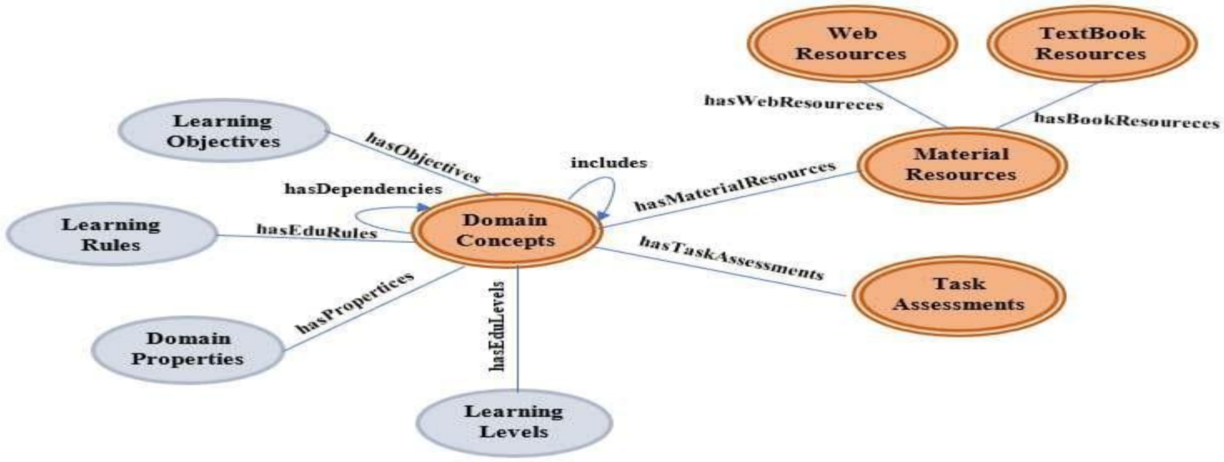

Figure 3.1 Knowledge model for the domain-specific concepts

Figure 3.2 displays the design and structure of a selected ontology knowledge module for the domain-specific concepts case study for the Python programming domain. Several relationships are applied to the domain-specific concepts selected in case examples. The relationships are generalization or specialization, dependency, and containment. Containment indicates that a specific domain concept within a given domain contains various concepts (has-a). The generalization or specialization shows particular topics or domains with specific concepts (is-a). Based on Figure 3.1 and Figure 3.2, the following displays a temporary explanation of a domain concept:

• Domain concepts: Class, Function.

• Domain properties: syntax.

• Task assessments: program, code review, project.

• Material resources: textbooks, web resources.

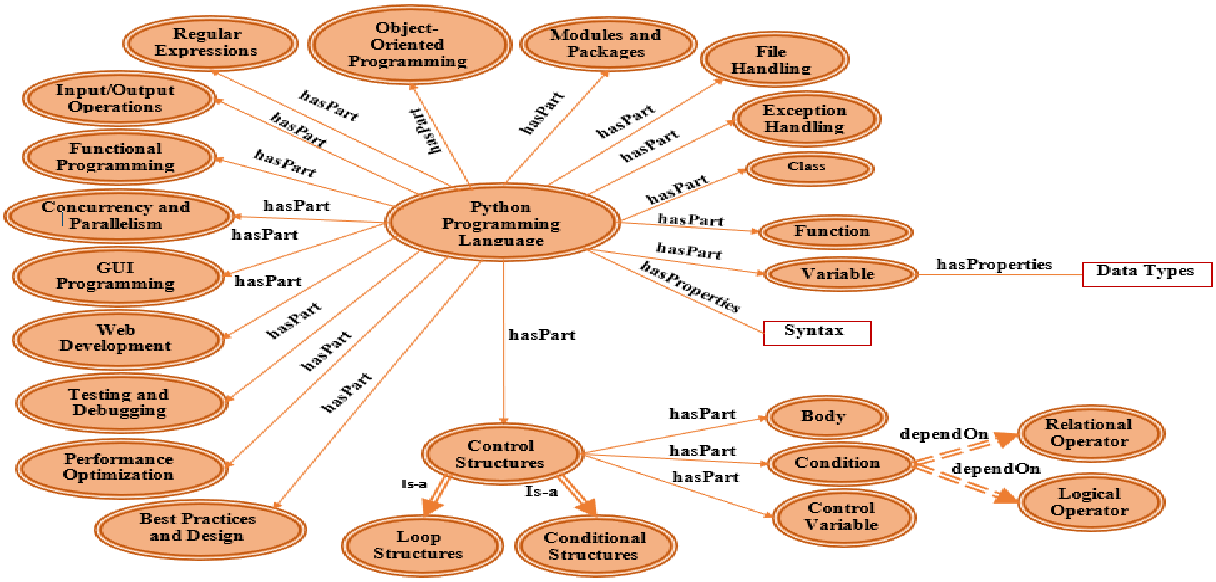

Figure 3.2 Specific knowledge model for the domain-specific concepts

3.2.3 Proposed Model Implementation

Computer Science and Information Technology disciplines offer numerous language modules and packages for developing and managing ontology models. Python is one of the most widely used and favored languages for implementing an ontology for domain-specific concept models. This interpreted, object-oriented, and extensible programming language is known for its exceptional clarity and versatility across various fields [13]. The paper [P8] used Python and Owlready2 to create the ontology and implement the domain knowledge. The domain-specific concept explored in this work is the "Basics of Computer Programming." The ontology is constructed using the "Python Programming Language." The Python and Owlready2 modules implement domain-specific concepts within the ontology. Owlready2 facilitates transparent access to ontologies, allowing for the manipulation of classes, individuals, object properties, data properties, annotations, property domains, ranges, constrained datatypes, disjoints, and class expressions, including intersections, unions, property value restrictions, and more. Python offers some functions and modules for managing ontologies to implement, create, and modify ontologies. The get_ontology() function allows building an empty ontology from its IRI using the Owlready2 module. Owlready2 uses the syntax "with ontology: ..." to demonstrate the ontology that will receive the new RDF triples. For creating an ontology, the following short code is used:

```
from owlready2 import *
ontology = get_ontology()
with ontology: <Python code>
```

Concerning the implementation of the domain-specific concepts and the construction of its components: the domain concepts, learning objectives, domain properties, task assessments, learning rules, material resources, and learning levels. Figure 3.3 shows a code dealing with the design of the core classes of the presented model. Figure 3.4 corresponds with some of the object property relationships defined for the constructed components of the selected model. Several tools are available to display the ontology graph. The tools are Synaptica, OWLGrEd, and Protégé. Protégé is the most commonly used tool to display the ontology graph of domain-specific concepts, as shown in Figure 3.5. The circular relationship lines in Figure 3.5 mean that each topic can depend on another topic and contain subtopics. For example, the iterative loop depends on variables, logical operators, and relational operators. Control sentences contain conditional sentences and iterative sentences. Figure 3.6 presents a SPARQL query as an example of visualizing all the domain concepts in the selected ontology domain-specific concepts regarding retrieving the domain concept and its description.

```
ontology = get_ontology("http://test.org/Domain_Specific_Concepts.owl#")
#Construction of the Domain Specific Concepts Components
with ontology:
    class DomainConcepts(Thing):
        def take():
            print("I take Domain Concepts")
    class LearningObjectives (Thing):
        def take(self):
            print('Learning Objectives')
    class DomainProperties (Thing):
        def take(self):
            print('Domain Properties related to the Domain Concept')
    class TaskAssessments (Thing):
        def take(self):
            print('Task Assessments related to the Domain Concept')
    class LearningRules (Thing):
        def take(self):
            print('Learning Rules related to the Domain Concept')
    class MaterialResources(Thing):
        def take(self):
            print('material resource related to the Domain Concept')
    class TextBookResources(MaterialResources): pass
    class WebResources(MaterialResources): pass
    class LearningLevels (Thing):
        def take(self):
            print('Learning Levels related to the Domain Concept')
```

Figure 3.3 Core classes of the presented model

```
    #The Object Property Relationships added to the Domain Specific Concepts
    class hasPart(DomainConcepts >> DomainConcepts): pass
    class partOf(DomainConcepts >> DomainConcepts):
        inverse = hasPart
    class hasDependency(DomainConcepts >> DomainConcepts): pass
    class dependencyOf(DomainConcepts >> DomainConcepts):
        inverse = hasDependency
    class associate(DomainConcepts >> DomainConcepts): pass
    class associatedBy(DomainConcepts >> DomainConcepts):
        inverse = associate
    class hasParent(DomainConcepts >> DomainConcepts): pass
    class parentOF(DomainConcepts >> DomainConcepts):
        inverse = hasParent
    class hasProperty(DomainConcepts >> DomainProperties): pass
    class propertyOf(DomainProperties >> DomainConcepts):
        inverse = hasProperty
    class hasTask(DomainConcepts >> TaskAssessments):pass
    class taskOf(TaskAssessments >> DomainConcepts):
        inverse = hasTask
    class hasMaterial(DomainConcepts >> MaterialResources): pass
    class materialOf(MaterialResources >> DomainConcepts):
        inverse = hasMaterial
    class hasDRule(DomainConcepts >> LearningRules): pass
    class druleUOf(LearningRules >> DomainConcepts):
        inverse = hasDRule
```

Figure 3.4 Object property relationships

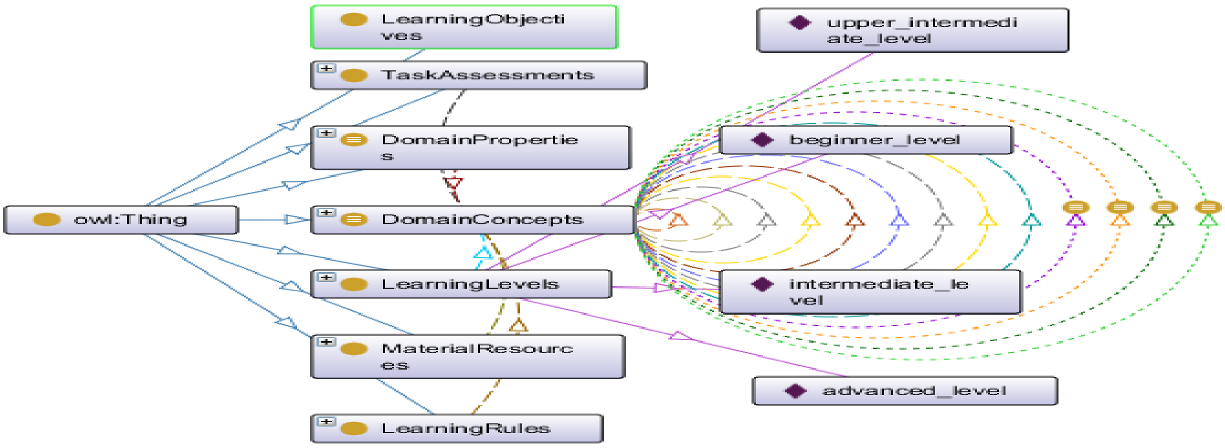

Figure 3.5 Domain-specific concepts ontology graph

```
result = """PREFIX dn: <http://test.org/Domain_Specific_Concepts.owl#>
   SELECT DISTINCT ?domain ?description
                     WHERE {
                     ?d a dn:DomainConcepts;
                     dn:domainName ?domain;
                     dn:domainDescription ?description.
                     }
    """
qres = g.query(result)
for row in qres:
    print('________________________________________________________________________________')
    print(f"Domain Concept: {row.domain}:\nDomain Description:\n{row.description}")
Domain Concept: Python Class:
Domain Description:
A class is a user-defined blueprint or prototype from which objects are created. Classes provide
    a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new
    instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state.
    Class instances can also have methods (defined by their class) for modifying their state.
```

Figure 3.6 A SPARQL query for retrieving the concept "python class" and its description

Natural language processing is used for automatic learning material generation, applying the Spacy module in Python and the rdflib module. Figure 3.7 and Figure 3.8 present the code that controls the ontology of domain-specific concepts. Figure 3.9 and Figure 3.10 display snapshots of SPARQL for generating task assessment and query results according to SPARQL selecting concepts. The results are domain concepts, task assessment, and asking questions in the form of multiple-choice questions. Regarding automatic learning materials generation, the system randomly generates task assessments as multiple-choice questions for the learner. The learner is asked to answer the question, and according to the answer, whether it is correct or not, the system will automatically generate learning materials for further reading. Figure 3.11 shows a snapshot of a task assessment question, whether the answer is correct, and the suggested learning material for the selected task.

```python
import rdflib
import spacy
from spacy.lang.en import English

# Load the English NLP model
nlp = English()

# Load the ontology
g = rdflib.Graph()
g.parse("dataset/update_py_onto_module.owl", format="xml")

# Extract concepts from the ontology
concepts = [str(concept) for concept in g.subjects()]

# Process the concepts using the NLP model
learning_materials = []
for concept in concepts:
    doc = nlp(concept)
    # Generate learning materials based on the processed concept
    definition = f"The term '{concept}' refers to {doc[0].text.lower()}."
    learning_materials.append(definition)

# Print the generated learning materials
for material in learning_materials:
    print(material)
```

Figure 3.7 Controlling the ontology of domain-specific concepts

```
The term 'http://test.org/Domain_Specific_Concepts.owl#c' refers to http://test.org/domain_specific_concepts.owl#c.
The term 'http://test.org/Domain_Specific_Concepts.owl#LearningObjectives' refers to http://test.org/domain_specific_concepts.owl#learningobjectives.
The term 'http://test.org/Domain_Specific_Concepts.owl#task1' refers to http://test.org/domain_specific_concepts.owl#task1.
The term 'http://test.org/Domain_Specific_Concepts.owl#intermediate_level' refers to http://test.org/domain_specific_concepts.owl#intermediate_level.
The term 'http://test.org/Domain_Specific_Concepts.owl#levelDescription' refers to http://test.org/domain_specific_concepts.owl#leveldescription.
The term 'http://test.org/Domain_Specific_Concepts.owl#propertyName' refers to http://test.org/domain_specific_concepts.owl#propertyname.
The term 'http://test.org/Domain_Specific_Concepts.owl#ruleSyntax' refers to http://test.org/domain_specific_concepts.owl#rulesyntax.
The term 'http://test.org/Domain_Specific_Concepts.owl#associatedBy' refers to http://test.org/domain_specific_concepts.owl#associatedby.
The term 'http://test.org/Domain_Specific_Concepts.owl#hasProperty' refers to http://test.org/domain_specific_concepts.owl#hasproperty.
The term 'http://test.org/Domain_Specific_Concepts.owl#taskCatogary' refers to http://test.org/domain_specific_concepts.owl#taskcatogary.
The term 'http://test.org/Domain_Specific_Concepts.owl#ruleID' refers to http://test.org/domain_specific_concepts.owl#ruleid.
```

Figure 3.8 The result of the ontology of domain-specific concepts

```sparql
    SELECT DISTINCT ?domain ?task ?question ?ans1 ?ans2 ?ans3 ?ans4
                    WHERE {
                    ?d a dn:DomainConcepts; dn:hasTask ?t;
                    dn:domainName ?domain.
                    ?t dn:taskName ?task;
                    dn:questionSchema ?question;
                    dn:a ?ans1;
                    dn:b ?ans2;
                    dn:c ?ans3;
                    dn:d ?ans4.
                    }
```

Figure 3.9 Task assessment generation

```
Task: Python classes
Task Question: What is a class in Python?
 a) A module
 b) A function
 c) A template for creating objects
 d) An array
your answer is: c
your answer is correct, because it match the system answer
you can learn more about this domain in the material: Python Classes and Objects  from the following Resources
 https://www.geeksforgeeks.org/python-classes-and-objects/
```

Figure 3.10 Task assessment and result sample

```
Domain Concept: Python Class: Task Assessment: Python classes
Task Questions What is inheritance in Python classes?
a) The process of creating multiple instances of a class
b) The process of passing attributes and methods from one class to another
c) The process of deleting a class object
d) The process of defining new methods in a class

Domain Concept: Python Class: Task Assessment: Python classes
Task Questions What is the output of the following Python code?
                        class MyClass:
                            def __init__(self, value):
                                self.value = value
                        obj = MyClass(10)
                        print(obj.value)

a) 0
b) 10
c) Error
d) None
```

Figure 3.11 MCQs task assessment

Figure 3.12 shows a potential system that uses an ontology-based method to generate adaptive learning materials and quizzes. It illustrates how an ontology of concepts and relationships guides the development of personalized quizzes and learning paths suited to different competence levels. At the same time, learner progress informs knowledge gap analysis and topic selection. The Python programming ontology is a hierarchical system that maps out Python concepts, relationships, and learner progression. It includes fundamental concepts like variables, data types, and functions. The system infers a learner's proficiency level based on how they perform in quizzes and assessments. The ontology can be modified dynamically with performance-related data. In addition, it provides data analytics on tracker progress, predictive analytics, and content optimization. The ontology-based quiz creation process is dynamic and automatic, using Python concepts and learning objectives. It integrates with the learning path generator that selects the questions depending on the learner's progress. The system can accommodate questions such as multiple choice, true/false, fill-in blanks, code snippets, and coding challenges for promoting knowledge retention and skill development. The traditional way of producing materials and questions is to establish the scope and topic sets, acquire information and resources, structure the content, build learning materials, build assessment questions, and create specific examples. The instructor could use book texts, online resources, or even their teaching notes to extensively deal with functions, parameters, return values, and scope. The content is divided into an introduction to the function, a function definition, parameters and arguments, return value, and function scope. Text-based learning materials, code-based learning materials, visuals, and exercises exist. Assessment questions can be multiple choice, code analysis, or code writing. Table 3.1 shows a comparison between traditional vs. ontology-based learning material creation. Examples include defining functions using the `return` statement and questioning about parameters in a function. This approach emphasizes the reliance on the instructor's knowledge and the step-by-step process of translating that knowledge into learning resources. The following is a case study considering the following code:

```
def add_numbers(x, y):
    result = x + y
    return result
sum = add_numbers(5, 3)
print(sum)
```

What is the purpose of parameters in a function?

(a) To give the function a name.

(b) To allow the function to accept input values.

(c) To specify the data type of the return value.

(d) To control the order in which code is executed.

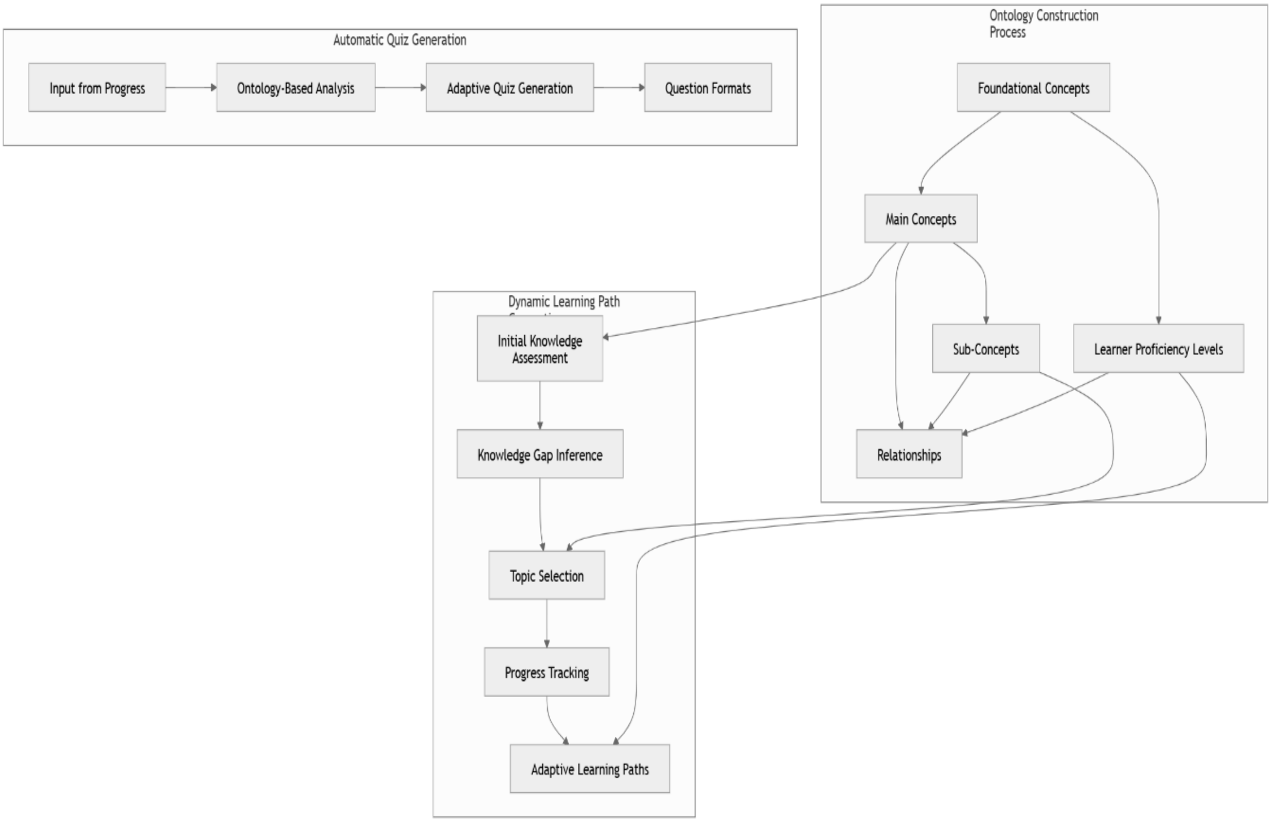

Figure 3.12 Ontology-based method to generate adaptive learning materials and quizzes

Table 3.1 Comparison between the traditional approaches and ontology-based approaches

Feature	Traditional Learning Material Creation	Ontology-Based Learning Material Creation
Content Organization	Linear and structured manually	Hierarchical and dynamically structured using ontology
Customization	Limited personalization	Highly personalized based on learners' needs
Content Reusability	Low content created from scratch	High, modular content reuse across different topics
Automation	Mostly manual work	AI-assisted generation and annotation
Content Consistency	It can be inconsistent across materials	Ensures uniform structure and terminology
Adaptability	Hard to update and adapt	Easily adaptable to new knowledge and learning trends
Efficiency	Time-consuming	Faster and more efficient due to automation
Interactivity	Mostly static content	Dynamic and interactive learning experiences
Scalability	Difficult to scale	Easily scalable across different subjects and learners

3.2.4 Proposed Ontology-Based Model Validation and Evaluation

For ontology-based model validation and evaluation, various tools can be utilized to ensure the ontology's accuracy, consistency, completeness, and pedagogical effectiveness. Using these tools, you can comprehensively validate and evaluate ontology-based models to ensure high-quality, effective learning materials. A robust continuous improvement framework is based on combining automated tools with expert reviews.

1. Ontology Evaluation: Ontology evaluation tools are essential in assessing ontology quality, reliability, and utility in many domains [23]. Ontology quality is measured with several metrics and methods, including quality metrics, consistency checkers, structural analysis tools, domain-specific evaluation tools, and usability evaluation tools [23]. Moreover, these tools also maintain the integrity and usefulness of ontologies across different domains. Automation, usability, interoperability, domain-specific adaptations, and capabilities for dynamic evaluation can be improved [23]. IRI_Debug is an ontology evaluation tool that enables detecting and correcting issues in the Internationalized Resource Identifiers (IRIs) [19]. It provides IRI validation, validation of errors, consistency checking, namespace control, and an easy-to-use interface [19]. However, it is unsatisfactory due to the effectiveness of ontology complexity and IRI usage patterns in ontology development, maintenance, and educational use. Continuous updates are necessary for evolving standards [19]. Owlready2 offers many reasoners for manipulating the domain ontology, such as Pellet, ELK, and HermiT. The HermiT reasoner is used, as shown in Figure 3.13, to check that the constructed ontology is consistent and allows the classification, instance checking, class satisfiability, and conjunctive query answering of the developed domain ontology for the selected model. It is the most commonly used in ontology engineering.

2. Ontology Validation: Ontology validation tools ensure ontologies' quality, reliability, and usability [94]. They identify issues related to consistency, completeness, correctness, and adherence to best practices [94]. Popular tools include OOPS!, OntoQA, OQuaRE, Pellet and Hermit, OntoMetric, BioPortal and AgroPortal, and OntoClean. OOPS! is a tool that helps ontology developers identify and address common pitfalls in ontology design [95]. It uses a set of pitfalls from best practices and expert recommendations, covering naming conventions, ontology structure, and logical inconsistencies [95]. The tool generates detailed reports detailing pitfalls, severity, and affected elements and provides recommendations for correcting each [95]. It can be integrated into ontology environments like Protégé, enhancing usability and promoting best practices [95]. Figure 3.14 shows the OntOlogy Pitfall Scanner tool for ontology validation, which is used for the validation process. The input values for this tool can be ontology URL or RDF file code. Figure 3.15 shows the OntOlogy Pitfall Scanner tool validation results.

```
owlready2.JAVA_EXE = "C:\\Program Files\\Java\\jre-1.8\\bin\\java.exe"
try:
    sync_reasoner()
    print("Ok, the constructed ontology is consistent and allows the classification, instance checking, class satisfiability, and  conjunctive query answe
except OwlReadyInconsistentOntologyError:
    print("The constructed ontology is inconsistent! and didn't allow the classification, instance checking, class satisfiability, and conjunctive query a
```

```
* Owlready2 * Running HermiT...
    C:\Program Files\Java\jre-1.8\bin\java.exe -Xmx2000M -cp C:\Users\jshbo\Python\Python311\Lib\site-packages\owlready2\hermit;C:\Users\jshbo\Python\Pyt
hon311\Lib\site-packages\owlready2\hermit\HermiT.jar org.semanticweb.HermiT.cli.CommandLine -c -O -D -I file:///C:/Users/jshbo/AppData/Local/Temp/tmp_grk
j_bf
Ok, the constructed ontology is consistent and allows the classification, instance checking, class satisfiability, and  conjunctive query answering.
* Owlready2 * HermiT took 1.0713214874267578 seconds
* Owlready * Reparenting Domain_Specific_Concepts.DomainProperties: {owl.Thing} => {Domain_Specific_Concepts.DomainConcepts}
* Owlready * (NB: only changes on entities loaded in Python are shown, other changes are done but not listed)
```

Figure 3.13 Consistency of the domain-specific concepts ontology

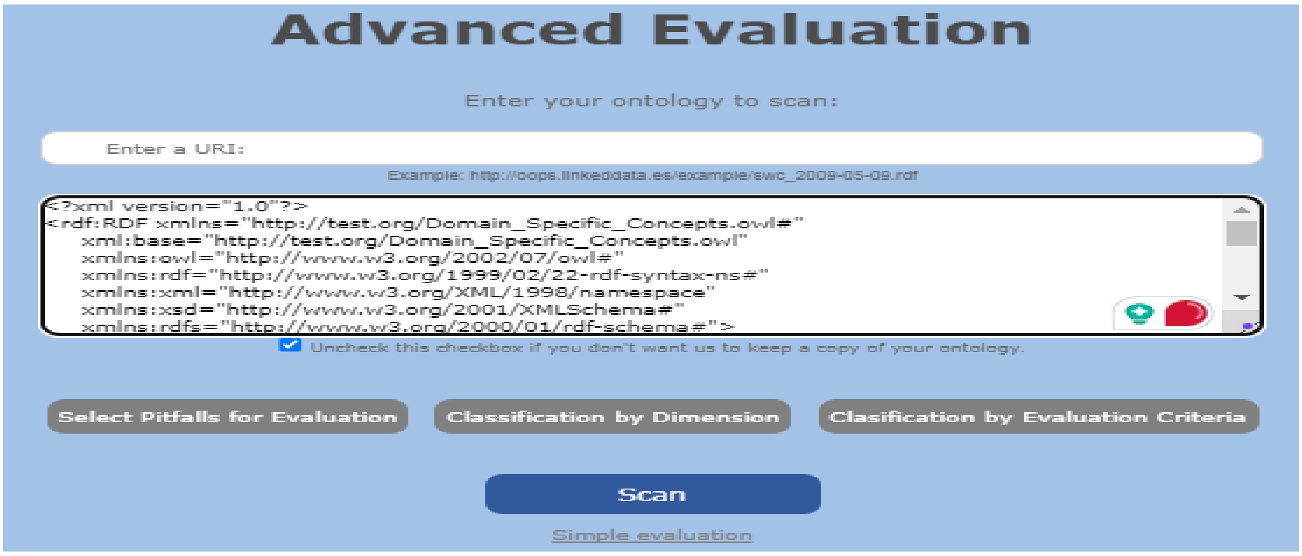

Figure 3.14 OntOlogy pitfall scanner tool

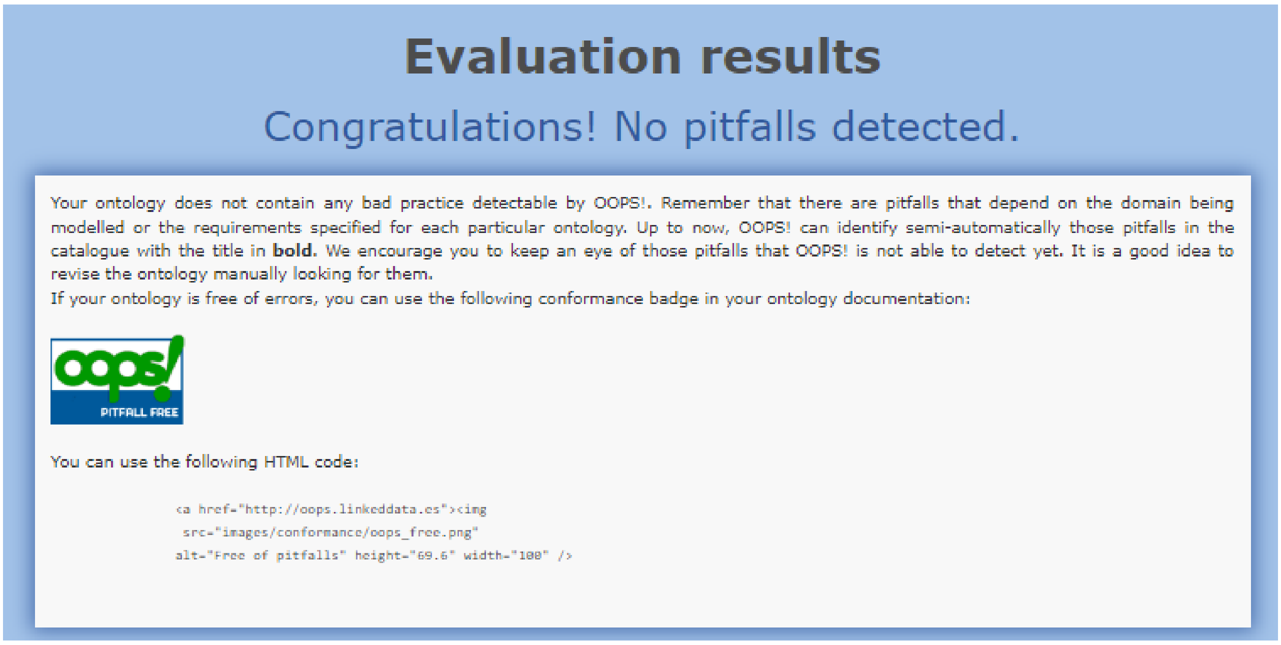

Figure 3.15 OntOlogy pitfall scanner tool results

3.3 Results

The ontology-based automatic generation of learning materials in the Python programming domain as a solution provides a more sophisticated system for generating learning materials. Assessing their quality accuracy, 98.5%, makes it a valuable tool in educational technology and content generation. The dataset used in this experiment is the Python programming language ontology [96]. To generate the learning materials, BERT embeddings have been used to measure the semantic similarity of generated learning materials to predefined reference materials. It also generates an evaluation table, Table 3.2, summarizing the results for each domain concept, as explained in the following steps:

1. Ontology and Learning Materials: An ontology is defined for various domain concepts (e.g., Python Programming, Data Structures), and learning materials are generated for each domain concept using predefined content.

2. BERT-based Accuracy Calculation: BERT model from the sentence-transformers library is used to compute embeddings for the generated learning materials and predefined reference materials. We then calculate the cosine similarity between these embeddings to determine the semantic accuracy of the generated content.

3. MCQ Generation: MCQs are generated for each domain concept and assess how much the learner understands it.

4. Evaluation Table: Table 3.2 shows how the create_evaluation_table function collected generated learning materials, accuracy scores, MCQs, and a brief description of results from the results set into a structured evaluation table with the help of pandas. Descriptions of the accuracy are offered as a categorical measure based upon the thresholds, "Excellent alignment" being the case when the accuracy is greater than 90%, "Good alignment" for anything from 70% to 90%, and "Moderate alignment" for a value that is less than 70%.

Table 3.3 compares the ontology-based model's performance across numerous samples of the Python programming topic Data Types, Control Flow, Functions, Error Handling, and OOP (Object-Oriented Programming), respectively. It shows how effectively the system can generate learning materials and assessments for each topic. As shown in Table 3.4, the ontology-based model's performance also changes according to the dataset size when presented with the task of generating Python programming learning materials. It shows accuracy and other improvements as the model processes more datasets and proves its scalability. Using the following formulas, the evaluation metrics such as accuracy, precision, recall, and F1 Score are calculated by the formulas from 3.1 to 3.4.

$$\text{Accuracy} = (\text{True Positives} + \text{True Negatives}) / (\text{Total Instances}) \tag{3.1}$$

$$\text{Precision} = \text{True Positives} / (\text{True Positives} + \text{False Positives}) \tag{3.2}$$

$$\text{Recall} = \text{True Positives} / (\text{True Positives} + \text{False Negatives}) \tag{3.3}$$

$$\text{F1\_Score} = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall}) \tag{3.4}$$

Data is split into training (80%) and testing (20%) sets using the train_test_split function from sklearn.model_selection. The final parameter is the split with test_size=0.2, and random_state=42 ensures reproducibility. Using dataset size, the training and testing percentages are calculated. The values for these datasets are explicitly defined and printed in the run_evaluation function to make it clear for model training and evaluating the dataset distribution. In this case, the accuracy calculation was measured using the BERT-based semantic similarity. A pre-trained BERT model was used to transform the generated and reference texts into vector embeddings. These embeddings were computed into cosine similarity values measuring their semantic closeness. A predefined threshold was set to verify if the generated content was accurate (e.g., 0.8 or 0.9). The accuracy was calculated as the percentage of correctly matched samples over the total number of samples.

Table 3.2 Evaluation table sample

Domain Concept	**Generated Learning Material**	**Accuracy Score (%)**	**MCQs**	**Description**
Python Programming	Python is a versatile programming language known for its simplicity and readability. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming.	98.50%	Q: What keyword is used to define a function in Python? - def - function - func - define Answer: def	Excellent alignment with reference material.
Data Structures	Common data structures in Python include lists, dictionaries, sets, and tuples. Each structure has unique properties and use cases.	95.85%	Q: Which of the following is an unordered collection in Python? - List - Tuple - Dictionary - String Answer: Dictionary	Excellent alignment with reference material.
Algorithms	Algorithms are step-by-step procedures for solving problems. In Python, you can implement algorithms for sorting, searching, and manipulating data in Python.	92.30%	Q: What is the time complexity of binary search? \n - O(n) \n - O(log n) \n - O(n log n) Answer: O(log n)	Excellent alignment with reference material.

Table 3.3 Ontology-based model evaluation: Python programming topics sample

Python Topic	**Number of examples**	**Percentage**	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Data Types (int, float, str)	390	39%	0.95	0.93	0.96	0.94
Control Flow (if, else, loops)	170	17%	0.91	0.89	0.92	0.90
Functions (def, arguments, return)	70	7%	0.93	0.91	0.94	0.92
Error Handling (try, except)	70	7%	0.89	0.86	0.91	0.88
Object-Oriented Programming (OOP)	360	36%	0.90	0.87	0.92	0.89

Table 3.4 Ontology-based model evaluation performance by dataset size

Dataset Size (Records)	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Small (500)	0.88	0.85	0.89	0.87
Medium (1500)	0.91	0.89	0.92	0.90
Large (5000)	0.985	0.92	0.95	0.93

Finally, the proposed system was deployed using Flask App, as shown in Figure 3.16. The final ontology-driven dataset contained 5,000 structured quiz examples. Each Example consists of a question, four options to choose from as an answer, and the correct answer. This study implements an ontology-driven quiz generation system that leverages structured knowledge representation to enhance Python programming education. By systematically aligning quiz content with formal ontological structures, the system introduces adaptive difficulty mapping and semantic similarity evaluation, ensuring learners engage with contextually relevant and appropriately challenging material. This principled approach differentiates itself from generic quiz generators by providing a structured framework that supports meaningful assessment while maintaining domain specificity. The semantic analysis components refine content alignment and facilitate the generation of quizzes that dynamically reflect the learner's evolving understanding. As part of its future trajectory, the system is designed to incorporate advanced natural language processing (NLP) techniques to enhance semantic alignment and question generation quality, thereby positioning this work at the intersection of structured knowledge representation and adaptive educational technology within the context of programming education.

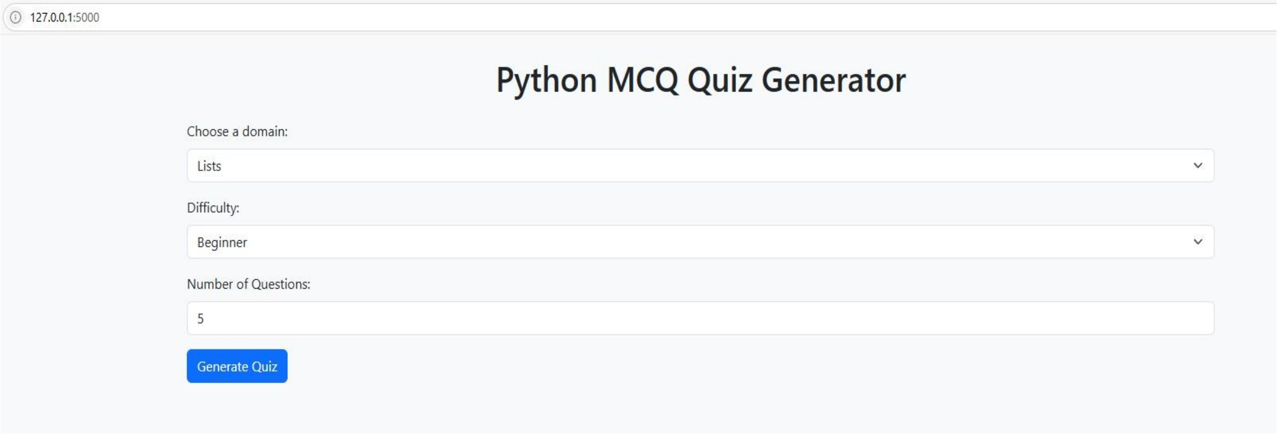

Figure 3.16 Python MCQ quiz generator flask app

3.4 Discussion

Ontology-based automatic generation of learning material is a technology that can potentially enhance learning experiences in almost any educational environment. From an instructor's point of view, it operates as an adaptive tool that can initiate customized tests based on the students' diagnostic results. In this way, it enables the emergence of personalized learning materials directed to certain weak spots and saves quiz creation and grading time.

This tech can provide a personalized learning path for learners, particularly Python programming students. An independent learner might start with a diagnostic test that covers basic topics such as data types, control flow, and functions. It can create debug tasks, discussions, and interactive lessons personalized to the student's needs based on their performance. It can also generate automatic feedback to highlight task errors, syntax errors, and possible solutions for student advancement. The instructor can use the same feedback to identify challenges faced by students and correspondingly grade the difficulty level of exercises so that support may be made more specific.

This technology is excellent for use in both self-paced and instructor-led learning environments. In a blended learning model, for example, a self-paced learner could work through the function modules, and an instructor could give the diagnostic quizzes to track progress. Real-time performance tracking enables educators to identify learning gaps and intervene effectively. Advanced learners can also use the system to focus on specialized topics, such as data manipulation using Pandas, with automatically generated complex coding tasks to support skill advancement.

Overall, the ontology-based approach allows instructors to align learning materials with specific learning objectives, ensuring learners receive contextually relevant, personalized content that enhances engagement and retention while improving instructional efficiency.

3.5 Conclusion

In the digital age, programming skills have become a requisite for practice in almost every professional sphere, increasing the need for the most effective learning materials in programming study and training. Generating educational resources of computer programming based on ontology is a promising way to improve the quality and efficiency of educational resources of computer programming.

This chapter developed and implemented an ontology-based framework to model Python programming concepts and their relationships, enabling the automatic generation of quizzes and learning materials aligned with these structures. Using BERT-based semantic similarity evaluations, the system achieved a high accuracy rate of 98.5%, validating its effectiveness in producing relevant, accurate, and pedagogically coherent content.

The novelty of this approach lies in its integration of structured ontological modeling with automated quiz generation, ensuring adaptive difficulty, semantic relevance, and alignment with instructional objectives in Python programming education.

Despite its contributions, this study acknowledges limitations. First, it primarily focused on Python programming, which may limit the generalizability of findings. Second, it requires further testing through controlled trials comparing ontology-based learning materials with traditional resources to evaluate impacts on retention, engagement, and mastery.

Future research should expand the system to support multi-language programming education, assess its effectiveness through controlled experiments, and integrate adaptive feedback mechanisms and advanced NLP to further enhance question generation quality.

3.6 Summary

Learning materials are essential for effective instruction in programming education. This chapter introduces an ontology-based approach for automatically generating learning materials for Python programming. The method harnesses ontologies to capture domain knowledge and semantic relationships, enabling the creation of personalized, adaptive content. The ontology serves as a knowledge base to identify key concepts and resources and map them to learning objectives aligned with user preferences.

The chapter outlines the design of a dual-module ontology: a general and a specific domain-specific concepts module. This design supports enhanced, tailored learning experiences, enhancing Python education by meeting individual needs and learning styles. The approach also increases the quality and uniformity of generated content, which can be reused for educational reasons. The system ensures alignment with reference materials by using BERT embeddings for a semantic similarity measurement, achieving a quality accuracy of 98.5%. It can be applied to improve Python education by providing personalized recommendations, hints, and problem-solution generation.

Future work will expand this system to support multi-language AQG (Automatic Question Generation), personalized hint generation, and advanced feedback loops, further enhancing programming instruction in scalable and adaptive learning environments.

Thesis 1: An ontology-based system was developed to automatically generate programming-related assessment questions directly from source code. The system enables semantic interpretation of programming constructs using structured domain knowledge, supporting concept-aware question generation without relying on adaptive learning mechanisms. **[P1, P2]**

Chapter 4 A Hybrid Approach for Automatic Question Generation from Program Codes

4.1 Introduction

Automating question generation has become significant with the increasing trend of online learning and its scalability in recent years. Technical courses like learning programming languages are more popular, and there is a massive demand for such subjects. Questions are the primary approach used to evaluate student knowledge [97]. Therefore, creating questions becomes more challenging as the constant growth of e-learning continues, more courses are made, and the pressure on teachers is high. Intelligent and deliberate questions can enhance student understanding and reduce the gap between theory and practice in programming subjects [98]. For example, the article [99] monitors the performance and behavior of students who engage in courses with self-assessment methods in programming and problem-solving. The research in [100] observes the decentralized practice by monitoring the intensity and timing of the impact on student learning and problem-solving in programming languages. The research paper [101] addresses interactivity while solving problems in programming languages based on learning objects. The article [102] tries to enhance the use of digital resources for students and instructors. The research papers [103] and [104] address the learning objects that can be used in different contexts using Web3. Finally, the article [105] suggests collaborative learning to help instructors engage students in generating and evaluating questions. The proposed method in this chapter focuses on translating Python code into text and uses an AI-based framework to generate questions from the text. We also use ontology to connect and conceptualize the logic of the programming language. Applying ontology ensures interoperability with other systems and reduces the overhead on educational platforms. This chapter contributes to e-learning platforms and improves the overall experience of programming language instructors. It also enhances the learning path for students who like to learn and do exercises without repeating the same questions. The outcome of this research is to generate meaningful questions based on Python code to assist instructors in creating more questions in a timely manner, thus ensuring student proper learning of the potential programming language. Unlike similar works, most recent research focuses on generating questions from text, while some research focuses on generating questions from visuals or images [106].

This chapter focuses on generating questions from code snippets using semantic relations to extract the concepts. Generating questions from unconventional sources, such as code snippets, becomes important in providing a better learning experience to large groups of students, especially when dealing with limited information. The main goal of this chapter is to assist instructors and students in properly evaluating student performance by generating Python-based programming questions from existing materials (i.e., code snippets). The automatic question generation from code snippets will add the possibility of generating a different set of questions based on the same code snippet. Therefore, it leads to a better understanding of the given topic. The research objectives of this chapter are to implement a framework that can interpret Python programming language into text, and enable the framework to comprehend the text and build connections between the programming structures and the semantic concepts.

The chapter is structured as follows: Section 4.2 details the methodology and framework. Sections 4.3 and 4.4 present results and discussion, respectively. Section 4.5 concludes the chapter. A brief summary is provided at the end.

4.2 Methodology

Question generation involves computer understanding of the available materials to propose plausible questions to students. However, two approaches are usually effective: AI-based or semantic-based. The current work uses a combination of semantic and AI methods to properly generate questions from code snippets based on semantic code conversion. The primary motivation for using the semantic approach is maintaining concept relations in the programming language keywords to increase system intelligence on the programming language rules. Other approaches would not accurately represent the programming language rules, keywords, and concepts. This section will detail the question generation framework architecture, the technology used, and the approach to generating questions.

4.2.1 Architecture

To generate questions from existing Python code snippets, an interpreter is needed to translate the code into more understandable concepts. Python or any other programming language is constructed using operators, variables, and functions. Operators such as +,-,AND usually do the actual computing. At the same time, variables are used to store values and recall them with operators to perform specific tasks. Functions contain a list of variables, loops, and operators to be executed in order. The ontology will categorize and conceptualize the list of commands (i.e., variables, operators, etc.) and the relationships between the concepts in the script. It will build an explained version of the code by processing the code line by line and creating semantic relationships based on the input. Subsequently, the translated code is generated and inserted into an AI question generator called "QuestGen" [107]. This model will generate open-ended questions. Figure 4.1 shows the framework data flow and its components. Awareness of existing technologies and software is essential to construct any framework or software. Such awareness can improve productivity and help address many issues that take a long time. As a result, we implemented a framework using various third-party software in this chapter. Table 4.1 describes this case's environment settings, tools, and applied libraries. The QuestGen AI model, an open-source natural language processing (NLP) library dedicated to creating simple question-generation methods, has been used. It is on a mission to become the world's most sophisticated question-generation AI by utilizing cutting-edge transformer models like T5, BERT, and OpenAI GPT-2, among others. The primary objective of QuestGen AI is to simplify the question-generation process, providing support to educators, content creators, and learners in developing educational materials. This tool significantly enhances the efficiency of teaching and learning resource development through automation, ultimately facilitating a more effective educational experience.

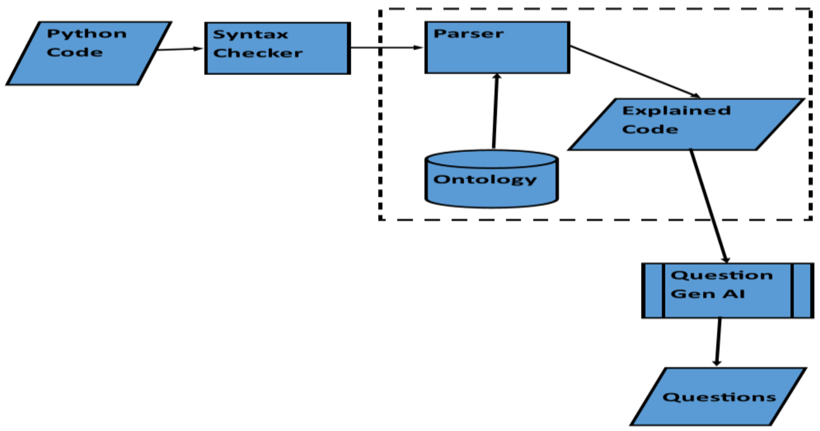

Figure 4.1 Proposed framework architecture

Before generating questions, the QuestGen AI model expects a text as input. The ontology mentioned next is responsible for converting the snippet code from the Python programming language into text that humans can understand. Subsequently, this model can generate questions based on the inserted text. The software supports four types of questions, and they are as follows:

- Questions with Several Choices (MCQs)
- Boolean (Yes/No) Questions
- Open-ended Questions

- Question Paraphrase

The current study considers Boolean, short, and open-ended questions. Since learning a programming language focuses on understanding the content of a code, such questions are more suitable for assessing student knowledge properly.

Table 4.1 Environment settings, tools, and applied libraries

Name	Description
OwlReady2	Python library to implement Ontology V 0.37
Protege	Software Application for viewing and modifying ontology
Jupyter Notebook	IDE to develop the framework
QuestGen	AI-based application to generate questions from the text
Python	V 3.11.1

4.2.2 Ontology Design

The ontology is built and compiled using the OWLReady2 library in Python. Such a library would support automating manual activities like adding instances to the ontology. However, the main components and the relationships between concepts should be implemented manually to maintain logical correctness. Translating code into text starts with assigning keywords to ontology classes and describing these keywords. For example, the "=" sign is described in the ontology as an "equal sign", a value of the Assignment subclass in the operator class. The output of the ontology implemented in Python and OwlReady2 is then imported into Protégé for visualization purposes, since the visualization is not yet supported on OwlReady2. Figure 4.2 shows the visualization of the ontology design in Protégé.

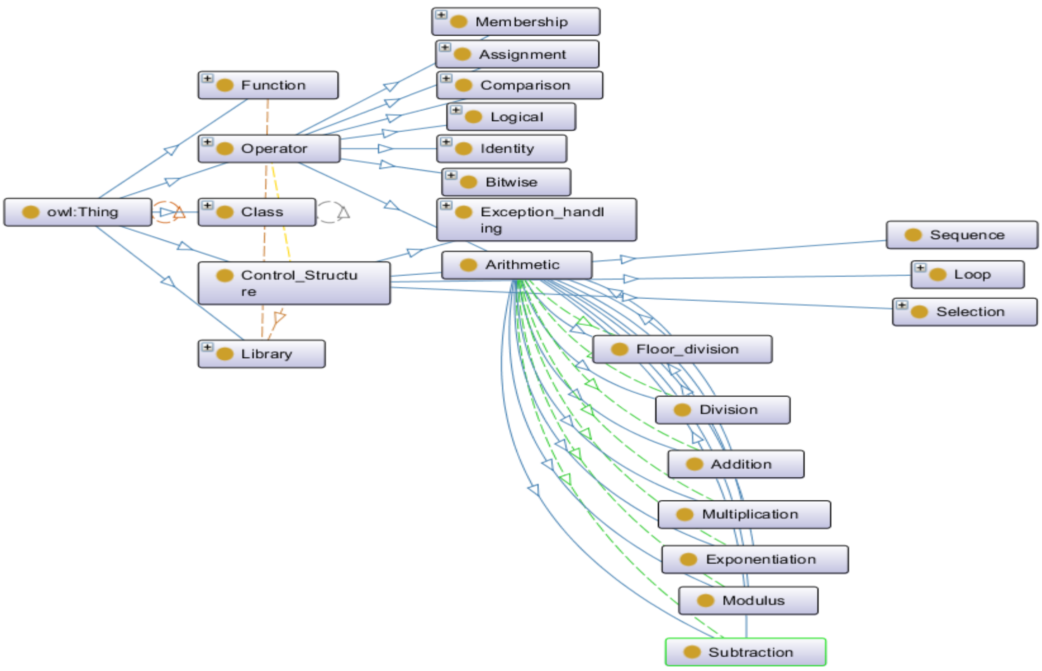

Figure 4.2 Ontology design visualization using protégé

Logical correctness would enforce semantic meaning on the written script. For example, an “elif” statement syntax is valid in Python. However, it cannot exist without having an “if” statement before it. An “elif” should only come after an “if”. Furthermore, logical correctness would connect all the keywords and describe the semantic relationship between steps. Most essential aspects of the Python programming language in the designed ontology are classified as classes and subclasses. For example, in this study, the Python language elements and constructs have been categorized into four main classes: Control Structure, Function, Library, and Operator. Each subclass of the Operator class contains several instances that would map each instance to the operator class. Such mapping would assist in enforcing the logical correctness of the translated snippet. Figure 4.3 shows an instance definition from the constructed ontology. The ontology's capabilities aim to structure the Python programming language to ensure that the computer can collect vocabulary text about the keywords and build sentences based on the combination of the programming language keywords, which can be fed later into the question generation model. The main limitation is that the ontology should be built manually by adding the explanation of all instances, which can be challenging to implement. Further research is needed to improve this approach.

```
<owl:Class rdf:about="#Subtraction">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#has_example"/>
      <owl:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Example
usage of Subtraction</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#has_description"/>
      <owl:hasValue
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Description of
Subtraction</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Arithmetic"/>
</owl:Class>
```

Figure 4.3 Instance definition of Subtraction

4.2.3 Parser

The parser's job is to detach a block of code into pieces that can match the ontology based on keywords and custom conditions. These conditions are adjusted depending on the inserted snippets. This model uses the ontology to create sentences. It analyzes keywords in the parser and generates sentences explaining the code. For example, a=10, the parser would create “a is a variable. a value is 10”. This parser helps turn Python code (and maybe other types later) into sentences using a set of rules. It maintains whatever logic the ontology possesses about the code. Then, it is fed into the AI model to generate proper questions based on the code interpretation by the ontology. Finally, the 'explained code' is passed to the QuestionGenAI framework to generate questions.

4.2.4 Question Generation

Over time, there is a growing demand for question generation, a trend that could significantly alleviate the burden on educators and trainers. This is particularly beneficial for scalable learning formats such as online courses. Many models exist for generating questions from regular text; however, understanding code and generating questions from code snippets is not applied due to its complexity. Code-to-text conversion is a challenging task. However, the semantic relationships between the concepts in the ontology are an excellent solution. Figure 4.4 shows the whole procedure for translating code into text. In Figure 4.4, the code undergoes validation by a parser checker responsible for scrutinizing its syntax. Once the code is confirmed as error-free, the checker directs it to the ontological translator, acting as the parser within our architecture. This parser transforms the code into coherent sentences, forwarding them to the Question Generator AI model to generate reasonable questions. An explanation of the Question Generator AI model is provided in the subsequent section.

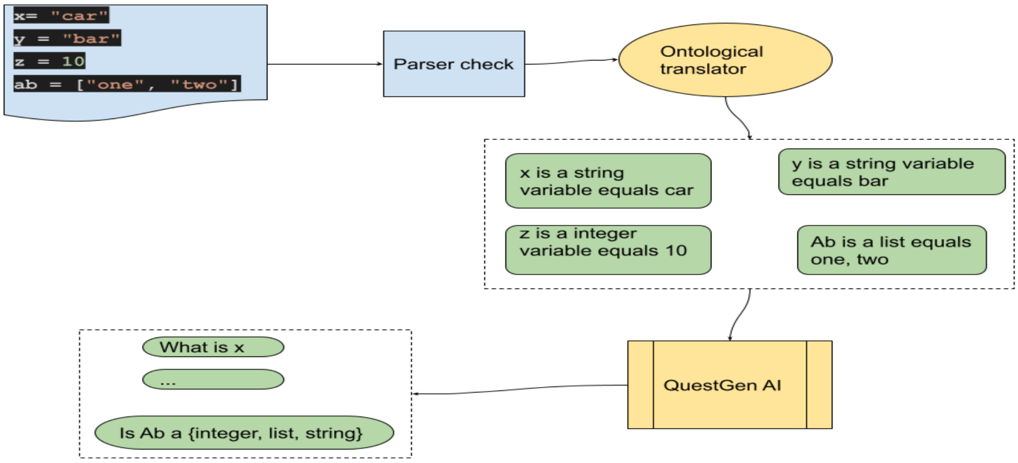

Figure 4.4 Question-generation process

4.2.5 QuestionGen AI

The QuestGen AI model is an AI model that can generate questions using AI. The QuestGen project is available in an open-source format [18]. The model is already trained and can generate high-quality questions based on text fed into the model. Instructors can choose the type of question that can be generated; however, Boolean, short, and open-ended questions have only been applied for this study. The results summarized in the subsequent section show that the AI model can generate reasonable questions based on the input text and its level of clarity.

- Input: The model can process various types of input, including structured, unstructured, and context-based content such as passages, documents, and articles.
- Field of application: The model is tailored to support the education field across diverse disciplines such as science, history, language arts, and more. However, it does not have the capability to execute or generate programming language code.
- Generation method: It is a semantic-based model designed to comprehend inserted text by leveraging concepts and contextual awareness. This procedure is divided into two main steps. Firstly, it begins with entity recognition, wherein the model extracts crucial information such as dates, names, and relationships, employing part-of-speech tagging. Next, the model applies question templates to the extracted information to match the most suitable predefined question template. To improve question quality, various methods are employed, including probabilistic approaches to refine wording and phrasing within the questions.
- Question format: The model can propose various formats, including open-ended, multiple choice, true/false, and short answer.
- Response format: The responses are generated in both text and JSON formats. Each type of question has its own format. For instance, multiple-choice questions prompt the system to produce the question stem and its corresponding answer choices. This distinction applies to all question types, and the resulting output is tailored accordingly.
- Example: The sentence inserted into the model is “In Python, a function is defined using the 'def' keyword, followed by the function name and parentheses containing any parameters. The function body is indented and contains statements that define the function's behavior.”
- The generated questions for a true/false type of question are:
    - “Is a function in Python defined using the 'def' keyword?”.

- “Do parentheses follow the function name in a Python function?”.
- “Does the function body in Python need to be indented?”.

4.2.6 Hybrid Question Generation from Program Codes

Algorithm 4.1 is a hybrid approach employed to automate the generation of programming-related questions from Python source code by integrating structural parsing with ontology-based semantic enrichment. Initially, source code samples are parsed using Python’s abstract syntax tree (AST) to identify constructs such as function definitions, class structures, variable assignments, and control flow statements. An ontology is constructed to represent these extracted elements and their semantic relationships, capturing contextual information regarding code dependencies and logical flow within the program. Using this enriched representation, the system generates diverse question types, including multiple-choice, short-answer, and open-ended questions, through either the Questgen neural generation model or a heuristic fallback mechanism when computational resources are limited.

Algorithm 4.1: Hybrid Approach for Question Generation from Program Codes

```
Input: Python source file path P
Output: Question set Q = {Q_b, Q_s, Q_o}
Parameters: max_questions, question_type
O ← BuildOntology()
C ← ReadFile(P)
AST ← Parse(C)
T ← ∅
for each node ∈ AST do
    switch node.type do
        case Assignment:
            ind ← Variable(node.target, node.value)
        case FunctionDef:
            ind ← Function(node.name, node.args)
        case ClassDef:
            ind ← Class(node.name, node.bases)
        case Call:
            ind ← Object(node.target, node.func)
        case Import, ControlFlow:
            ind ← CreateIndividual(node)
    end switch
    AddToOntology(O, ind)
    semantic_desc ← QueryOntologyRelations(O, ind)
    T ← T ∪ {semantic_desc}
end for
text ← Concatenate(T)
if QuestGen_Available() then
    Q ← QuestGen_AI_Model(text, max_questions, question_type)
else
    Q ← HeuristicFallback(text, max_questions, question_type)
end if
return Q
```

4.3 Results

The results are generated in two versions, one utilizing our proposed model and the other without its use (i.e., by directly inserting the code into the QuestGen AI), as depicted in Figure 4.5. The implemented framework facilitates the question-generation process, empowering teachers to automatically generate Python programming language assessment questions for testing students' knowledge. Figure 4.6 depicts a straightforward code snippet featuring variable definitions. This figure illustrates specific variables alongside their assigned values, incorporated as a script within the ontology. A Python parser is employed to validate the text as proper code before generating any flawed or erroneous questions to mitigate the potential for incorrect syntax within the inserted code. Figure 4.7 displays the translated text derived from the code, providing a textual interpretation for each line. The interpreter presents the variable type and specifies the assigned value for each variable.

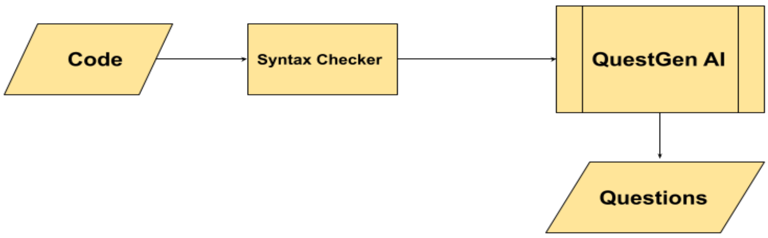

Figure 4.5 Generating questions directly from code

```
# Example Python script to analyze
python_script = """
xfoo = "foo"
ab = ["one", "two"]
cd = ["a boy", 33, "sudden"]
ef = 10
"""
```

Figure 4.6 A code snippet with variable definitions

```
xfoo is a string variable and its value is 'foo'
ab is a list variable and it has 2 items
cd is a list variable and it has 3 items
ef is an integer variable and its value is 10
```

Figure 4.7 Generated text from a code snippet

Figure 4.8 showcases the outcomes resulting from inserting the aforementioned text into the QuestGen AI model. Figure 4.9 can be seen without having a context. The question generator failed to produce any meaningful questions except for the list variable, where it managed to generate a relevant question. However, the AI model could not comprehend all the lines, hence the presence of the ZERO {} symbol.

Figure 4.10 exhibits a Python code comprising class and object definitions presented as a string and passed through an ontology to translate it into text. Subsequently, this text is fed into the QuestionGen model to generate questions. In the subsequent examples, only the generated questions and context from QuestGen AI will be showcased, omitting the complete outputs.

```
Running model for generation
{'questions': [{'Question': 'What is the value of xfoo?', 'Answer': 'foo', 'id': 1, 'context': "xfoo is a string variable an
its value is 'foo'"}]}
{'questions': [{'Answer': 'foo',
            'Question': 'What is the value of xfoo?',
            'context': "xfoo is a string variable and its value is 'foo'",
            'id': 1}],
 'statement': "xfoo is a string variable and its value is 'foo'"}
Running model for generation
{'questions': [{'Question': 'What are the items in the list variable ab?', 'Answer': 'items', 'id': 1, 'context': 'ab is a list
variable and it has 2 items'}]}
{'questions': [{'Answer': 'items',
            'Question': 'What are the items in the list variable ab?',
            'context': 'ab is a list variable and it has 2 items',
            'id': 1}],
 'statement': 'ab is a list variable and it has 2 items'}
Running model for generation
{'questions': [{'Question': 'How many items does cd have?', 'Answer': 'items', 'id': 1, 'context': 'cd is a list variable
and it has 3 items'}]}
{'questions': [{'Answer': 'items',
            'Question': 'How many items does cd have?',
            'context': 'cd is a list variable and it has 3 items',
            'id': 1}],
 'statement': 'cd is a list variable and it has 3 items'}
Running model for generation
{'questions': [{'Question': 'What is the value of ef?', 'Answer': 'value', 'id': 1, 'context': 'ef is an integer variable and
its value is 10'}]}
{'questions': [{'Answer': 'value',
            'Question': 'What is the value of ef?',
            'context': 'ef is an integer variable and its value is 10',
            'id': 1}],
 'statement': 'ef is an integer variable and its value is 10'}
```

Figure 4.8 Generated questions for variable definitions

```
ZERO{}
ZERO{}
Running model for generation
{'questions': [{'Question': 'What is the meaning of cd = "a boy"?', 'Answer': 'boy', 'id': 1, 'context': 'cd = ["a boy", 33,
"sudden"]'}]}
{'questions': [{'Answer': 'boy',
            'Question': 'What is the meaning of cd = "a boy"?',
            'context': 'cd = ["a boy", 33, "sudden"]',
            'id': 1}],
 'statement': 'cd = ["a boy", 33, "sudden"]'}
ZERO{}
```

Figure 4.9 Generated questions without using the proposed approach

```python
# Example Python script to generate explanations
python_script = """
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class Student(Person):
    def __init__(self, name, age, school):
        super().__init__(name, age)
        self.school = school

var1 = Person("Jane", 25)
var2 = Student("John", 20, "ABC School")
"""
```

Figure 4.10 Python code for defining classes and objects

Moving on to Figure 4.11, it explains the preceding code snippet depicted in Figure 4.10 using natural language, preparing it for input into the AI generator.

```
Person is a class definition
  __init__ is a method
    name is an instance of the property
    age is an instance of the property
Student is a class definition
  __init__ is a method
    school is an instance of the property
  Student inherits from the Person class
var1 is an instance of the Person class with name 'Jane' and age 25
var2 is an instance of the Student class with name 'John', age 20, and school 'ABC School'
Student inherits from Person
```

Figure 4.11 Generated explanation of the code in Figure 4.10

Following this, Figure 4.12 displays the questions generated from the snippet description, demonstrating the relevance of the generated questions. However, Figure 4.13 illustrates the outcome of generating questions without providing a snippet description, resulting in improper questions marked by ZERO{} symbols and inaccuracies. This indicates the necessity of providing a description for accurate question generation.

```
[{'Question': 'What is person?, 'context': 'Person is a class definition'}]
[{'Question': 'What is __init__?','context': '__init__ is a method'}]}
[{'Question': 'Name is an instance of a property?','context': 'name is an instance of the property'}]
[{'Question': 'What is age an instance of?','context': 'age is an instance of the property'}]
[{'Question': 'What is a student a class definition?','context': 'Student is a class definition'}]
[{'Question': 'What is __init__?','context': '__init__ is a method'}]
[{'Question': 'What is a school?','context': 'school is an instance of the property'}]
[{'Question': 'What class does a student inherit from?','context': 'Student inherits from the Person
class'}]
[{'Question': 'What is var1 an instance of?','context': "var1 is an instance of the Person class with
name 'Jane' and age 25"}]
[{'Question': "What is the instance of the Student class with name 'John', age 20, and school
'ABC School'?",'context': "var2 is an instance of the Student class with name 'John', age 20, and
school 'ABC School'"}]
[{'Question': 'Who does a student inherit from?','context': 'Student inherits from Person'}]
```

Figure 4.12 Generated questions for the code in Figure 4.10

```
ZERO{}
[{'Question': 'What is the age of the person in def __init__?','context': 'def __init__(self, name, age):'}]
ZERO{}
[{'Question': 'What does age mean?',  'context': 'self.age = age self.age = age'}]
ZERO{}
[{'Question': 'What is the age of the child?','context': 'def __init__(self, name, age, school):'}]
[{'Question': 'What is super().__init__(name, age)?','context': 'super().__init__(name, age)'}]
[{'Question': 'What is self.school?','context': 'self.school = school self.school = school'}]
ZERO{}
[{'Question': 'What is the value of student(John, 20, "ABC School")?','context': 'var2 = Student("John", 20, "ABC
        School")'}]
```

Figure 4.13 Generated questions without using the proposed model

In the third example, depicted in Figure 4.14, a function is defined to compute the area of a circle based on its radius. This code incorporates arithmetic operations and utilizes Python's 'math' module. Subsequently, Figure 4.15 exhibits the output resulting from describing the aforementioned code to input into the AI model. Meanwhile, Figure 4.16 displays the generated questions derived from the description of the code snippet involving mathematical operations. Conversely, Figure 4.17 showcases a question generated without describing the snippet. The results depicted in all figures are formatted in JSON, containing both the question and its solution. For open-ended questions, the QuestGen model provides the answer alongside the question, excluding the options. It is worth noting that there are warnings due to deprecated libraries utilized by the QuestionGen model, prompting necessary updates by the authors. Results indicate that generating questions directly from code without semantic translation yields poor quality, while ontology-based translation enables the generation of meaningful, contextually aligned questions using QuestGen.

```python
# Define the Python code to be analyzed
python_code = """
import math
def area(radius):
    area = math.pi * radius ** 2
    return area
r = 5
a = area(r)
"""
```

Figure 4.14 Code snippet containing a function and arithmetic operations

```
Imported module: math
area_of_circle is a method definition
rd is a variable
  Its value is Constant(value=5)
ar is a variable
  Its value is Call(func=Name(id='area_of_circle', ctx=Load()), args=[Name(id='r', ctx=Load())], keywords=[])
```

Figure 4.15 Generated explanation of the code in Figure 4.14

```
[{'Question': 'What is the name of the module that is imported?', 'context': 'Imported module: math'}]
[{'Question': 'What is a method definition?', 'context': 'area is a method definition'}]
[{'Question': 'What is r?','context': 'r is a variable of type unknown'}]
[{'Question': 'What is Constant(value=5)?','context': 'Its value is Constant(value=5) Its value is Constant(value=5)'}]
[{'Question': 'What is a variable of type unknown?','context': 'a is a variable of type unknown'}]
[{'Question': 'What is the calculated area of the circle?','context': "'a' represents the calculated area of the circle."}]
[{'Question': "What is the value of the call(func=Name(id='area', ctx=Load()), args=[Name(id='r', ctx=Load())?",'context': "Its value is Call(func=Name(id='area', ctx=Load()), args=[Name(id='r', ctx=Load())], keywords=[])"}]
```

Figure 4.16 Generated questions using the proposed model

```
ZERO{}
ZERO{}
[{'Question': 'What is the area of the math.pi * radius?', 'context': 'area = math.pi * radius ** 2'}]
ZERO{}
ZERO{}
ZERO{}
```

Figure 4.17 Generated questions without using the proposed model

4.4 Discussion

In this experiment, various code snippets were tested for translation using the proposed ontology and fed into the QuestionGen model to create open-ended questions. Table 4.2 outlines the test cases, the generated questions, and the difficulty level of the tested code. It is noticed that human evaluation of AQG results is more accurate than automatic assessments [106]. Based on the literature, no evaluation metrics are specific to question generation from source code. The validity of the generated code is rated on a scale of 1 to 5, where one represents the least validity and five indicates the highest validity. Difficulty is assessed based on script logic, with five denoting complexity and one representing simplicity. For instance, identifying variable assignments is relatively straightforward, while understanding inheritance is more challenging. Generating appropriate questions from sophisticated or advanced code snippets, such as those utilizing third-party libraries, still presents limitations. Composing accurate questions becomes increasingly tricky as code complexity and inter-line relationships grow. Consequently, further development is necessary to enhance outcomes. Addressing this need will lead to more advanced results. Nevertheless, this study introduces a new dimension to e-learning and supplements existing question-generation approaches that have proven effective in textual sources.

Table 4.2 Types of syntax covered

Test case	Code level of difficulty	A generated question	Context	Generated question validity
a) Variable declaration	1	What is the value of xfoo?	xfoo is a string variable and its value is 'foo'	4
b) List declaration	2	'What are the items in the list variable ab?	'ab is a list variable and it has 2 items'	5
c) Class declaration	3	What is a person?	Person is a class definition	5
d) Instance and property initialization	4	What is a school an instance of?	'school is an instance of the property'	3
e) Variable initialization, instance initialization, property.	5	'What is var1 an instance of?'	var1 is an instance of the Person class with name 'Jane' and age 25"	4
f) Inheritance identification	5	Who does a student inherit from?	Student inherits from Person	5
g) Libraries import	4	What is the name of the module that is imported?	Imported module: math	4
h) Functions	4	What is a method definition?	area is a method definition	3
i) Variable type	4	What is r?	'r is a variable of type unknown'	4
j) Functions result	5	'What is the calculated area of the circle?	'a' represents the calculated area of the circle.	5

4.5 Conclusion

E-learning has become very popular recently, notably accelerated by the onset of the pandemic. One area that has gained considerable attention among researchers is the automatic generation of questions derived from learning materials. However, the predominant focus of existing efforts lies in generating questions from textual content. This work, however, concentrates on generating questions tailored for Python programming language learners derived explicitly from code snippets found in textbooks and course materials. Leveraging ontologies, this approach demands fewer computational resources, enhancing the scalability of the framework across diverse systems. The proposed framework harnesses ontological mapping, associating each syntactic element with its corresponding meaning and explanation. The process involves translating code into text and subsequently feeding this translated text into an AI-based model for question generation. It aims to alleviate the burden on educators and reduce the repetition of the same questions for different groups of students. Moreover, the generated questions from code snippets serve to evaluate students' general understanding.

However, the proposed approach still has some limitations. The generation of questions relies solely on the QuestGen AI model, which can occasionally result in poorly phrased questions due to its AI nature. Additionally, the model might struggle to identify certain third-party libraries in complex code snippets. Hence, it represents an opportunity for future work to facilitate the insertion and categorization of concepts from all libraries. Finally, exploring alternative models such as GPT and expanding the framework to recursively process all imported libraries would enable a deeper understanding of complex syntactic structures. This enhancement would empower the ontology to explain code snippets better and generate more nuanced and fitting questions.

4.6 Summary

Generating questions is one of the most challenging tasks in the natural language processing discipline. With the significant emergence of electronic educational platforms like e-learning systems and the large scalability achieved with e-learning, there is an increased urge to generate intelligent and deliberate questions to measure students' understanding. Many works have been done in this field using different techniques; however, most approaches work on extracting questions from text. This research developed a model that can conceptualize and generate questions on the Python programming language from program codes. Different models are proposed by inserting text and generating questions; however, the challenge is understanding the concepts in the code snippets and linking them to the lessons so that the model can generate relevant and reasonable questions for students. Therefore, the standards applied to measure the results are the code complexity and question validity regarding the questions. The method used to achieve this goal combines the QuestionGenAi framework and ontology based on semantic code conversion. The results produced are questions based on the code snippets provided. The evaluation criteria were code complexity and question validity. This work has great potential for improving the e-learning platforms to improve the overall experience for both learners and instructors.

Thesis 2: A hybrid system was developed that combines static code analysis and natural language processing using word embeddings to generate programming-related questions from source code. This approach improves contextual variety and semantic relevance by linking syntactic structures with conceptual representations. **[P3]**

Chapter 5 Evaluating Large Language Models for Code-Based Question Generation in Programming Education

5.1 Introduction

The field of natural language processing (NLP) has seen major progress in recent years, attributable mostly to the ever-growing corpus of textual data and remarkable developments in language modeling. These large language models have become the basis of the NLP revolution. They have shown abilities in comprehending and producing human language that have attracted researchers and developers. Large language models (LLMs) have played an important role in NLP. Models such as GPT-3.5, GPT-4, Llamas, Falcon, and Vicuna impact fields beyond NLP, such as code generation and understanding. Each model has a particular set of characteristics regarding performance, response quality, and efficiency in addressing different tasks. The number and complexity of datasets used in language modeling have recently increased. Language models have been the basis for NLP progress because of their language understanding and generation capabilities.

In the domain of coding and software development, the computational capacity of these models has been used to automate the process of generating code-related questions. Consider a script written in a programming language like Python. This script is considered input to these large language models through an API connection. The output would be a collection of relevant questions about the input (e.g., Python script). This functionality speeds up code evaluation and is important for tasks such as code review, online technical support, and programming education.

The large number of accessible language models creates a challenge. With all these options available, comparing them in terms of performance and output quality is necessary. The present study addresses this challenge by conducting a comparative evaluation of popular large language models. This study proposes a set of evaluation criteria to systematically assess and benchmark the performance of these models. These criteria represent essential aspects, including relevance, clarity and coherence, conciseness, and coverage. Every aspect has been examined to assess the performance of the large language models under investigation. This study evaluates these models, clarifying their distinctive characteristics and shortcomings. This study seeks to uncover insights that may be vital in various applications. Highlighting these best performers would allow educators, developers, and researchers to make informed decisions about adopting large language models for code-related question generation tasks. The study evaluates a diverse set of state-of-the-art LLMs. The primary objectives of this research paper are as follows:

1. To define a set of evaluation criteria, including relevance, clarity and coherence, conciseness, and coverage, to measure the quality of questions generated by LLMs.
2. To develop an approach for evaluating and comparing the performance of LLMs in question generation from program codes (code-based question generation).
3. To empirically evaluate and rank the selected LLMs based on their performance in question generation from program codes (code-based question generation).

This chapter is structured as follows. Section 5.2 outlines the methodology and describes the dataset used for evaluation. Section 5.3 provides a detailed account of the experimental setup. Section 5.4 presents the evaluation results along with the ranking of the large language models. Section 5.5 discusses the findings and explores the potential applications of large language models in question generation from program code. Section 5.6 concludes the chapter. A brief summary is provided at the end of the chapter.

5.2 Methodology

The methodology section in this chapter describes the proposed approach to evaluate and compare the performance of various large language models (LLMs) in generating questions from program codes. It presents the research phases, including data collection and preparation, the selection of

LLMs, evaluation metrics, execution of experiments, and the criteria for model ranking. This methodological framework is proposed to ensure a thorough assessment of the models and to reveal the top-performing models in question generation from program codes. As input to the models, Python, C++, and Java scripts are chosen in the experiments. This research would also be suitable for other programming languages. By providing a clear and structured methodology, the aim is to contribute to the existing research by understanding the strengths and weaknesses of these LLMs and their applicability in generating questions, thereby paving the way for more informed decision-making in real-world applications where such capabilities are crucial. Similar studies have been carried out previously by [108], [109], and [110]. Algorithm 5.1 shows the pipeline of the proposed framework. It compares LLMs on how well they generate questions about code, using a reference evaluator model, and produce quantitative metrics. Given a set of code samples, each model generates questions for each sample using a consistent prompting strategy. A reference model then evaluates these generated questions to assess their quality along dimensions such as relevance and clarity. The algorithm computes the average score for each model and optionally tracks repetition rates to measure question diversity. It further constructs pairwise win matrices, computes win rates, and calculates ELO ratings to rank models based on relative performance. The outputs, including average scores, win rates, ELO ratings, repetition rates, and comparison matrices, are then summarized.

```
Algorithm 5.1: Multi-Model Code Question Generation and Evaluation
Input: Set of Code Samples (D), List of LLM Model Names (MODELS), Reference Evaluation Model (EVAL_MODEL)
Output: Summary of Model Performance Metrics (SMPM)
1: Initialize scores_by_model, reps_by_model, results as empty.
2: For each sample in D do:
    3: For each model_name in MODELS do:
      4: prompt ← build_generation_prompt(sample.code, sample.language)
      5: questions ← LLM(model_name).generate_questions(prompt)
      6: metrics ← evaluate_questions(questions, EVAL_MODEL)
      7: score ← average_scores(metrics)
      8: repetition ← repetition_rate(questions) // optional
      9: Store (model_name, sample, metrics) in results
      10: Append score to scores_by_model[model_name]
      11: Append repetition to reps_by_model[model_name]
    12: End For
13: End For
14: wins, comparisons ← build_win_matrix(scores_by_model)
15: win_rate ← win_rates(wins, comparisons)
16: elo ← elo_ratings(scores_by_model)
17: repetition ← aggregate_repetition(reps_by_model)
18: Construct SMPM as {ranking(scores_by_model), win_rate, elo, repetition, wins, comparisons}
```

5.2.1 Data Collection

The methodology employed in this research involves using a diverse dataset of Python, C++, and Java code snippets, which was prepared previously to cover a wide range of these languages' syntax. We utilized our software tool to have each LLM generate questions based on these code samples. The generated questions are then assessed against the predetermined evaluation criteria, and the models are ranked according to their overall performance.

1) Selection of large language models: The research commenced with meticulously selecting large language models (LLMs) for inclusion in the evaluation. The chosen LLMs represent a diverse spectrum of model sizes, architectures, and capabilities, ranging from smaller, established models to novel and expansive ones. This diverse selection is crucial for thoroughly examining the LLMs'

proficiency in question generation from program codes (e.g., Python scripts). We chose the following LLMs for evaluation:

1) gpt-4-0314
2) Llama-2-70b-chat-hf
3) gpt-4-0613
4) Llama-2-13b-chat-hf
5) Claude-2
6) gpt-3.5-turbo-0613
7) h2ogpt-gm-oasst1-en-2048-falcon-40b-v1
8) h2ogpt-gm-oasst1-en-2048-falcon-40b-v2
9) vicuna-33b-v1.3
10) falcon-40b-sft-top1-560
11) h2ogpt-research-oasst1-llama-65b
12) mixtral-8x7b-instruct-v0.1
13) h2ogpt-gm-oasst1-en-2048-falcon-7b
14) h2ogpt-gm-oasst1-en-2048-falcon-7b-v3
15) falcon-40b-instruct

These models were selected to encompass various sizes, ensuring a comprehensive performance evaluation. Table 5.1 shows the availability of each model and the number of parameters it has. All the models are based on transformer architecture; therefore, we did not mention the architecture in the table.

Table 5.1 Selected large language models

Model	Parameters	Availability
gpt-4-0314	175B	Paid
llama-2-70b-chat	70B	Free
gpt-4-0613	175B	Paid
llama-2-13b-chat	13B	Free
claude-2	130B	Paid
gpt-3.5-turbo-0613	175B	Paid
falcon-40b-v1	40B	Free
falcon-40b-v2	40B	Free
vicuna-33b-v1.3	33B	Free
llama-65b	65B	Free
falcon-40b-sft-top1-560	40B	Free
mixtral-8x7b-instruct-v0.1	56B	Free
falcon-7b-v3	7B	Free
falcon-40b-instruct	40B	Free
falcon-7b	7B	Free

2) Data Preparation: The heart of the experiment lies in the quality and diversity of the data used. A curated set of Python, C++, and Java scripts prepared covering an array of programming concepts, complexities, and domains. We used three programs: procedural, object-oriented, and general. The general code was taken from online sources. The two other codes were prepared during some classes taught at the university. In these programs, we collected the diversity of programming elements so that all basic topics (from the Python/C++/Java language reference) are represented. This set of scripts was meticulously vetted to ensure its relevance, representativeness, and suitability for evaluating the LLMs' question-generation capabilities.

5.2.2 Question Generation

The next phase involved instructing each one of the selected LLMs to generate a diverse set of questions based on the attached scripts. This process required the formulation of a carefully crafted prompt, which was used as input for each LLM. All the models used the same role and content to get measurable results. The prompt served as a crucial communication channel between the software and the models, guiding them to generate questions relevant to the script provided. The entire script was passed to each of the abovementioned LLMs as part of the prompt. The models were instructed to generate diverse questions based on the attached script. The prompt utilized for generating the question set is given in Figure 5.1. It was designed to be informative and specific, conveying the task of clearly generating questions from the Python/C++/Java script to the LLMs. Figure 5.2 shows an example of responses to the presented prompt.

```
[{"role": "system",
  "content": "you are an experienced software engineer having expertise in Python"},

{"role": "user",
  "content": "Produce a set of questions that thoroughly examine the Python script.\
    These questions should address different sections, variables, and logical flows\
      within the code to ensure comprehensive coverage."}]
```

Figure 5.1 Prompt to generate questions from source code

```
"questions": [
      "How many iterations does the while loop have in total?",
      "What is the final value of 'num' after the loop exits?",
      "What is the purpose of the 'factorial' function?",
      "How does the 'factorial' function calculate the factorial?",
      "What is the purpose of the 'num' variable?",
      "Explain the purpose of the 'while' loop in the 'factorial' function.",
      "How are the values of 'num' and 'n' updated in each iteration of the 'while' loop?",
]
```

Figure 5.2 Response to a prompt

A Python script, taken from the prepared collection, was provided as input to each LLM as part of the prompt. Figure 5.3 shows an example of another Python script. Each script in our dataset was processed sequentially, and the LLMs were prompted to generate 50 questions based on each attached script. The scripts are publicly available on GitHub [111].

As the questions were generated, they were associated with the script from which they were derived. This association was needed in the evaluation process as it allowed us to accurately assess the generated questions' relevance to the script content. Questions were considered relevant if they indicated a clear contextual connection to the associated script, contributing to understanding its content.

Combining diverse LLMs and carefully curated scripts forms the basis for systematically evaluating these models in generating questions. The methodology emphasizes the importance of this data-driven approach in ensuring meaningful and insightful results. This intermediary step in the solution provided a substantial collection of questions generated by each LLM for every script. The generated questions were then subjected to a comprehensive evaluation process explained in the subsequent sections of this methodology.

```
def factorial(n):
    num = 1
    while n >= 1:
        num = num * n
        n = n - 1
    return num

f = factorial(5)
print(f)
```

Figure 5.3 Sample Python script

5.2.3 Performance Metrics

The next step in the approach was to assess and compare the performance of the selected LLMs. The assessment goes by analyzing and evaluating the generated questions. To achieve this, a multifaceted set of evaluation techniques was used, which included a hybrid of objective and subjective metrics comprising the following aspects:

1) Relevance: To evaluate the relevance of the generated questions, GPT-4-0314 was used as an A/B tester. It rated the LLM-generated questions on a scale of 1 to 10, considering how well they related to the content of the codes.

2) Clarity and Coherence: The clarity and coherence of the generated questions were assessed by obtaining ratings from the GPT-4-0314 as an A/B tester. LLM-generated questions were rated on their ability to convey ideas clearly and logically.

3) Conciseness: Another important dimension of the evaluation was the conciseness of the questions. This aspect was measured by assessing the length and verbosity of the generated questions to identify concise and to-the-point questions.

4) Coverage: To determine how LLMs address the content of scripts, the coverage of different aspects of the scripts within the generated questions was measured. This involved comparing how well the LLMs addressed various sections and script details.

5) Leveraging GPT-4-0314 as an A/B Tester: To gain deep insights into aspects like clarity, conciseness, and coverage, an in-depth analysis of the generated questions was performed using

"GPT-4-0314." This allowed a better understanding of the specific strengths and weaknesses of the LLMs when generating questions.

6) Human Evaluation: It offers indispensable value to automated evaluation by validating core findings while enriching the analysis with pedagogically grounded insights. The convergence between algorithmic educational scoring and expert human judgment reinforces the credibility of computational methods in educational research. However, human evaluators contribute an irreplaceable dimension: a sensitivity to real-world classroom relevance and instructional applicability. This perspective, often absent from purely algorithm-driven metrics, underscores the importance of adopting a holistic, multi-layered framework for evaluating question quality in educational technology. Relevance and Educational Value are used for this purpose.

In summary, the research employed a robust methodology involving data collection, a diverse selection of LLMs, comprehensive performance metrics, custom software development, and leveraging the SOTA GPT-4-0314 model to evaluate and compare the performance of LLMs in generating questions related to the codes. The results provide valuable insights into the strengths and weaknesses of various LLMs, aiding in selecting the most suitable models for different application scenarios, which are shared in the next sections. Based on the input shown in Figure 5.3, some of the generated questions and their evaluations are shown in Figure 5.4.

```
"question": "How many iterations does the while loop have in total?",
"criteria_scores": {
    "Relevance": 9,
    "Clarity": 9,
    "Coherence": 9,
    "Conciseness": 9,
    "Coverage": 8,
    "Average": 8.8
}

"question": "What is the final value of 'num' after the loop exits?",
"criteria_scores": {
    "Relevance": 8,
    "Clarity": 9,
    "Coherence": 9,
    "Conciseness": 8,
    "Coverage": 8,
    "Average": 8.4
```

Figure 5.4 Evaluation of the generated questions

5.2.4 Experimental Setup

This section provides a detailed description of the experimental setup employed for evaluating the performance of the selected models in generating questions from codes. The objective of this setup was to get a collection of reliable results that would facilitate the comparison of LLMs and the identification of the top-performing models. A custom software was developed to serve this purpose. This software accepts program codes as input, invokes the selected LLMs via API calls, and collects the generated questions. For each LLM, the software collected a substantial sample of questions for analysis.

5.2.4.1 Software Environment

The software environment was configured with the following components:

- Operating System: Windows 10 Pro distribution to provide a stable and efficient computing environment.

- Python: The programming language to implement the custom software tool that interfaces with the LLMs.
- Deep Learning Frameworks: PyTorch 2.1 and Hugging Face v3 Transformers library were employed for managing and interfacing with the LLMs.
- API: Different APIs were used for every model.
- AWS Instances: Different AWS instances were used to deploy open-source LLMs.

5.2.4.2 Data Splitting

To ensure the robustness and reliability of the experiments, a collection of code scripts was submitted at once to provide context to the model and, therefore, assist in generating more robust questions. Thereafter, the LLMs were instructed to generate questions based on the input.

5.2.4.3 Evaluation Metrics

The LLM-generated questions were evaluated using a combination of quantitative and qualitative metrics. As mentioned in the methodology section, these metrics include relevance, clarity and coherence, conciseness, and coverage. While the human evaluation metrics include relevance and educational value.

5.2.4.4 Model Execution

Execution of the experiments was a systematic approach. Each LLM was fed scripts individually as prompts through the custom software. The LLMs generated a set of questions for each script, which were recorded. The generated questions were associated with their script for accurate evaluation. The experiments were executed sequentially for all selected LLMs to maintain consistency and avoid potential bias that may arise from parallel execution.

5.2.4.5 Model Ranking Criteria

The model ranking criteria were established based on the aggregated performance results across the evaluation metrics. The models that showed high performance across these criteria were identified as the top-performing LLMs for the task of generating questions from codes (code-based question generation). This experimental setup was designed to provide a reliable and comprehensive assessment of LLMs' capabilities in question generation from program codes.

5.2.4.6 Repetition Rate

This criterion determines if questions are repeated in any model based on each 10-question batch increase. For instance, each model is required to generate the first 10 questions, then 20, then 30, and so on. The goal is to calculate the repeated questions generated for each model. This calculation is done manually by searching through the questions. The automatic evaluation for this part is avoided because some of the questions may be paraphrased.

5.3 Results

This section presents the results of the comprehensive evaluation of various LLMs for the task of generating questions from program codes. The evaluation encompassed a diverse set of metrics, including relevance, clarity and coherence, conciseness, and coverage. Based on the accumulated data and the aforementioned evaluation criteria, the LLMs have been ranked, highlighting their strengths and weaknesses in question generation.

5.3.1 Model Rankings

Table 5.2 presents the average of each criterion score for each model of the LLMs based on the question generated.

Table 5.2 Average criteria scores

Model	Relevance	Clarity and Coherence	Conciseness	Coverage
gpt-4-0314	9.85	8.87	8.13	8.57
gpt-4-0613	8.46	8.23	8.80	9.22
gpt-3.5-turbo-0613	9.37	7.84	8.69	7.61
claude-2	7.86	7.97	8.80	7.96
falcon-7b-v3	8.45	8.52	8.26	7.32
vicuna-33b-v1.3	8.84	8.04	7.51	7.88
falcon-40b-v2	7.93	8.38	7.59	7.65
llama-2-13b-chat	7.69	8.22	7.63	8.14
llama-2-70b-chat	7.76	7.71	6.27	7.60
mixtral-8x7b-instruct-v0.1	6.51	6.55	7.62	7.46
falcon-40b-v1	6.63	7.53	6.68	6.36
falcon-40b-sft-top1-560	7.51	7.88	6.54	7.29
llama-65b	7.45	6.85	7.54	7.53
falcon-7b	7.23	7.83	6.83	7.76
falcon-40b-instruct	7.12	8.03	6.83	7.58

The model average score is based on the cumulative scores of each criterion across all the questions, with higher ratings indicating superior accuracy in question generation from scripts. The rankings reveal that “gpt-4-0314” and “gpt4-0613” secured the first and second positions, respectively, demonstrating their effectiveness and proficiency in generating relevant, high-quality questions.

Furthermore, to gain a comprehensive perspective on the performance of the LLMs under evaluation, their Average Win Rate was analyzed against all other models. The term Win Rate is used to identify the cumulative score for each model and helps determine the best model. For example, if a question is generated by gpt-4-0314 model and compared to the Claude-2 model, and the winner for that particular question is gpt-4-0314, this would add a point to the gpt-4-0314 model. Then, gpt-4-0314 is compared to other models; if any model wins a point, its score grows, and then finally, all the models’ scores are calculated, and the highest winner is ranked first. The approach allows us to identify models that have similar win rates to other models. This analysis offers valuable insights into how each LLM fared directly compared to its peers, assuming uniform sampling and no ties in the evaluation metrics. The chart presented in Figure 5.5 illustrates the results of this assessment, highlighting the models that consistently outperformed others in generating questions from codes. The following formulas, 5.1 and 5.2, would calculate the New Rating and the Predicted Rating, respectively [112]. This technique is used here for the artificial intelligence evaluation domain; it is derived from tournaments in sports, where it is often used.

$$\text{New Rating} = \text{Old Rating} + \mathrm{K} \times (\mathrm{W} - \mathrm{P}) \tag{5.1}$$

Where K refers to the maximum adjusted value, in this context, it is a constant integer number like 32. The W is the actual result of the game (1 for a win, 0.5 for a draw, and 0 for a loss). Finally, the P is the expected result, calculated using the logistic function in equation 5.2.

$$P = \frac{1}{1 + 10^{\frac{(Mo - Mp)}{\text{score point}}}} \quad (5.2)$$

P refers to the expected outcome for a certain model, M_o is for the model opponent, and M_p refers to the model player. The traditional constants in this formula are the score points, 1 and 10, as those are customized. The score point in this context is 400. These two equations form the core of the Elo rating system, originally developed by Arpad Elo to provide a fair and dynamic method of ranking chess players based on match outcomes. The Elo system has been widely adopted in various competitive domains beyond chess, including online gaming, sports tournaments, and AI benchmarking, due to its simplicity and effectiveness in capturing relative skill levels. The second equation computes the expected probability of a player winning against an opponent based on their rating difference, while the first updates the player's rating after each match according to the actual outcome and the expected result. This combination allows the system to adjust ratings to reward unexpected wins and penalize unexpected losses, ensuring ratings remain reflective of current performance.

The "Average Win Rate" metric provides a clear and quantitative view of the models' relative strengths, highlighting their competitiveness in question generation. Figure 5.5 illustrates the Average Win Rate of each language model against all other models in the evaluation, assuming uniform sampling and no ties. The Average Win Rate is a valuable metric for understanding how each LLM performed directly compared to its peers in generating questions from program codes. Figure 5.6 shows the Win rate matrix for every model.

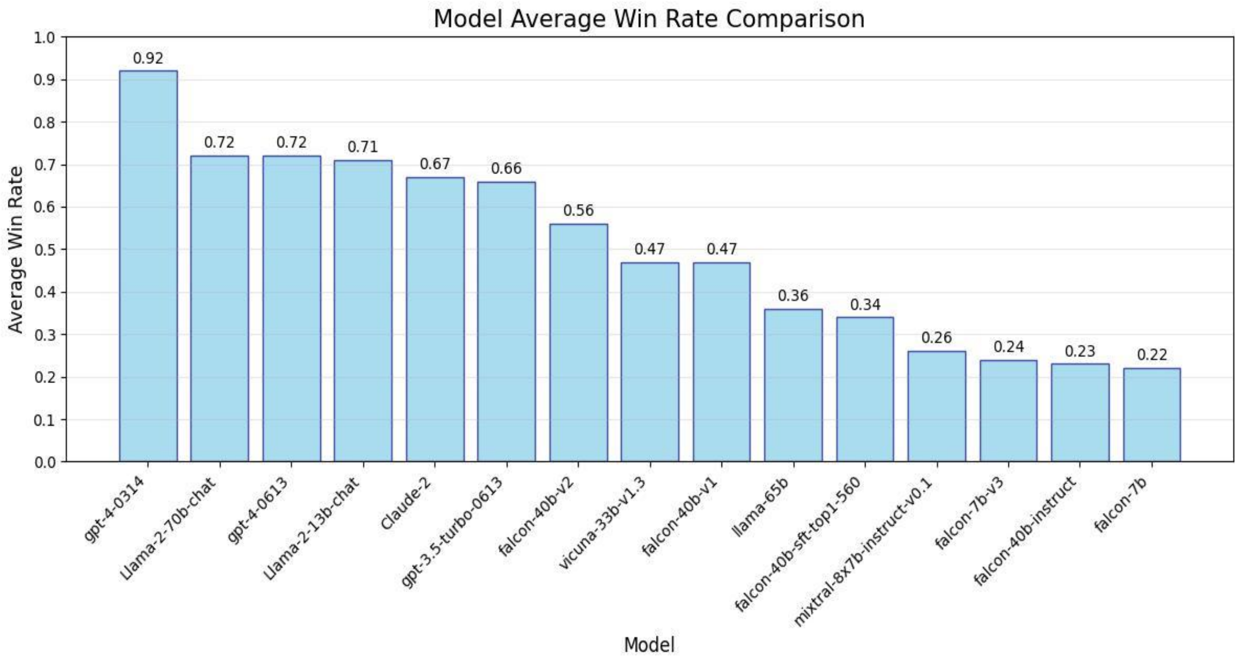

Figure 5.5 Average win rate against all other models

5.3.2 Observations and Insights

- gpt-4-0314 and gpt-4-0613: These two models consistently outperformed the others across multiple evaluation criteria. They demonstrated a strong ability to generate relevant, clear, and comprehensive questions. Their top positions highlight their suitability for question-generation tasks related to the scripts.

- Relevance: “gpt-4-0314” and “gpt-4-0613” excelled in relevance, providing questions that were contextually connected to the script content and clearly articulated.
- Coverage: Some models, such as falcon-40b-v1 and mixtral-8x7b-instructv0.1demonstrated limited coverage, with questions that missed certain key aspects of the scripts.

Figure 5.7 shows the metric score for the models and compares relevance, clarity and coherence, conciseness, and coverage. Clearly, gpt4-0314 shows superiority.

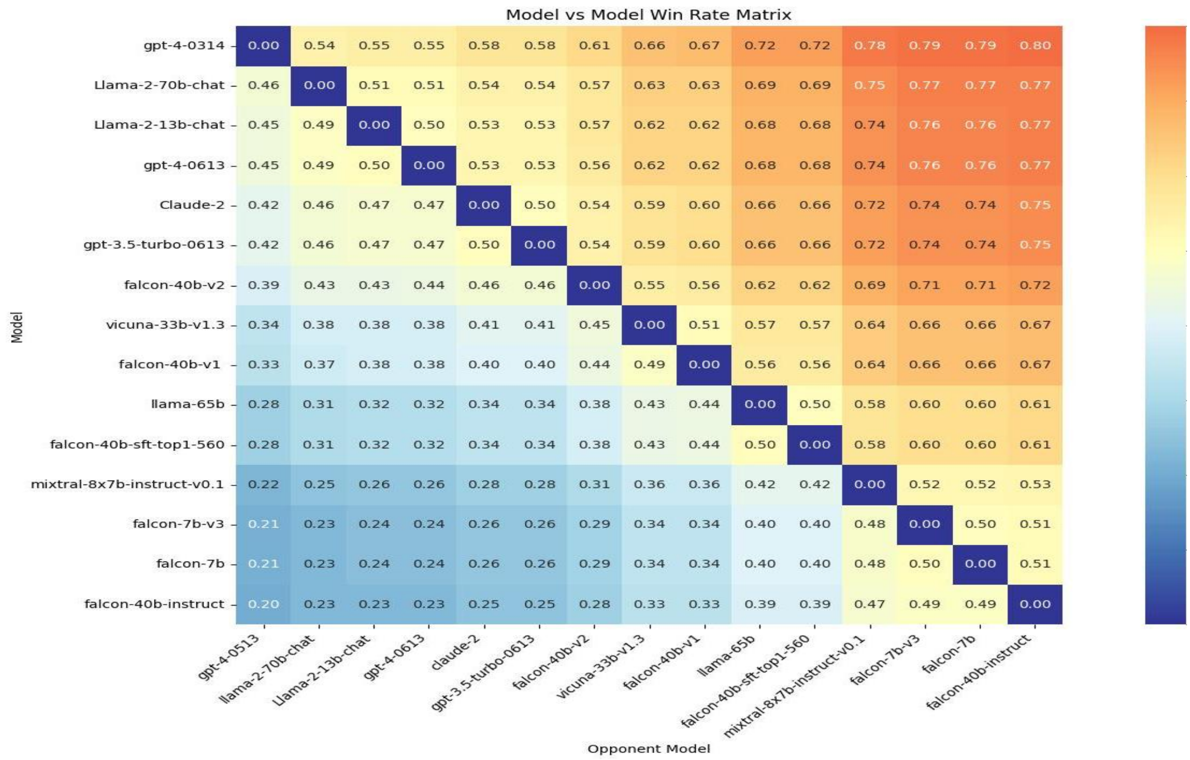

Figure 5.6 Win rate matrix

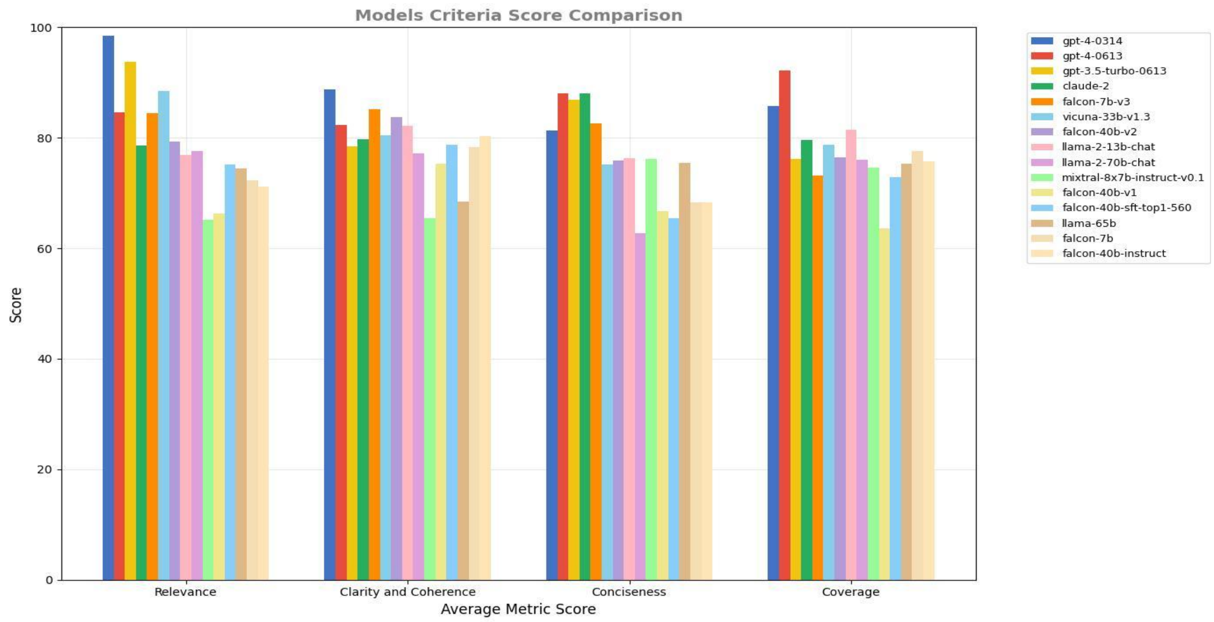

Figure 5.7 Models criteria score comparison

5.3.3 Repetitive Evaluation

Table 5.3 shows the repeated question rate results. The table shows that gpt-4-0314 has the best rate among the other models. Figure 5.8 shows a visual representation of the data. As shown in Figure 6.8, GPT-4 in both versions had the lowest question repetition. At the same time, falcon-7b had the highest number of repeated questions.

Table 5.3 Repetition rates for each model at different question levels

Model	10 questions	20 questions	30 questions	40 questions	50 questions
gpt-4-0314	0	0	0	1	1
llama-2-70b-chat	0	0	1	1	2
gpt-4-0613	0	0	1	1	2
llama-2-13b-chat	0	1	1	2	2
claude-2	0	1	1	2	3
gpt-3.5-turbo-0613	0	1	1	2	3
falcon-40b-v2	1	1	2	2	3
vicuna-33b-v1.3	1	2	3	3	4
falcon-40b-v1	1	2	3	3	4
llama-65b	2	3	3	4	5
falcon-40b-sft-top1-560	2	3	3	4	5
mixtral-8x7b-instruct-v0.1	3	4	4	5	6
falcon-7b-v3	3	4	4	5	6
falcon-40b-instruc	3	4	4	5	6
falcon-7b	3	4	5	6	7

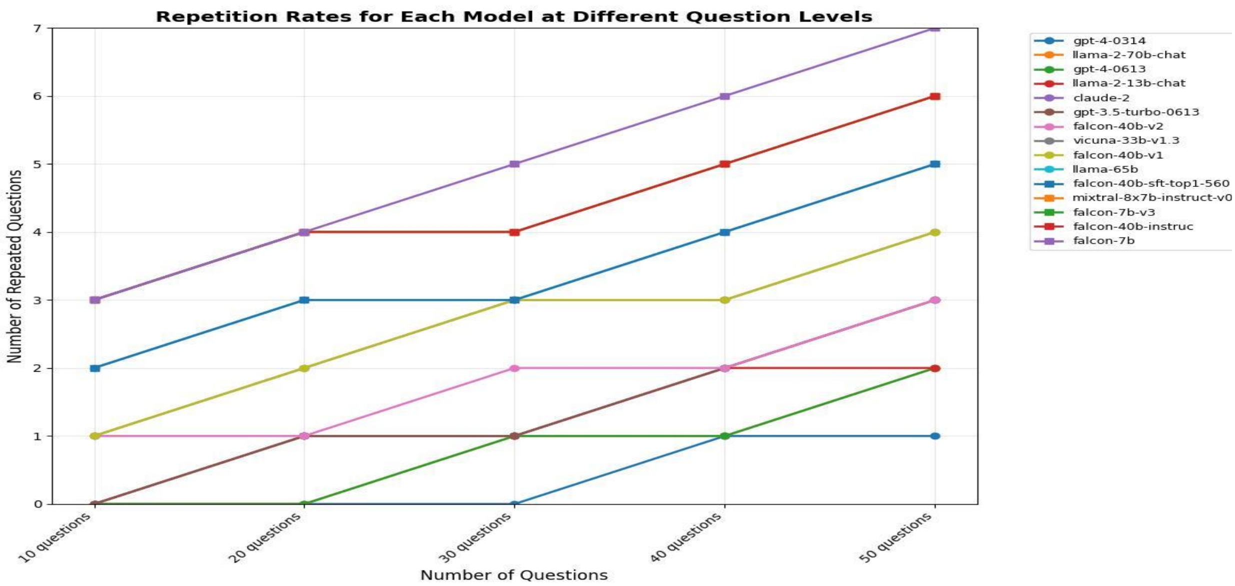

Figure 5.8 Repetition rate

5.3.4 Human Evaluation

While the study incorporates well-defined automated evaluation metrics, relying solely on algorithmic assessment can limit the contextual and pedagogical nuance captured in generated questions. To address this limitation, human evaluation was introduced as a complementary measure. Five educators independently assessed a stratified sample of 45 automatically generated questions; 15 per programming language (C++, Java, and Python). Using a 5-point Likert scale (1 = poor, 5 = excellent), each question was rated along two key dimensions: relevance and educational value. Table 5.4 summarizes the human evaluation scores across the three programming languages and code types.

Table 5.4 Human evaluation summary table

Language	Code Type	Relevance	Educational Value
Python	General	4.8	4.75
Python	Procedural	4.85	4.83
Python	Object-Oriented	4.95	4.87
Java	General	4.85	4.78
Java	Procedural	4.88	4.86
Java	Object-Oriented	4.94	4.92
C++	General	4.65	4.58
C++	Procedural	4.72	4.65
C++	Object-Oriented	4.85	4.8
Average Score	All	4.83	4.78

Table 5.5 presents the results of the repeated-measures ANOVA on relevance and educational value metrics. The analysis revealed no statistically significant differences across programming languages ($F(2,8) = 0.96$, $p = 0.4239$), suggesting that language choice did not affect perceived question relevance meaningfully. A similar pattern was observed for the educational value metric ($p = 0.0689$), which approached but did not reach the conventional threshold for significance ($\alpha = 0.05$). Post-hoc pairwise comparisons, summarized in Table 5.6 and Table 5.7, support this finding. No significant differences emerged between language pairs concerning relevance, as all adjusted p-values exceeded the threshold for statistical significance. About educational value, the comparison between C++ and Python yielded the lowest p-value ($p = 0.0186$); however, after applying the Bonferroni correction, the adjusted p-value rose to 0.0557. This result may be considered marginally significant. Finally, a Pearson correlation analysis revealed a weak positive relationship between relevance and educational value ($r = 0.30$). It suggests that while the two metrics are related, they capture distinct aspects of human-perceived question quality.

Table 5.5 Repeated measures ANOVA results

Metric	F-value	Num DF	Den DF	p-value
Relevance	0.957	2	8	0.424
Educational Value	3.808	2	8	0.069

Table 5.6 Post-hoc pairwise comparisons – relevance (Bonferroni Corrected)

Language 1	Language 2	t-stat	p-value	Bonferroni Adjusted p
C++	Java	0.784	0.477	1.000
C++	Python	-0.459	0.670	1.000
Java	Python	-1.633	0.178	0.533

Table 5.7 Post-hoc pairwise comparisons – educational value (Bonferroni Corrected)

Language 1	Language 2	t-stat	p-value	Bonferroni Adjusted p
C++	Java	-1.907	0.129	0.388
C++	Python	-3.833	0.019	0.056
Java	Python	-0.514	0.634	1.000

5.4 Discussion

This research is particularly unique as it addresses a gap in the literature concerning artificial intelligence-based question generation for programming education. Earlier studies, such as the one conducted by Maity et al. [113], focused on how LLMs can generate different kinds of questions, including open-ended and multiple-choice formats. Although these studies focused on generating questions about multi-language and multi-format general educational purposes, they did not consider programming-related artifacts such as program codes. Similarly, Tran et al. [114] and Doughty et al. [115] addressed the use of LLMs for generating and answering multiple-choice questions (MCQs) in computing education. Still, their focus was mainly on modifying existing questions rather than generating new ones from program codes. Their work indicated how effective models like GPT-3 and GPT-4 are in assessing and generating MCQs related to specific learning objectives. The current research builds on this existing work by utilizing LLMs to generate new questions directly from program code, an area that has not been extensively explored. Unlike previous research that depended on text-based datasets or learners' input, the proposed method assesses how well LLMs can convert program codes into educational questions. This method addresses a significant gap by providing automated, context-specific question generation tools tailored to programming education.

Studies such as those by Baral et al. [116] and Kargupta et al. [117] worked on the assessment capabilities of LLMs. They focused on evaluating student responses rather than generating questions. The current study complements these initiatives by focusing on the initial phase of educational assessments (developing high-quality questions that align with programming curricula). The current research enhances understanding of LLM capabilities using evaluation metrics such as relevance, clarity and coherence, conciseness, and coverage. These metrics offer a more detailed perspective than previous studies, which typically focused on general performance benchmarks. These findings improve the use of artificial intelligence-driven tools in programming education, providing scalable solutions for educators and learners alike. The rankings and observations from this evaluation have significant implications for applications that involve generating questions from program codes. The models “gpt-4-0314” and “gpt-4-0613” are well-suited for tasks where the generation of questions that are both relevant and coherent with the script content is critical. Moreover, this research also highlights the importance of using a combination of metrics to comprehensively evaluate LLMs for question generation. The four metrics and the win rate offer a well-rounded view of a model’s performance in this complex task. The proposed framework can assist teachers and online instructors in assessing and testing student knowledge with a large question base. Furthermore, different tests are performed on various models to assist in selecting the best one. The framework also helps in testing model capability in case other models are released in the future.

The proposed LLM-based framework outperforms some existing approaches in programming education assessment by addressing their core limitations. The ontology-based system [P1], though structured via semantic similarity using BERT embeddings (98.5% accuracy), is constrained to Python and lacks human evaluation, limiting its pedagogical depth. It fails to assess cognitive alignment or instructional appropriateness, which are essential for effective educational questions. The hybrid semantic-AI method [P3], relying solely on human evaluation, introduces scalability challenges and conceptual limitations. Its single-language focus and absence of automatic metrics hinder systematic, repeatable assessment across broader educational contexts. The template-based approach [P5] supports multiple programming languages and incorporates both human and automated evaluation. However, low quality scores (0.57–0.59) indicate limited effectiveness, with constrained adaptability to diverse programming constructs. In contrast, the proposed multi-language LLM-based system (Python, C++, Java) integrates both robust automatic metrics (e.g., relevance: 9.85; clarity: 8.87) and expert human evaluation (relevance: 4.83; educational value: 4.78). This dual-layered assessment ensures both technical correctness and pedagogical soundness, offering comprehensive coverage and educational alignment previously unmet by prior models.

In summary, the evaluation has provided valuable insights into the capabilities of various LLMs in generating questions from program codes. The top-performing models can be valuable assets in applications such as educational platforms, code analysis, and automated documentation generation, where high-quality question generation is essential.

5.5 Conclusion

Large language models (LLMs) were extensively explored to evaluate their ability to generate questions from program code. Python, C++, and Java codes were used as input for this purpose. The study involved a diverse range of LLMs for generating code-based questions and automatically evaluating them. These models' substantial dataset of questions was collected and analyzed systematically. The approach used evaluation metrics, including relevance, clarity and coherence, conciseness, and coverage, to comprehensively assess their question-generation abilities. Human evaluation was also introduced as a complementary measure.

The results of the current research are clear and compelling. The models, gpt-4-0314, gpt-4-0613, and Llama-2-70b-chat, consistently ranked as top-performing LLMs across various evaluation criteria. These models demonstrated their proficiency in generating contextually relevant questions while maintaining clarity, conciseness, and comprehensive coverage of the source code content. Their performance emphasizes their suitability for educational platforms, code analysis, and automated documentation generation applications.

The metrics provided quantitative insights into the syntactic and semantic correctness of the generated questions. Ratings were conducted using automatic artificial intelligence assessments (gpt-4-0314), ensuring the generated questions were grammatically correct, semantically meaningful, and contextually appropriate.

The implications of the findings extend beyond question generation. They hold practical value for fields that rely on effective natural language understanding and generation. As artificial intelligence systems increasingly facilitate human-computer interaction, understanding the strengths and limitations of LLMs is essential.

While gpt-4-0314, gpt-4-0613, and Llama-2-70b-chat led the rankings, other evaluated LLMs also showed value in specific use cases and can be advantageous for tasks emphasizing particular question generation aspects.

This research introduces a modern approach to automatically generate and evaluate questions from program codes, contributing to understanding the evolving role of LLMs in this area. A valuable resource has been created for decision-makers employing LLMs in diverse applications by assessing their performance. The findings demonstrate that, as artificial intelligence technologies advance, models like gpt-4-0314 and gpt-40613, and Llama-2-70b-chat set new benchmarks in natural language generation, driving innovation and potential in various domains.

5.6 Summary

Artificial Intelligence and Large Language Models (LLMs) are growing rapidly. E-learning platforms demand effective question-generation methods, and LLMs have made this process much easier. While recent studies have focused on generating questions from text, no prior research has evaluated LLMs' ability to generate questions from program codes (code-based question generation). This study introduces a framework for assessing LLMs' performance in generating questions from program codes. Tailored software was developed to prompt Python, C++, and Java scripts (as input) to LLMs via APIs and evaluate the generated questions. The study compared diverse models, including GPT-3.5, GPT-4, Llama, and Claude2, using automatic evaluation metrics such as relevance, clarity and coherence, conciseness, and content coverage. Human evaluators assessed the generated questions using relevance and educational value metrics to complement the automatic evaluation. Results indicated that gpt-4-0314 and gpt-4-0613 outperformed other models across metrics, highlighting their effectiveness in question-generation tasks. This article discusses the present research's conduct and outcomes, delivering perspectives regarding the models' strengths and limitations while guiding future research. The findings provide insight for educators, software developers, and the academic community. The methodology can help software developers and researchers implement and evaluate these models effectively.

Thesis 3: A systematic evaluation framework was developed to assess the question generation capabilities of large language models, using both automatic and human-centered evaluation metrics. The findings provide insights into their strengths and limitations in generating programming-related assessment questions for potential educational use. **[P4]**

Chapter 6 Template-Based Question Generation from Code Using Static Code Analysis

6.1 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 [P2]. This challenge intensifies in multi-language 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 AQG tools that parse code structures, extract semantic elements, and populate pedagogical templates [118], [119]. Traditional AQG systems relied heavily on template-based approaches that limited question diversity and contextual relevance [P3]. 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++ [120], [121], [122]. These technological advancements coincide with growing pedagogical demands for personalized learning pathways and competency-based assessment frameworks in computer science education [44]. 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 (single-language) without fully automated evaluation [P2], [P3]. As a result, the main added value of this chapter is dealing with multi-language code-based question generation and automating the evaluation process.

This chapter presents a multi-language 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. Static code analysis is adopted to generate questions from program code. 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 templates for different code elements to create varied questions. The research objectives of this study are:

1. Developing a multi-language code question generator capable of automatically producing assessment questions for Python, C++, Java, and C codes (code-based question generation).
2. Establishing 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.

This chapter is structured as follows: Section 6.2 outlines the methodology and the system architecture. Section 6.3 presents the results of the multi-language question generation and evaluation. Section 6.4 discusses the findings, contributions, and limitations. Section 6.5 concludes the chapter with key insights, and a summary is provided at the end.

6.2 Methodology

This chapter proposes a multi-language code question generator capable of automatically producing assessment questions for Python, C++, Java, and C codes. 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 [123]. While the paper [46] 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 6.1 shows the proposed methodology for a code-based multi-language question generator. The research methodology behind the multi-language code-based question generator involves several

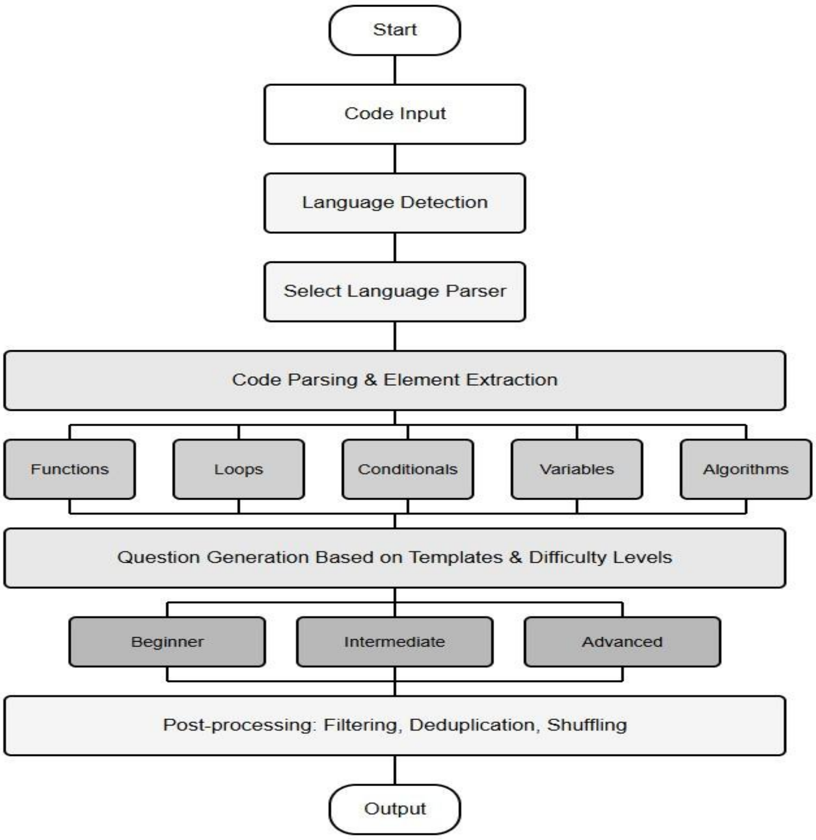

Figure 6.1 Methodology for multi-language code-based question generation

interconnected components that work together to analyze code snippets and generate relevant questions. A detailed explanation of the methodology follows.

6.2.1 Language-Specific Parsing

Parsing is the process of checking the structure of the code and identifying elements like keywords and variables. The foundation of the system is a modular parser that handles multiple programming languages:

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

6.2.2 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 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).

6.2.3 Template-Based Question Generation

The question generation process uses templates customized for different code elements and difficulty levels, as shown in Figure 6.2:

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 template parameters with specific code elements. For example, function parameter examples are generated based on parameter names using heuristic rules.

```
'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}?",],
```

Figure 6.2 Sample of templates used for code-based question generation

6.2.4 Cognitive Science-Based Question Design

The templates are designed based on principles from cognitive science and educational theory, as shown in Figure 6.2:

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.

6.2.5 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.

6.2.6 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. It uses experiments conducted on a real-world dataset to demonstrate its effectiveness in generating questions from program code. 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 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:

$$\text{Linguistic Complexity} = \begin{cases} 0.6 \cdot \text{Normalized Grade Level} \\ + 0.4 \cdot \text{Normalized Sentence Length} \end{cases} \tag{6.1}$$

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

$$\text{Code Coverage} = \begin{cases} 0.4 \cdot \text{Variables Coverage} \\ + 0.6 \cdot \text{Functions Coverage} \end{cases} \tag{6.2}$$

5. Precision: Defined as the ratio of relevant or correct questions to the total number of questions generated by the system.

$$\text{Precision} = \text{True Positives} / (\text{True Positives} + \text{False Positives}) \tag{6.3}$$

6. Recall: Assesses the system’s ability to generate all relevant or expected questions, using code coverage as a proxy indicator for recall.

$$\text{Recall} = \text{True Positives} / (\text{True Positives} + \text{False Negatives}) \tag{6.4}$$

$$\text{F1\_Score} = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall}) \tag{6.5}$$

7. Novelty: Measures the originality of the generated questions using the formula:

$$\text{Novelty} = \begin{cases} 0.4 \cdot \text{Bloom Score} + 0.3 \cdot \text{Code Elements} \\ + 0.3 \cdot \text{Advanced Question Types} \end{cases} \tag{6.6}$$

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

$$\text{Educational Alignment} = \begin{cases} 0.7 \cdot \text{Expected Bloom Match} \\ + \, 0.3 \cdot \text{Expected Linguistic Complexity Match} \end{cases} \tag{6.7}$$

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

$$\text{Cognitive Diversity} = 0.4 \cdot \text{Bloom Score}/6 \; + \; 0.6 \cdot \text{Entropy} \tag{6.8}$$

$$\text{Entropy} \; = \; -\sum \text{p} \cdot \log(\text{p})/\log(6) \tag{6.9}$$

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).

Algorithm 6.1 shows a multi-language template-based question generation and evaluation algorithm. A template-based pipeline aligned with Bloom’s taxonomy and difficulty levels is utilized to generate and evaluate high-quality programming questions from code samples across multiple programming languages. In this pipeline, source code samples undergo parsing using language-specific parsers to enable accurate syntactic and structural analysis. From the parsed code, meaningful elements such as functions, loops, and conditional statements are extracted, and abstract syntax trees (ASTs) are constructed to represent the hierarchical structure of the code. Relevant predefined templates are then selected and instantiated based on the extracted elements, generating candidate questions contextualized to each specific code sample. The generated questions are post-processed to enhance linguistic clarity, eliminate redundancy, and align with pedagogical standards. Each question is labelled with the corresponding Bloom’s level and an estimated difficulty tag to facilitate adaptive learning scenarios. The generated questions are subsequently evaluated using automated metrics to assess quality, novelty, and cognitive diversity, and the labelled questions, along with the evaluation statistics, are aggregated and stored for further analysis and visualization within the system’s reporting modules.

To summarize the overall generation process, the multi-language question generator algorithm is the main engine that orchestrates the entire question generation process. It first detects the programming language of the code snippet, selects the appropriate parser, and parses the code. It then extracts various code elements (functions, loops, conditionals, variables) and identifies the algorithm implemented in the code. Based on the language and extracted elements, it generates appropriate questions. It falls back to generic questions if no specific questions can be generated. Finally, it shuffles the questions and returns the requested number. Next, language detection algorithm uses pattern matching to identify the programming language of the code snippet. It looks for language-specific keywords and syntax patterns to differentiate between Python, Java, C++, and C. Following this, algorithm identification uses regex pattern matching to identify common programming algorithms in the code. Each language parser maintains a dictionary of algorithm names mapped to regex patterns. It returns the name of the first matching algorithm or null if none is detected. Afterward, question generation by element type generates questions for a specific type of code element (functions, loops, conditionals, etc.). It also uses predefined templates for each element type and difficulty level. Finally, mixed-difficulty question generation generates questions at beginner, intermediate, and advanced difficulty levels. It combines questions from different difficulty levels and eliminates duplicate questions to ensure variety.

Algorithm 6.1: Multi-Language Template-Based Question Generation and Evaluation

```
Input: Set of code samples in various programming languages (SourceCodeSamples),
       Predefined question templates mapped to Bloom's taxonomy and difficulty levels (Templates)
Output: Generated questions with Bloom's level and difficulty tags (LabelledQuestions),
        Evaluation statistics for generated questions (EvaluationMetrics)
for each CodeSample in SourceCodeSamples do
    ParsedCode ← Parse(CodeSample, LanguageSpecificParser)
    CodeElements ← ExtractCodeElements(ParsedCode)
    AbstractRep ← GenerateAST(ParsedCode)
    CandidateQuestions ← ∅
    for each Element in CodeElements do
        RelevantTemplates ← SelectTemplates(Element, Templates)
        for each Template in RelevantTemplates do
            Question ← InstantiateTemplate(Template, Element)
            CandidateQuestions ← CandidateQuestions ∪ {Question}
        end for
    end for
    FilteredQuestions ← Postprocess(CandidateQuestions)
    LabelledQuestions ← LabelQuestions(FilteredQuestions)
    EvaluationMetrics ← Evaluate(LabelledQuestions, CodeSample)
    Store(LabelledQuestions, EvaluationMetrics)
end for
GenerateReportsAndVisualizations()
```

6.3 Results

This chapter presents a multi-language code-based question generator capable of automatically producing assessment questions across the top four programming languages (Python, C++, Java, and C) chosen according to the TIOBE Index. 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 [124]. 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 6.2.6 of the 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.

A sample transformation from code to question is presented in Table 6.1

Table 6.1 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?"

Figure 6.3 presents Bloom's Taxonomy coverage. Bloom’s Taxonomy cognitive level distribution was computed using a detailed multi-step process. 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 to 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 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 6.4 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. By default, the distribution of generated questions is set to a 2:2:2 ratio - two beginner, two intermediate, and two advanced. This deliberate balance ensures that one-third of the questions target each difficulty level, providing a well-rounded assessment experience.

Figure 6.5 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.

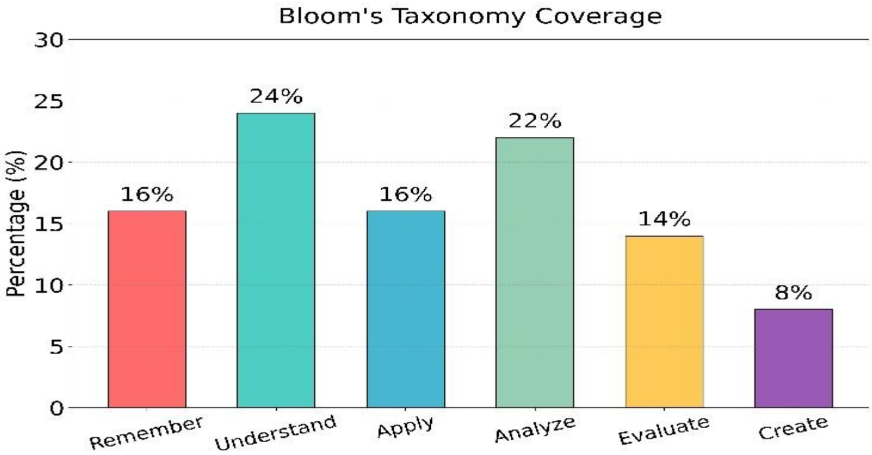

Figure 6.3 Bloom's taxonomy coverage

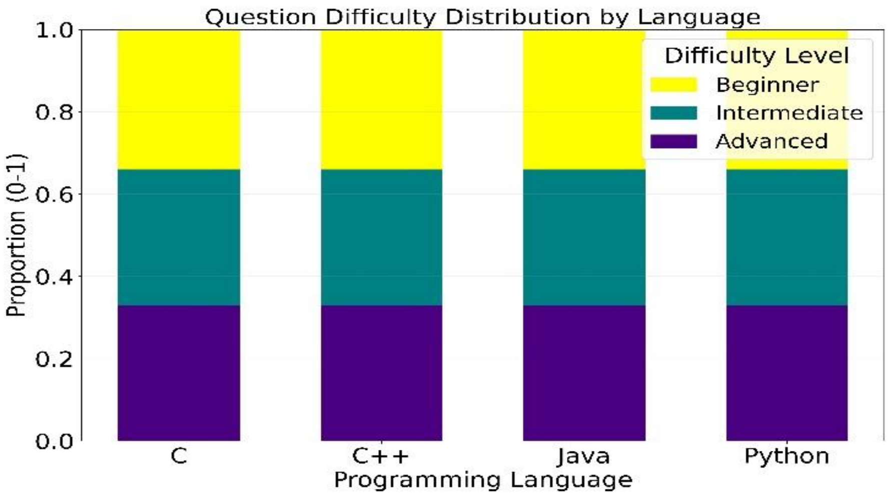

Figure 6.4 Question difficulty distribution by language

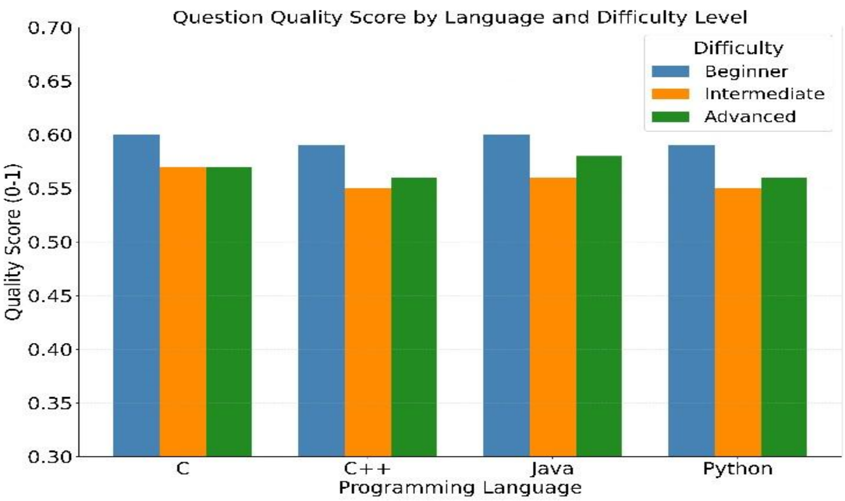

Figure 6.5 Question quality score by language and difficulty level

Figure 6.6 focuses on the question quality score by language and code complexity. The scores shown in this visualization were calculated through a multi-step process. 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.

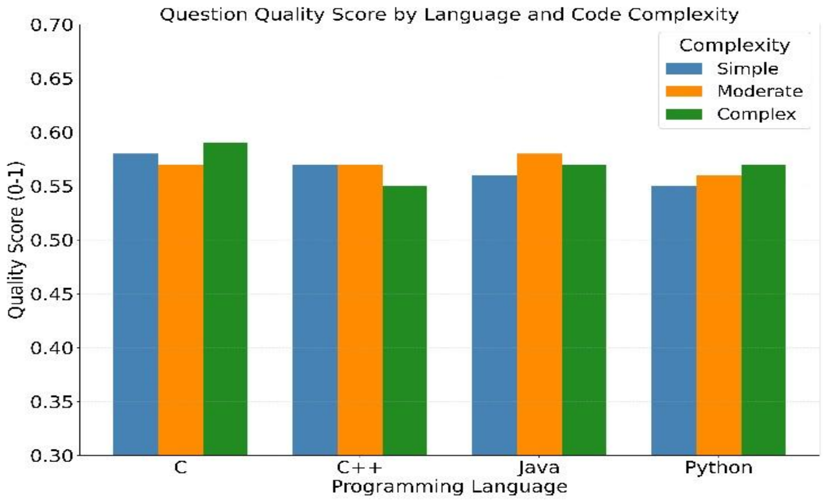

Figure 6.6 Question quality score by language and code complexity

Figure 6.7 visualizes the linguistic complexity of different programming languages (C, C++, Java, and Python) across three difficulty levels: Beginner, Intermediate, and Advanced. In general, linguistic complexity often tends to increase with difficulty level.

The linguistic complexity scores were calculated using a structured, multi-step process. First, basic text metrics, including word and sentence counts, were computed for each question to analyze sentence structure and length. Next, readability metrics - including Flesch-Kincaid Grade Level - were generated using the Textstat library to assess how readable and educationally appropriate the questions were. To further evaluate syntactic complexity, the average sentence length was calculated. All these metrics were then normalized to a 0–1 scale for comparability, with sentence length capped at 25 words and the grade level normalized to a maximum of 10. Using these normalized values, a final linguistic complexity score was derived using a weighted formula: 0.6 times the normalized Flesch-Kincaid Grade plus 0.4 times the normalized sentence length. Finally, the scores were aggregated based on difficulty level - Beginner, Intermediate, and Advanced - to analyze patterns in linguistic complexity across question tiers.

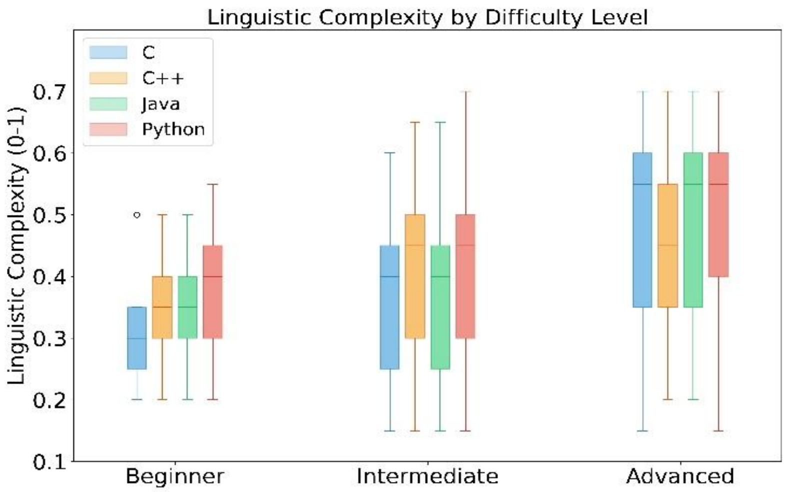

Figure 6.7 Linguistic complexity by difficulty level

Figure 6.8 shows that the average question diversity scores varied by language, ranging from 0.63 for C to 0.55 for C++. The diversity scores were calculated through a structured, multi-step process using Shannon entropy to measure how evenly questions were distributed across different question templates and types. This differs from cognitive diversity, which specifically measures the distribution of Bloom's taxonomy levels. The question diversity metric aggregates scores by programming language by collecting template usage patterns across different algorithms and averaging them across each language's question set. All diversity scores were normalized to a 0–1 scale for cross-language comparison. The results suggest that C code naturally elicits the most diverse range of question types (0.63), followed by Java (0.59) and Python (0.57), while C++ generates the least diverse questions (0.55). This variation may reflect the inherent structural differences between programming languages, with C's lower-level constructs potentially offering more varied questioning opportunities compared to C++'s more standardized object-oriented patterns.

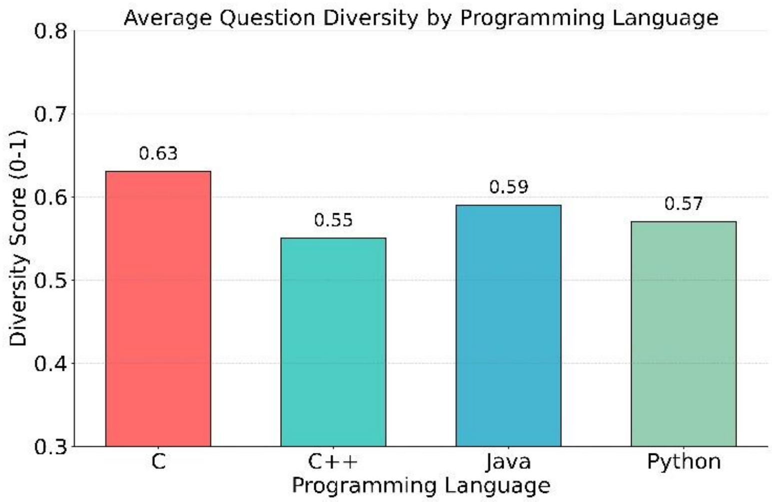

Figure 6.8 Average question diversity by programming language

Table 6.2 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.

Table 6.2 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	$F(3,452) = 5.01, p < 0.01$
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	$F(3,452) = 3.35, p < 0.05$
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	$F(3,452) = 4.61, p < 0.01$

There is no agreed-upon or standard evaluation metric for code-based question generation for educational purposes. 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 6.3 shows human evaluation metrics for code-based question generation across four programming languages. Table 6.3 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 compares C versus each of the average scores of Python, Java, and C++, as shown in Table 6.4. Two, one-way ANOVA comparing average scores across all four languages (F-statistic: 48.44, p-value: 1.01e-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 6.3 Human evaluation results by programming language (N=40)

Metric	Python	Java	C++	C
Relevance	3.8	3.7	3.7	3.8
Educational Value	3.1	3.2	3.2	3.2
Average Score	3.45	3.45	3.45	3.50

Table 6.4 Paired t-test results for human evaluation differences

Comparison	t-statistic	p-value	Significant? (α=0.05)
C vs Python	7.22	0.00005 (very low)	Yes
C vs Java	9.64	0.000005 (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.

6.4 Discussion

6.4.1 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.

6.4.2 Limitations

While this chapter'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 parsing, 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.

6.4.3 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.

6.5 Conclusion

This chapter 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 ($F = 48.44$, $p = 1.01e-12$), confirming its practical applicability. The generated questions spanned all six Bloom's taxonomy levels. The levels are 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 multi-language 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. While performance was consistent, the approach may not capture advanced or creative problem-solving nuances. Current diversity and quality scores highlight room for improvement. Future work should expand template libraries, improve 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 strong 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 assessment systems, bridging static analysis and educational theory to meet the evolving demands of computer science education.

6.6 Summary

This chapter presents a multi-language 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.

Thesis 4: A modular system was developed for automatic question generation using template-based static code analysis, enabling modular question generation designed to be extensible with minimal integration overhead. The framework supports multiple programming languages through customizable parsing templates within a unified architecture. **[P5]**

Chapter 7 Multi-Language Static-Analysis System for Automatic Code-Based Question Generation

7.1 Introduction

Automatic question generation (AQG) has become an important approach as the assessment in programming education has grown into a significant challenge. Computer programming education is considered increasingly important in the age of technology, and coding education is now regarded as a fundamental skill in many fields other than computer science [125]. The growth of programming education is accompanied by the increasing difficulty of educators in defining a diverse and high-quality set of assessment applications that can reasonably assess student knowledge of various programming languages, algorithms, and problem-solving abilities in different cognitive levels [P2]. Automatic question generation from program code has also become a major research topic, with the demand growing for resourceful education tools and automatic assessment models in computer science [126]. Automatic generation of questions has become popular, especially in education, when individualized assessment is required [P2], [P3]. Manual development of questions is time-consuming. Thus, the automatic formulation has been investigated [127]. The creation of questions manually is time-consuming and labor-intensive. It may lead to weak coverage of programming concepts and cognitive skills, which causes large gaps in student assessment and learning outcomes.

Control Flow Graphs (CFG) and Program Dependence Graphs (PDG) are important intermediate representations and are structured views of the complicated control and data dependences in a program [128]. The graphs are useful in building a strong basis that extracts semantically useful information that can be used to develop interesting and challenging questions. More recent developments in deep learning have resulted in the development of code-generation models that can generate source code based on natural language and code-based hints with high accuracy [129]. Automatic programming, as a field, seeks to reduce human interaction in the production of executable code and has singled out code search, code generation, and program repair as the major topics [130]. The main purpose of this chapter is to discuss a synergistic combination of CFG-based and PDG-based analyzers regarding the scenario of generating questions about program codes, including the approaches, results, and possible future aspects.

It has been suggested to use graphs to encode both the syntactic and semantic structure of code and then use graph-based deep learning algorithms to either learn or reason about program structures [33]. Such methods fail to capture dependencies over long distances that are created when the same variable or function is used in widely separated places. Static analysis tools are used to analyze code and provide suggestions for auto-completion, which are usually organized alphabetically [131]. Modern integrated development environments have the code completion feature, contributing greatly to programming efficiency and eliminating code errors [131]. Graph-based program representations, such as CFGs and PDGs, increase the avenues of understanding behavior offered by encoding control flow and data dependency graph representations. This more elaborate representation permits the generation of questions to focus on particular elements of functionality, logic, and possible code weaknesses, thus facilitating a more thorough evaluation of the programmer's knowledge [33].

There is a specific challenge related to the multi-language nature of programming education. During their studies, students study a variety of programming languages, beginning at lower levels, such as Python, and moving on to systems programming languages, such as C and C++, and to object-oriented languages, such as Java. All languages have distinct paradigms, syntaxes, and idiomatic constructs and need specialized parsing and analysis algorithms. These challenges are further added by the difficulty of programming education today. Learners are required to learn through numerous programming languages, learn the different paradigms of thinking algorithmically, and acquire skills at several cognitive levels, including concrete syntax recall, abstract problem-solving, and code-writing. Conventional evaluation methods have a problem covering these dimensions comprehensively and sustaining consistency and quality. This shortcoming is especially acute in large-scale education contexts where hundreds or thousands of students need tailored assessment materials. A general question generator must cover this multi-language aspect across languages with uniform quality and coverage. The chapter deals with the background of multi-language nature in the context of education in programming by proposing a consistent model for code analysis and question generation in four commonly accepted programming languages. It presents a force-balanced generation procedure, which guarantees overall and even coverage in multiple dimensions, a serious shortcoming of other current technologies. This shows that at all levels of cognitive difficulty, advanced graph-based code analysis techniques can effectively generate higher-quality questions, and the whole scope of assessment can be increased. It offers a strategic scheme to assign different difficulty levels to programming languages per the general computer science learning route. It comes up with a list of general evaluation criteria to determine the future of research and development on automatic question generation. Such contributions open up major implications in programming education, especially by easing a potential burden on educators, providing higher quality and broader assessment coverage, and an enhanced learning experience for students in various programming languages and levels of proficiency. The research objectives of this chapter are:

1. To design and implement three automated pipelines (CFG-based, PDG-based, and CFG-PDG Synergetic) for code-based question generation, each leveraging different code analysis strategies to explore their effectiveness in producing high-quality, pedagogically aligned questions.
2. To develop an organizational multi-dimensional evaluation system to measure the system performance in terms of coverage balance, quality of questions, linguistic complexity, and diversity in all dimensions. This framework encompasses automated measures along with human assessment measures.

The remainder of this chapter is organized as follows: Section 7.2 presents the multi-language question generator system methodology, including the system architecture, language-specific parsing techniques, and advanced code analysis methods. Section 7.3 presents the system evaluation results, including coverage balance, question quality, linguistic complexity, diversity metrics, and human evaluation metrics. Section 7.4 discusses the implications of the results, the contributions and limitations of the study, and directions for future research. Section 7.5 concludes the chapter. A brief summary is provided at the end of the chapter.

7.2 Methodology

This chapter presents a code-based multiple-language generator and evaluator system that is capable of generating coding questions in various languages, and in this case, the identified languages are Python, C++, Java, and C. These four language choices were the result of being some of the most popular languages at the moment, as classified by the May 2025 listing of the TIOBE Index and ranking software development languages and their current popularity list [123]. It uses an advanced pipeline structure to transform source code written in several programming languages into good-quality assessment questions distributed across different dimensions in a balanced manner. This section presents a comprehensive description of every element within the pipeline and interconnected characteristics and functions of the general system. Figure 7.1 shows the development of the multi-language code-based question generator and evaluator system. The methodology is a complex of several important elements that interact with each other to interpret code fragments and generate useful, applicative questions. The following sections have a step-by-step analysis of how everything works. This section delivers the complete multi-language code-based question generator and evaluator system methodology, in which the architecture, implementation, and evaluation framework are outlined. The system was developed to tackle severe shortcomings of available automated assessment frameworks on programming education with novel parsing, analysis, generation, and evaluation strategies.

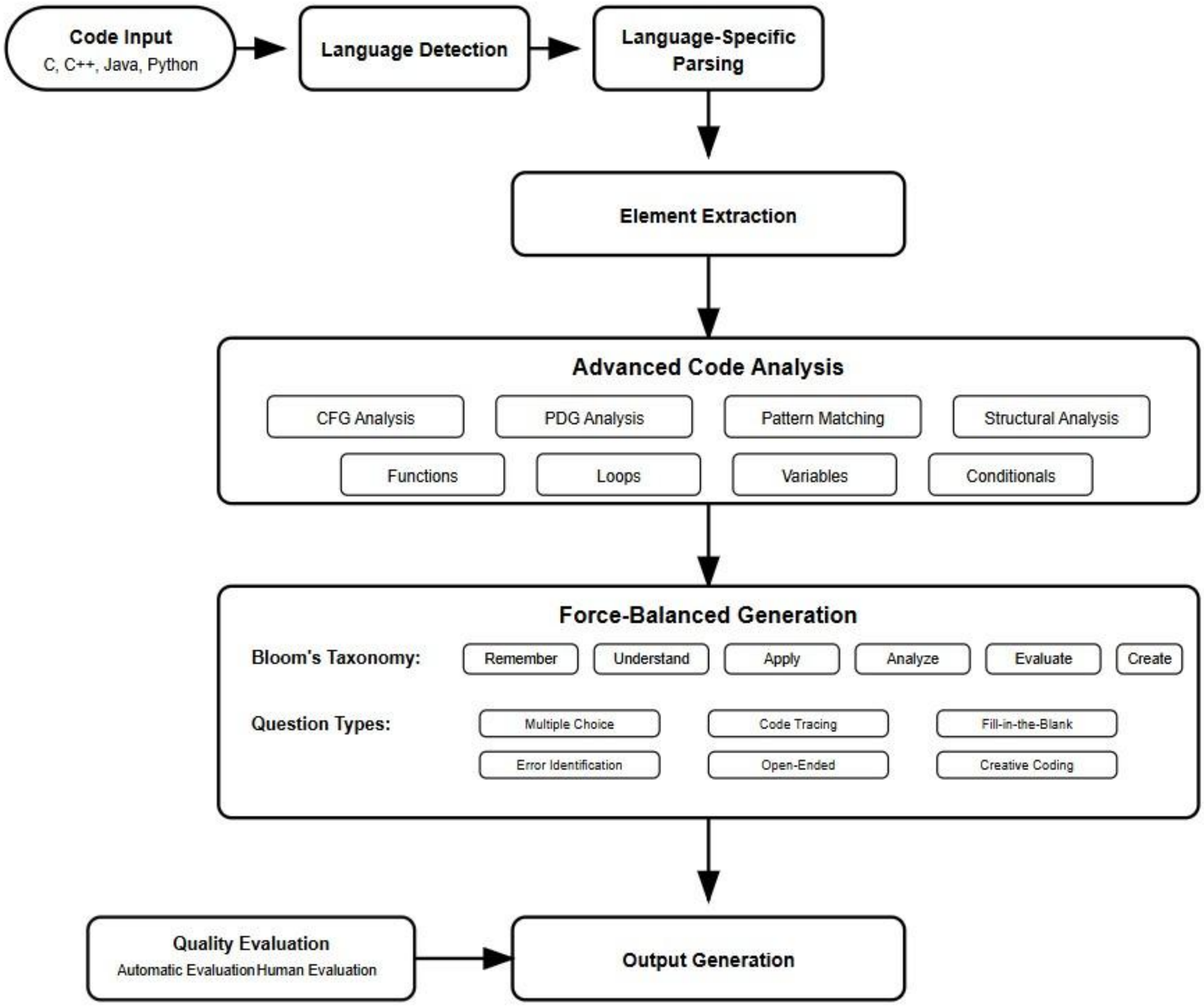

Figure 7.1 Comprehensive pipeline for multi-language code-based question generator and evaluator system

7.2.1 System Architecture and Design Philosophy

The objective of building a multi-language code-based question generator and evaluator system is to support the growing demands to meet the assessment issues in programming education, which traditional manual methods cannot prospectively accommodate the demands of scaling with an expanding enrollment base and range of curriculum needs. Four basic design principles that informed each detail of architecture and implementation governed the system:

1. Language Inclusivity Principle: The system supports Python, Java, C++, and C programming languages, as these are the four most taught programming languages in computer science education, as per the TIOBE Index. This multi-language strategy curbs the limitations of current systems by being multi-language to the level that students could get constant assessment throughout their whole programming program.
2. Algorithmic Diversity Principle: The system includes a collection of 19 fundamental algorithms offered in 6 categories: sorting algorithms (Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort), searching algorithms (Binary Search, Linear Search, Knuth-Morris-Pratt), graph traversal algorithms (Depth-First Search, Breadth-First Search, Topological Sort), shortest path algorithms (Dijkstra algorithm, Floyd Warshall algorithm, A* Search), minimum spanning

tree algorithms (Kruskal, Prim), and optimization techniques (Dynamic Programming, Greedy Algorithms, Huffman Coding). This extensive coverage will allow the students to be assessed on the entire range of algorithmic concepts required in computer science learning.

3. Cognitive Alignment Principle: The system creates questions that cover each of the six levels of Bloom’s Taxonomy: remembering, understanding, applying, analyzing, evaluating, and creating, so that the cognitive information is thoroughly assessed at both ends of the spectrum in recollection and way high up in terms of solving problems and also devising codes. Such consistency with pre-existing structures in education generates questions predisposed toward gradual skill-building hierarchies and critical thinking.

4. Comprehensive Evaluation Principle: The system consists of an automated measure (parsing success rates, placeholder resolution rates, balance scores, and quality metrics), in addition to human assessment by subject matter experts, to ensure that the questions generated are of high quality and pedagogically sound for use in education.

The pipeline shown in Figure 7.1 starts by feeding in source code, possibly choosing four supported programming languages: Python, Java, C++, or C. This is used as a preliminary before further analysis and to clear up any problems with encoding, remove comments, normalize whitespace, and do other simple preprocessing chores. The system accepts codes with diverse levels of complexity, which may range from simple to intricate codes of implementation algorithms. The architecture has seven interconnected parts that run code snippets via a chain of specialized transformations and analyses:

1. Language Detection: The system detects the programming language of the code by passing a language identifier.

2. Language-Specific Parsing: It uses language-specific optimized parsers: Python AST module with ast2json and astunparse extensions to provide full syntax tree capabilities, javalang library to provide structured Java code coverage, Clang to provide support of C code, and a custom Clang and LLVM-based parser to provide C++ coverage.

3. Element Extraction: It automatically recognizes and stores programming elements such as functions, classes, variables, loops, conditionals, data structures, and language-specific constructs into an index. This component applies language-specific extraction rules and consistently covers as many pertinent programming elements as possible across languages.

4. Advanced Code Analysis: It incorporates CFG and PDG construction employing NetworkX-based implementations. CFG identifies loops, execution paths, and branching conditionals; PDG captures variable relationships and data dependencies. These graphical representations allow a more complex analysis of the program behavior and the algorithmic patterns.

5. Force-Balanced Generation: It takes dynamic measures to ensure the selection probabilities are readjusted during the final stages of generating solutions.

6. Quality Evaluation: It integrates automated and human-based evaluation to assess question quality on technical accuracy, semantic relevance, educational value, and linguistic clarity.

7. Output Generation: It generates structured questions with detailed metadata that contains the type of question, the difficulty, the level of Bloom's taxonomy, and the question. Due to the output format, the content can be easily scaffolded into learning management systems and educational platforms.

The Python parsing component can use the built-in AST module in Python and additional libraries to analyze and manipulate code in detail. This style gives good insight into the syntactic structure of Python code and is compatible with the complete Python language specification. Java parsing component supports Java analysis, using the javalang library to examine Java sources and incorporating the latest Java features like generics, annotations, lambda expressions, and modular programming constructs. The C parser was first implemented using the pycparser library, which deals with the C programming language. But it was skipping much of the code. As a result, Clang was adopted for C parsing. The C++ parsing unit uses the Clang/LLVM system to execute the analysis of all modern C++ code. The system uses a common parser interface, which offers uniform access to language-specific language-niche parsing features without sacrificing individual parser features and capabilities. This is facilitated by the unified parser interface, which allows the seamless addition of language-specific parsing capabilities with the flexibility of using the individual advantages of different parsers. This architecture helps in an eventual expansion to other programming languages and parsing methods while still being compatible with the current parts.

7.2.2 Advanced Code Analysis Techniques

Control Flow Graph analysis helps one understand the program flow and control structures needed to formulate complex instructions for a program. It enables the full generation and analysis of CFGs with NetworkX-based representations of programs that provide the complete control flow behavior of programs over all supported languages.

Program Dependence Graphs analyze the program dependency and relationships between variables and the information about the control flow given by a CFG analysis. The ability in PDG generation and analysis of the programs in the form of NetworkX-based graph representations facilitates the generation of questions regarding data flow, variable scope, and program semantics. The component of PDG analysis creates detailed representations of all dependencies within programs that reveal the critical data flow and control relationships. The resulting PDGs supplement CFG analysis to give a fully rounded view of both program form and behavior, allowing complex question generation aimed at both semantics and data flow knowledge of programs.

The system has complex graph-matching pattern recognition technology that uses CFG and PDG analysis to detect patterns in algorithms and programming structures. The strategy allows the correct classification of algorithms and helps create algorithm-specific questions aimed at developing a comprehensive knowledge of algorithmic principles. Pattern recognition systems can be based on graphs, and the system obtains high accuracy in the algorithm classification process as it integrates several sources of information, such as structural patterns determined using graph analysis, textual patterns determined using code analysis, and even semantic patterns obtained by program behavior analysis. Such an inclusive model can be used to identify trustworthy algorithms that cut across language programming and coding styles.

Algorithm 7.1 shows the CFG pipeline algorithm for code question generation and evaluation. Its main objective is to generate questions by extracting control flow information from code. It parses code to extract CFG nodes (basic blocks) and edges (control transitions). Then, it analyzes control paths, loops, and branching structures. Finally, it generates tracing, MCQ, and basic error-identification questions based on flow paths.

Algorithm 7.2 shows the PDG pipeline algorithm for code question generation and evaluation. Its main objective is to generate questions using data and control dependencies in the program. It parses code and extracts PDG, capturing data dependencies, variable usage, and control dependencies. Then, it analyzes data flows, variable lifetimes, and semantic relationships. Finally, it generates dependency, comprehension, and advanced error-identification questions.

Algorithm 7.3 shows the CFG-PDG pipeline algorithm for code question generation and evaluation. Its main objective is to generate advanced, diverse questions using a synergistic integration of CFG and PDG. It parses and simultaneously extracts CFG and PDG representations. Next, it integrates structural (CFG) and semantic (PDG) information. Then, it identifies algorithm types with enriched features. Finally, it generates a balanced set of questions, including creative coding and higher-order Bloom questions.

Algorithm 7.1: CFG Pipeline for Code QG and Evaluation

Input: Source Code (SC)

Output: Question Set (QS)

1: Parse SC using language-specific parser.

2: Construct CFG from SC.

3: Identify algorithm type using CFG patterns.

4: Compute cyclomatic complexity for difficulty estimation.

5: Select Bloom-level-aligned templates for CFG-based QG.

6: Fill placeholders using CFG nodes and control paths.

7: Generate QS with tracing, MCQ, and error-identification questions.

8: Evaluate QS using quality and diversity metrics.

Algorithm 7.2: PDG Pipeline for Code QG and Evaluation

Input: Source Code (SC)

Output: Question Set (QS)

1: Parse SC using language-specific parser.

2: Construct PDG from SC.

3: Identify algorithm type using PDG and textual features.

4: Analyze data dependencies for semantic complexity estimation.

5: Select Bloom-level-aligned templates for PDG-based QG.

6: Fill placeholders using PDG nodes and dependency structures.

7: Generate QS with dependency, error identification, and comprehension questions.

8: Evaluate QS using quality and diversity metrics.

Algorithm 7.3: CFG&PDG Synergetic Pipeline for Code QG and Evaluation

Input: Source Code (SC)

Output: Question Set (QS)

1: Parse SC using language-specific parser.

2: Construct CFG and PDG from SC.

3: Integrate CFG and PDG for a unified structural-semantic representation.

4: Identify algorithm type using integrated features.

5: Compute complexity and dependency scores for difficulty estimation.

6: Select templates aligned with Bloom's taxonomy and algorithm type.

7: Fill placeholders using CFG paths and PDG dependencies.

8: Generate QS covering tracing, dependency, error identification, creative coding, and MCQs.

9: Evaluate QS using comprehensive quality, novelty, and diversity metrics.

7.2.3 Evaluation Metrics

The same automatic evaluation metrics as the baseline model (6.2.6 Evaluation Approach) are utilized in the system, such as overall quality score, linguistic complexity, precision, recall, f1-score, novelty score, educational alignment, and cognitive diversity [P5]. Overall Quality Score: Aggregates linguistic quality, technical correctness, and clarity. Linguistic Complexity: Measures readability and sophistication. Precision and Recall: Evaluate generation accuracy and coverage. F1-Score: Balances precision and recall. Novelty Score: Measures uniqueness across questions. Educational Alignment: Alignment with programming learning objectives. Cognitive Diversity: Distribution across Bloom's taxonomy levels.

Relevance and educational value measures were adopted from the baseline system [P5] for human evaluation metrics. Five human-evaluated dimensions are conceptualized to measure the pedagogical soundness, clarity, and cognitive relevance of generated programming questions to measure their quality beyond automatic metrics:

1. Relevance: This metric addresses how well a question aligns with the programming education goal and profession. It encompasses curriculum fit (e.g., ACM/IEEE standards), relevance to real-world scenarios, alignment with learning objectives, significance, and suitability with the target programming language.
2. Difficulty Appropriateness: quantifies the extent to which an author designed a question to unequivocally appear at the cognitive level (Beginner versus Intermediate versus Advanced) to which it is targeted. It considers the prerequisite knowledge needed, the cognitive load, the complexity of the problem, the duration required to solve the problem, and whether the question is scaffolded appropriately for the learners.
3. Clarity: The aspects of how clearly a question is and whether or not it is ambiguous. It encompasses the quality of the grammar, instructional accuracy, suitability of terminology, visual presentation (e.g., readability of the code), and the removal of possible ambiguities.
4. Educational Value: This value reflects the question's ability to foster learning and skill acquisition. Evaluation is based on the depth of understanding of the underlying concept, capability to develop programming skills, portability to other situations, interest and value of engagement, and contribution to learning.
5. Cognitive Level Match: Analysis of the question focuses on the level of Bloom's taxonomy. It evaluates to what extent relevant those cognitive operations included (e.g., remembering, applying, analyzing), the promotion of higher-order thinking, and whether the question was a valid instrument of cognitive assessment.

7.3 Results

The experimental evaluation demonstrated the effectiveness of the proposed approach in generating relevant and challenging questions from program codes. The system successfully generated comprehensive programming questions datasets spread across Bloom levels. Table 7.1 demonstrates how CFG-based, PDG-based, and CFG-PDG approaches distribute across Bloom's Taxonomy, illustrating their alignment with cognitive engagement in algorithm learning. The PDG-based method supports lower to mid-level cognitive processes, particularly remembering, understanding, and analyzing, through its visual and structural program representations. In contrast, CFG-based and CFG-PDG approaches maintain consistent engagement at higher-order levels, specifically in evaluating and creating tasks related to algorithm design and optimization. This distribution highlights how each approach differentially contributes to fostering cognitive development, providing a nuanced basis for aligning teaching strategies with targeted learning outcomes in programming education. The dataset of code snippets used is available on GitHub [124], the same dataset used for the baseline system [P5]. Established educational assessment metrics, outlined in section "7.2.3 Evaluation Metrics" of the methodology, were used to evaluate the generated questions.

Table 7.1 Bloom's taxonomy distribution

Cognitive Level	CFG-Based	PDG-Based	CFG-PDG	Primary Focus Areas
Remembering	76	370	57	Algorithm facts, terminology, syntax
Understanding	76	357	38	Code behavior, step-by-step execution
Applying	76	95	57	Algorithm adaptation, implementation
Analyzing	76	370	57	Efficiency analysis, code structure
Evaluating	76	40	-	Algorithm selection, trade-off analysis
Creating	76	-	38	Algorithm design, optimization

Table 7.2 outlines how various question types are distributed across CFG-based, PDG-based, and CFG-PDG, illustrating their alignment with cognitive skill development in algorithm learning. Multiple-choice, code tracing, and fill-in-the-blank formats are prevalent across all approaches. PDG-based shows higher frequencies, underscoring their effectiveness in reinforcing fundamental concepts and procedural fluency. Error identification tasks appear exclusively within CFG-based activities, aligning with its strengths in syntax analysis and debugging practices. Open-ended questions, promoting reflective reasoning and synthesis, are most prominent in CFG-based tasks but are also utilized within PDG-based and CFG-PDG contexts, supporting deeper cognitive engagement. Creative coding tasks in PDG-based and CFG-PDG approaches highlight these methods' emphasis on practical application and design-oriented learning. This distribution demonstrates a strategic alignment of question types with each pedagogical strength of the approach, ensuring targeted cognitive development within programming education.

Table 7.2 Dataset question type distribution

Cognitive Level	CFG-Based	PDG-Based	CFG-PDG
Multiple Choice	76	357	57
Code Tracing	76	370	57
Fill-in-the-Blank	76	370	57
Error Identification	76	-	-
Open-Ended	152	40	38
Creative Coding	-	95	38

Table 7.3 presents the comparative evaluation of the CFG-based, PDG-based, and CFG-PDG synergistic pipelines, demonstrating clear advancements in automatic question generation for programming education. The CFG-PDG synergistic pipeline consistently achieved the highest overall quality and linguistic complexity scores (0.83), outperforming both the CFG-based (0.78, 0.77) and PDG-based (0.72, 0.62) pipelines. This indicates that the integration of structural (CFG) and semantic (PDG) analyses contributes to the generation of questions that are not only technically sound but also pedagogically rich and linguistically diverse. Precision was similarly highest in the CFG-PDG pipeline (0.83), underscoring its effectiveness in producing relevant, accurate questions. Recall remained modest across all systems, indicating a shared opportunity for future expansion in question variety. The CFG-PDG pipeline maintained a balanced F1-score (0.15), competitive with CFG-based

(0.19) and superior to PDG-based (0.11), demonstrating its capacity to balance quality with breadth despite the inherent challenges in automatic assessment generation. The novelty scores were notably high for both the CFG-PDG (0.96) and PDG-based (0.95) pipelines, illustrating the semantic depth added by PDG analysis, which enhances the diversity of questions beyond surface-level syntax. All systems achieved perfect educational alignment (1.00), reflecting their capacity to generate questions aligned with Bloom's taxonomy and curriculum goals. Importantly, the CFG-PDG pipeline achieved the highest cognitive diversity (0.31), supporting a broader range of question types that facilitate deeper learning and higher-order cognitive engagement. Collectively, these results affirm the CFG-PDG synergistic pipeline as the most robust and effective approach for scalable, high-quality, and cognitively diverse question generation from source code. It successfully bridges the structural strengths of CFG analysis and the semantic insights of PDG analysis, meeting the evolving needs of programming education. Future research should focus on enhancing recall and extending template libraries for rare constructs.

Table 7.3 Automatic evaluation results by approach

Performance Metric	**CFG-Based**	**PDG-Based**	**CFG-PDG**
Overall Quality Score	0.78	0.72	0.83
Linguistic Complexity	0.77	0.62	0.83
Precision	0.77	0.62	0.83
Recall	0.11	0.06	0.08
F1-Score	0.19	0.11	0.15
Novelty Score	0.86	0.95	0.96
Educational Alignment	1.00	1.00	1.00
Cognitive Diversity	0.20	0.29	0.31

Table 7.4 underscores the superiority of the CFG-PDG synergistic pipeline in generating high-quality programming assessment questions across C, C++, Java, and Python. This integrated approach consistently achieved the highest quality scores (0.81–0.85), demonstrating its adaptability across procedural, object-oriented, and scripting languages. The CFG-based pipeline also performed reliably (0.77–0.78), highlighting the value of structural (control-flow) analysis for generating clear and pedagogically sound questions. In contrast, the PDG-based pipeline scored lower (0.71–0.72), reflecting its strength in semantic insight while revealing limitations when used without structural context. These results confirm that combining CFG and PDG analysis is essential for producing scalable, high-quality, language-agnostic question generation, addressing a critical challenge in automated programming education assessment. The CFG-PDG synergistic pipeline thus emerges as a robust solution for educators seeking consistent, meaningful, and pedagogically aligned assessments across diverse programming curricula.

Table 7.4 Quality score by approach per programming language

Programming Language	**CFG-Based**	**PDG-Based**	**CFG-PDG**
C	0.77	0.72	0.84
C++	0.78	0.71	0.85
Java	0.77	0.71	0.82
Python	0.78	0.72	0.81

While the study employs well-defined metrics, the absence of human evaluation limits the contextual accuracy of generated questions. As a result, seven human evaluators were used to complement the automatic evaluation. Five educators independently evaluated a sample of 48 automatically generated questions (12 per programming language). Each question was assessed using a 5-point Likert scale, where 1 represented poor performance and 5 represented excellent performance. The evaluation covered five dimensions: relevance, difficulty, appropriateness, clarity, educational value, and cognitive level alignment. Table 7.5 shows human evaluation metrics for code-based question generation using CFG-PDG across four programming languages. Table 7.5 shows C++ leads slightly. Two tests were conducted to understand whether this slight difference has statistical significance. First is a paired t-test comparing the average of the C++ versus each of the Python, Java, and C scores, as shown in Table 7.6. Two is a one-way ANOVA comparing average scores across all four languages (F-statistic: 1.20, p-value: 0.3098). 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 slight. Table 7.6 shows that all three comparisons show that C++ received significantly higher evaluation scores than C, Java, and Python, confirming that C++ questions were rated most favorably by human evaluators across all metrics.

Table 7.5 Human evaluation results by programming language (N=48)

Metric	C	C++	Java	Python
Relevance	4.31	4.39	4.15	4.07
Difficulty Appropriateness	4.31	4.40	4.17	4.09
Clarity	4.29	4.42	4.17	4.02
Educational Value	4.33	4.41	4.21	4.05
Cognitive Level Alignment	4.27	4.42	4.16	4.01
Average Score	4.30	4.41	4.17	4.05

Table 7.6 Paired t-test results for human evaluation differences

Comparison	t-statistic	p-value	Significant? (α=0.05)
C++ vs. C	2.847	0.031	Yes
C++ vs. Java	6.172	0.001	Yes
C++ vs. Python	8.924	<0.001	Yes

The human evaluation is a valuable counterpart to automated assessment, reinforcing core findings while offering critical insights from an educational perspective regarding question quality. Both methods consistently identified C++ as the stronger performer; however, human reviewers observed a noticeable performance difference across different languages than automated metrics initially indicated. The fact that there should be no difference between automated educational scoring and the evaluations of a human being highlights the validity of using computers in educational settings. However, human involvement in consideration of practical classroom application brings in a critical context that purely algorithmic approaches do not have, reinforcing the need for a multidimensional measurement framework in educational technology research.

7.4 Discussion

The section critically analyzes and breaks down the findings of the experiments and presents their overall implications on programming education, automated assessment, and educational technology.

The discussion delves into the implications of the findings, limitations and challenges, and the broader impact of multi-language code-based question generation on computer science education.

7.4.1 The Proposed Systems and the Baseline Comparison

Figure 7.2 shows a clear performance metric improvement across the four programming languages in the new systems compared to the baseline template-based AQG system introduced in Chapter 6. The comparison between the new systems and the baseline shown in Figure 7.3 reveals substantial improvements across nearly all performance metrics, indicating that the new systems are significantly more effective in generating high-quality programming questions.

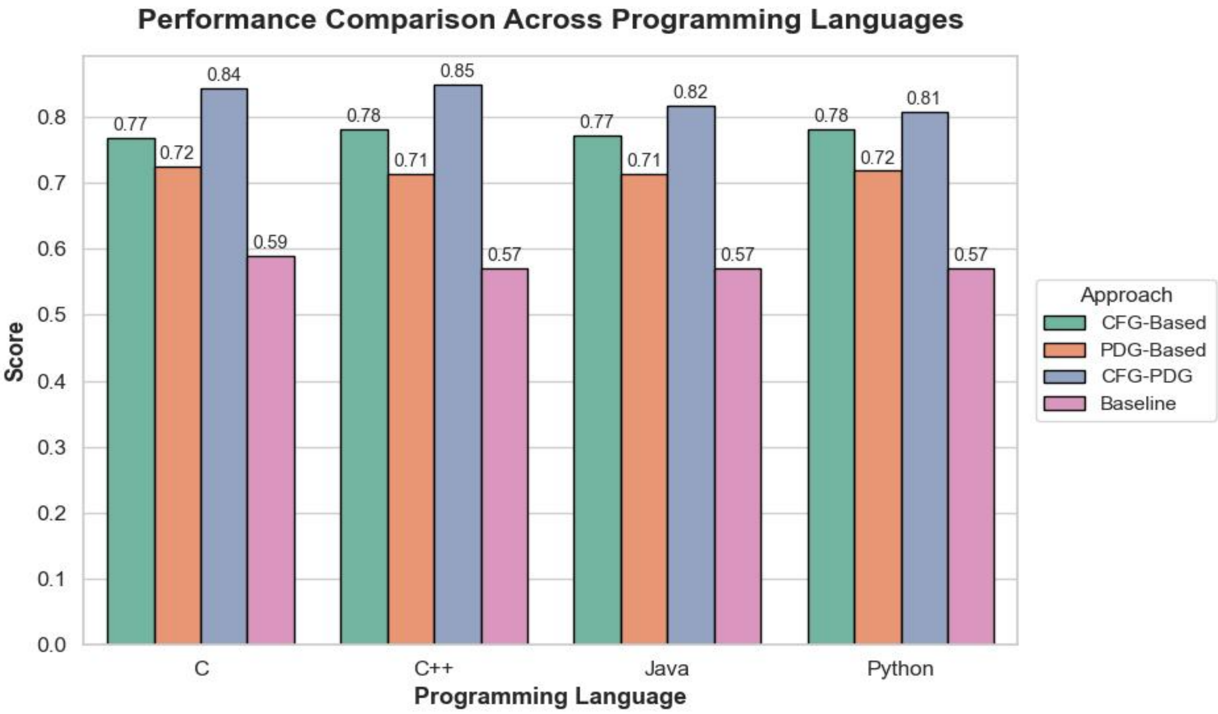

Figure 7.2 Quality score per language for the three approaches compared with the baseline

Figure 7.3 compares the CFG-based, PDG-based, CFG-PDG synergistic, and the baseline template-based AQG system across the evaluation metrics. CFG-PDG synergistic pipeline consistently demonstrates superior performance, achieving the highest overall quality score (0.83) and linguistic complexity (0.83). This suggests that integrating control-flow and semantic dependency analyses enables the generation of questions that are technically accurate and articulated in linguistically rich and varied forms, essential for maintaining learner engagement and supporting nuanced comprehension. The CFG-based pipeline follows closely (0.78, 0.77), indicating that control-flow analysis provides a reliable structure for generating clear and pedagogically aligned questions.

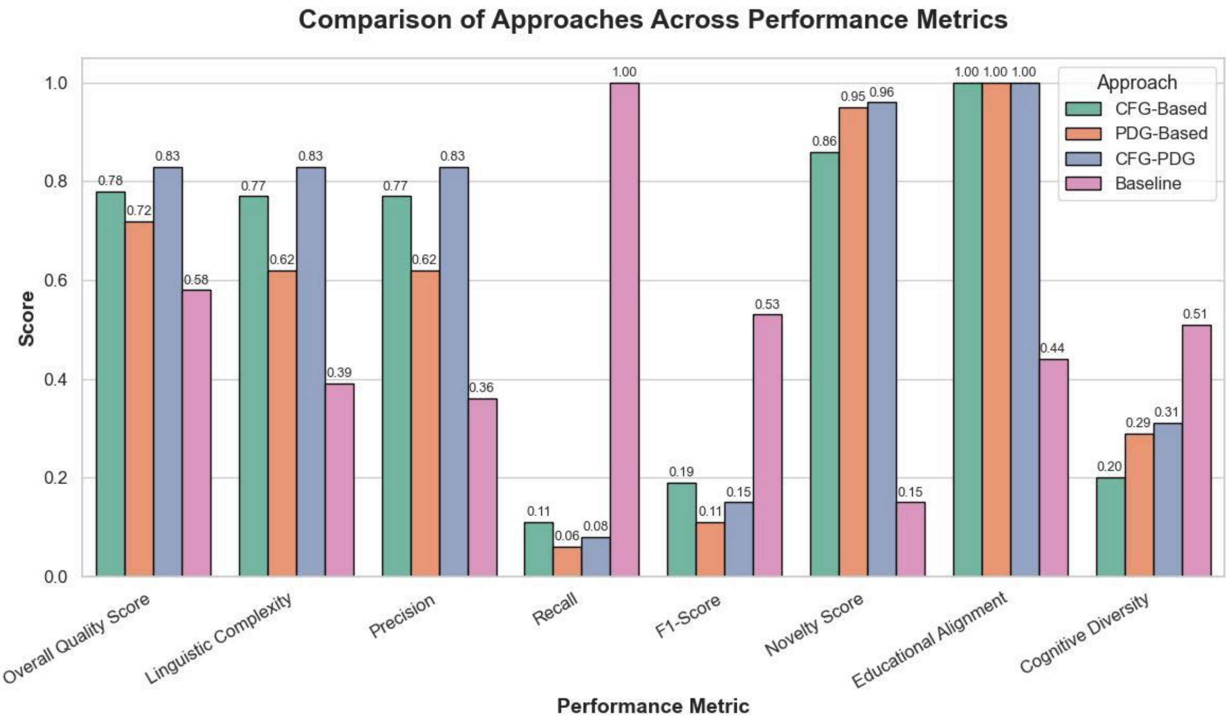

Figure 7.3 Comparison between the proposed approaches and the baseline

However, it lacks the semantic depth required for advanced comprehension and higher-order question types. The PDG-based pipeline, while lower in quality (0.72) and linguistic complexity (0.62), contributes semantic insights that enhance novelty and cognitive diversity, albeit with challenges in clarity and consistency when used independently. In contrast, the baseline template-based AQG system underperforms (0.58 quality, 0.39 linguistic complexity), revealing the limitations of shallow syntax-based approaches that cannot capture deeper structures or semantics of code, often resulting in repetitive and low-cognitive-load questions. The CFG-PDG pipeline demonstrates high precision (0.83), aligning with the CFG-based (0.77) and outperforming the PDG-based (0.62) and baseline (0.36) systems. This indicates the system's capacity to generate relevant, targeted questions with minimal irrelevant outputs, ensuring assessment quality. However, recall remains a shared challenge across all graph-based systems, with scores of 0.08 (CFG-PDG), 0.11 (CFG-based), and 0.06 (PDG-based), compared to the baseline system's high recall (1.00). However, the baseline's perfect recall is misleading; it achieves high coverage by generating a large volume of low-quality, repetitive questions, reflected in its low quality and linguistic complexity scores. The CFG-PDG pipeline, while generating fewer questions, prioritizes relevance and cognitive alignment, as demonstrated by its higher precision, ensuring that the generated assessments are meaningful rather than voluminous. The F1-score further confirms this trade-off, with the CFG-PDG pipeline scoring 0.15, lower than the baseline's 0.53, but aligning with the commitment to precision and quality over sheer quantity. This suggests that a high F1-score driven by excessive recall without quality control in educational assessment may not equate to pedagogical effectiveness. Notably, the CFG-PDG pipeline achieves the highest novelty score (0.96), marginally surpassing the PDG-Based (0.95) and outperforming the CFG-Based (0.86) and Baseline (0.15). This indicates that incorporating semantic dependency analysis allows the system to generate diverse, non-trivial questions that push learners beyond rote memorization, enhancing engagement and learning outcomes. Educational alignment remains perfect (1.00) across all graph-based systems, underscoring their consistent alignment with learning objectives and Bloom's Taxonomy levels. In contrast, the baseline system's lower alignment score (0.44) highlights its inadequacy in maintaining pedagogical coherence. Cognitive diversity is highest in the CFG-PDG pipeline (0.31), followed by the PDG-Based (0.29) and CFG-Based (0.20), indicating the CFG-PDG pipeline's superior ability to generate questions spanning various cognitive levels, including analysis, evaluation, and creative coding. Despite a numeric cognitive diversity score of 0.51, the baseline system often produces superficially diverse but low-order questions, lacking depth and true cognitive challenge.

7.4.2 Research Contributions and Educational Implications

The generator is a key event in automatic assessment. Its capability to produce balanced content in terms of languages, difficulty levels, Bloom's taxonomy, and the form and types of questions helps address the bias inherent to manual question generation. The proposed study contributes to educational technology by showing that rich computational modeling strategies can reliably operationalize abstract educational concepts like cognitive complexity, difficulty progression, and content balance. The empirical implication of the possibility of automating cognitive assessment in programming instruction is that Bloom's taxonomy classification of expert evaluation was systematic in that the feasibility of cognitive assessment in programming education was proven. The four programming languages are empirically supported with consistent performance based on theories that focus on conceptual rather than memorization of languages. The fact that it included all 19 fundamental algorithms and divided them into six categories covers areas of common curriculum deficiencies, with some algorithms being emphasized more than others. The pedagogical system ensures that the students will get an in-depth exposure to algorithmic concepts needed to learn computer science.

7.4.3 Research Limitations

The focus on 19 algorithms excludes advanced topics (e.g., machine learning, cryptography). Limited language support (Python, Java, C++, C) misses functional and web languages. The system emphasizes algorithmic tasks over higher-order software engineering skills. Standardized formats may not fully capture real-world complexity or creativity. Static analysis limits insight into run-time behavior.

7.4.4 Future Research Directions

Future development should prioritize expansion to additional programming languages, particularly those representing different paradigms such as functional programming, concurrent programming, and domain-specific languages. The modular architecture provides a foundation for such expansion, though each new language will require careful consideration of paradigm-specific concepts and assessment approaches. Integration with adaptive learning platforms could provide personalized educational experiences based on individual student progress and learning patterns. Longitudinal studies of student learning outcomes would provide crucial evidence for the educational effectiveness of automated question generation. Such studies should examine immediate learning gains, retention, transfer to new contexts, and development of expert-like problem-solving skills.

7.5 Conclusion

This study presents a robust, scalable, and pedagogically aligned system for automatic question generation (AQG) from source code, leveraging Control Flow Graph (CFG), Program Dependence Graph (PDG), and a synergistic CFG-PDG pipeline to address the longstanding challenge of generating high-quality programming assessments across Python, Java, C++, and C. The developed system systematically covers 19 fundamental algorithms, six levels of Bloom's taxonomy, and a diverse range of question types with balanced distribution across beginner, intermediate, and advanced difficulties. Empirical results demonstrated that the CFG-PDG synergistic pipeline consistently outperformed standalone CFG-based and PDG-based pipelines across key metrics, achieving an overall quality score of 0.83, linguistic complexity of 0.83, and precision of 0.83. Notably, it maintained the highest novelty (0.96) and cognitive diversity (0.31), underscoring its ability to generate diverse, semantically rich, and cognitively engaging questions essential for programming education. The system's perfect educational alignment (1.00) across all pipelines confirms its compatibility with curriculum goals, facilitating integration into adaptive learning platforms and scalable online courses. Despite these advancements, limitations remain, particularly in expanding coverage to functional and web languages and in capturing dynamic program behaviors. Future work will prioritize template library expansion, dynamic analysis integration, and longitudinal studies to assess the system's impact on learning outcomes, engagement, and skill retention in diverse learning contexts. In conclusion, this work establishes a foundational advancement in automated programming assessment, offering a practical tool for educators to deliver high-quality, equitable, and cognitively diverse evaluations in computer science education.

7.6 Summary

The increasing demand for high-quality and cognitively aligned assessments in programming education presents a significant challenge for educators, particularly within multi-language, large-scale instructional settings. This study introduces a multi-language automatic question generation system leveraging Control Flow Graph (CFG), Program Dependence Graph (PDG), and a synergistic CFG-PDG pipeline to generate diverse, high-quality programming questions directly from source code written in Python, Java, C++, and C. The system systematically covers 19 fundamental algorithms, six Bloom's taxonomy levels, and a range of question types with balanced difficulty distributions. Empirical evaluation shows that the CFG-PDG synergistic pipeline achieved superior performance with an overall quality score of 0.83, linguistic complexity of 0.83, precision of 0.83, and novelty score of 0.96, while maintaining perfect educational alignment (1.00). Compared to CFG-based and PDG-based pipelines, the CFG-PDG approach demonstrated enhanced cognitive diversity (0.31), supporting the generation of questions spanning higher-order cognitive levels and promoting deeper learning engagement. Human evaluations further confirmed the system's pedagogical value, with C++ questions receiving the highest ratings but maintaining consistent quality across all languages. This research contributes to educational technology by operationalizing advanced static analysis for scalable, adaptive, and cognitively rich assessments in programming education. Future work will focus on expanding language coverage, integrating dynamic analysis, and conducting longitudinal studies to evaluate learning impacts. The system offers a practical, effective tool for educators to enhance programming assessment practices, aligning with the evolving demands of computer science education.

Thesis 5: A modular static analysis framework was developed for automatic question generation across multiple programming languages. The system integrates language-specific analyzers within a unified architecture designed to support consistency in question generation. **[P6]**

Chapter 8 Conclusion

8.1 Contributions

This dissertation has established a comprehensive, systematic approach to advancing programming education through automated, high-quality, and pedagogically aligned question generation and learning material creation. Across ontology-based models, hybrid AI frameworks, template-driven static analysis, LLM evaluation, CFG pipeline, PDG pipeline, and CFG–PDG pipeline, the research consistently demonstrates scalable, effective methodologies that address critical gaps in assessment practices within multi-language programming education. The findings provide educators and technology developers with validated, actionable frameworks to enhance learning engagement, assessment quality, and instructional efficiency, paving the way for further innovations in automated programming education tools. The main scientific results achieved during the completion of this research are summarized in five thesis points.

8.1.1 Thesis 1

An ontology-based system was developed to automatically generate programming-related assessment questions directly from source code. The system enables semantic interpretation of programming constructs using structured domain knowledge, supporting concept-aware question generation without relying on adaptive learning mechanisms. **[P1, P2]**

8.1.2 Thesis 2

A hybrid system was developed that combines static code analysis and natural language processing using word embeddings to generate programming-related questions from source code. This approach improves contextual variety and semantic relevance by linking syntactic structures with conceptual representations. **[P3]**

8.1.3 Thesis 3

A systematic evaluation framework was developed to assess the question generation capabilities of large language models, using both automatic and human-centered evaluation metrics. The findings provide insights into their strengths and limitations in generating programming-related assessment questions for potential educational use. **[P4]**

8.1.4 Thesis 4

A modular system was developed for automatic question generation using template-based static code analysis, enabling modular question generation designed to be extensible with minimal integration overhead. The framework supports multiple programming languages through customizable parsing templates within a unified architecture. **[P5]**

8.1.5 Thesis 5

A modular static analysis framework was developed for automatic question generation across multiple programming languages. The system integrates language-specific analyzers within a unified architecture designed to support consistency in question generation. **[P6]**

8.2 Future work

Each of the five thesis points opens up unique and practical directions for continued research. The following recommendations aim to build on their individual contributions, offering ways to refine current methods, broaden their reach, and address some of the open challenges highlighted throughout the work.

1. Ontology-Based Automatic Generation of Learning Materials for Python Programming: Future research could extend the ontology-based approach beyond Python to include a broader range of

programming languages. This would involve designing cross-language ontological frameworks or language-specific extensions that preserve semantic coherence across diverse syntactic constructs. Additionally, conducting controlled experimental studies comparing ontology-generated questions with manually crafted ones could yield valuable insights into their educational effectiveness, particularly in terms of learner comprehension, retention, and perceived usefulness.

2. A Hybrid Approach for Automatic Question Generation from Python Program Codes: One promising direction is to enhance the system's ability to process more complex programming structures, especially those involving third-party libraries, nested functions, and interdependent statements. Improving the semantic interpretation pipeline, possibly by incorporating deeper NLP techniques or lightweight learning models, could help generate more sophisticated and context-aware questions. Future research may also explore how to adapt the system automatically to different code domains or programming paradigms.
3. Evaluating Large Language Models for Code-Based Question Generation in Programming Education: Future work in this area could involve refining the evaluation framework to capture more nuanced aspects of question quality, such as semantic subtlety, creativity, and alignment with pedagogical goals. Incorporating qualitative feedback from educators alongside quantitative metrics could further ground the evaluation process in real instructional needs. Additionally, exploring emerging models, including domain-specific large language models or those designed to support multiple programming languages, may offer deeper insights into their effectiveness across diverse educational contexts.
4. Template-Based Question Generation from Code Using Static Code Analysis: Subsequent research may focus on developing dedicated language-specific parsers for Java, C++, and C to improve upon the current reliance on pattern-based extraction methods. Adding runtime analysis or symbolic execution could improve the system's contextual accuracy and support questions based on actual program behavior. The integration of adaptive or machine learning-driven components might also enable context-sensitive template selection. Longitudinal classroom studies would help assess how such systems impact student learning and engagement over time.
5. Multi-Language Static-Analysis System for Automatic Code-Based Question Generation: Further development could extend the system to include functional, concurrent, and domain-specific languages, making it more adaptable to a wide range of curricular needs. By combining dynamic and static program analysis, the system could generate richer, behavior-aware questions, especially in tasks involving edge-case reasoning or algorithmic logic. Another important direction involves linking the framework with adaptive learning platforms that personalize questions based on individual learner progress. Finally, conducting long-term educational studies would provide essential data on how the system influences knowledge retention, problem-solving skills, and transfer of learning across different instructional settings.

8.3 Author's Publications

Publications Related to the Dissertation

Journal Articles in Q Ranking

[P1] J. Alshboul and E. Baksa-Varga, "Ontology-Based Automatic Generation of Learning Materials for Python Programming," International Journal of Advanced Computer Science and Applications, vol. 16, no. 5, 2025, doi: 10.14569/IJACSA.2025.0160508. Quartile: **Q3**.

[P2] J. Alshboul and E. Baksa-Varga, "A Review of Automatic Question Generation in Teaching Programming," International Journal of Advanced Computer Science and Applications, vol. 13, no. 10, 2022, doi: 10.14569/IJACSA.2022.0131006. Quartile: **Q3**.

[P3] J. Alshboul and E. Baksa-Varga, “A Hybrid Approach for Automatic Question Generation from Program Codes,” International Journal of Advanced Computer Science and Applications, vol. 15, no. 1, 2024, doi: 10.14569/IJACSA.2024.0150102. Quartile: **Q3**.

[P4] J. Alshboul and E. Baksa-Varga, “Evaluating Large Language Models for Code-Based Question Generation in Programming Education,” Status: To Be Submitted (International Review of Applied Sciences and Engineering). Quartile: **Q2**.

[P5] J. Alshboul and E. Baksa-Varga, “Template-Based Question Generation from Code Using Static Code Analysis,” Pollack Periodica: An International Journal for Engineering and Information Sciences, vol. Production Phase, Jun. 2025, doi: 10.1556/606.2025.01413. Quartile: **Q3**.

[P6] J. Alshboul and E. Baksa-Varga, “Multi-Language Static-Analysis System for Automatic Code-Based Question Generation,” Status: To Be Submitted (IEEE Access). Quartile: **Q1**.

Other Publications

Journal Articles in Q Ranking

[P7] S. Mokhtar, J. A. Q. Alshboul, and G. O. A. Shahin, “Towards Data-driven Education with Learning Analytics for Educator 4.0,” Journal of Physics: Conference Series, vol. 1339, no. 1339, p. 012079, Dec. 2019, doi: https://doi.org/10.1088/1742-6596/1339/1/012079. Quartile: **Q4**.

[P8] H. A. A. Ghanim, J. Alshboul, and L. Kovacs, “Development of Ontology-based Domain Knowledge Model for IT Domain in e-Tutor Systems,” International Journal of Advanced Computer Science and Applications, vol. 13, no. 5, 2022, doi: 10.14569/IJACSA.2022.0130505. Quartile: **Q3**.

International Journals

[P9] J. Alshboul, H. A. A. Ghanim, and E. Baksa-Varga, Semantic Modeling for Learning Materials in E-tutor Systems, Journal of Software Engineering & Intelligent Systems 6(2) pp. 1-5. (2021), Journal Article.

Local Journals

[P10] J. Alshboul and E. Baksáné-Varga. “Student Academic Performance Prediction,” Production Systems and Information Engineering, vol. 9, no. 1, pp. 36–53, 2020, Accessed: July. 09, 2025. [Online]. Available: https://ojs.uni-miskolc.hu/index.php/psaie/article/view/3822.

International Conference Proceedings

[P11] 17th Miklós Iványi International Ph.D. & DLA Symposium: Architectural, Engineering and Information Sciences. Title: Development of A Semantic Model for Learning Materials in Intelligent Tutoring Systems. Organizer: Faculty of Engineering and Information Technology, University of Pécs, Pécs, Hungary. Date: 25th-26th October, 2021.

[P12] Language in the Human-Machine Era Training School. Title: E-Learning and Automatic Resource Generation for Learning Materials. Date: 05th to 9th June 2023. Location: University of Pristina, Kosovo. Organizer: EU agency "European Cooperation in Science and Technology".

Local Conference Proceedings

[P13] J. Alshboul and E. Baksáné-Varga. A Survey of Domain Model Representations in Intelligent Tutoring Systems. Miskolc, Hungary: Faculty of Mechanical Engineering and Informatics PhD Forum Proceedings Book, University of Miskolc, 2021.

[P14] J. Alshboul and E. Baksáné-Varga. Code, Feedback, And Question Generation on Programming Topics Using ChatGPT API. Miskolc, Hungary: Faculty of Mechanical Engineering and Informatics PhD Forum Proceedings Book, University of Miskolc, 2023.

Book of Abstract

[P15] J. Alshboul, H. A. A. Ghanim, and E. Baksa-Varga. Development of a Semantic Model for Learning Materials in Intelligent Tutoring Systems, International PhD & DLA Symposium 2021, Pollack Press (2021). pp. 91-91, Abstract.

[P16] J. Alshboul and E. Baksa-Varga. A Generator-Evaluator Framework for Automatic Question Generation from Program Codes, International Conference on AI Transformation 2024, Publisher: Corvinus University of Budapest (2024). pp. 19-20, Abstract.

GitHub Code Repository

1- Chapter3: https://github.com/jalshboul/ontology-based-learning-materials

2- Chapter4: https://github.com/jalshboul/hybrid-question-generation

3- Chapter5: https://github.com/jalshboul/llm-question-generation

4- Chapter6: https://github.com/jalshboul/template-based-question-generation

5- Chapter7: https://github.com/jalshboul/MultiProgLang_CFG_AQG

6- Chapter7: https://github.com/jalshboul/MultiProgLang_PDG_AQG

7- Chapter7: https://github.com/jalshboul/MultiProgLang_CFG-PDG_AQG

References

[1] N. Mulla and P. Gharpure, “Automatic Question Generation: A Review of Methodologies, Datasets, Evaluation Metrics, and Applications,” *Progress in Artificial Intelligence*, vol. 12, no. 1, pp. 1–32, Jan. 2023, doi: 10.1007/s13748-023-00295-9.

[2] D. Vergara, M. L. Fernández, and M. Lorenzo, “Enhancing student motivation in secondary school mathematics courses: A methodological approach,” *Educ. Sci*, vol. 9, no. 2, 2019, doi: 10.3390/educsci9020083.

[3] M. Liu, Y. Ren, L. M. Nyagoga, F. Stonier, Z. Wu, and L. Yu, “Future of education in the era of generative artificial intelligence: Consensus among Chinese scholars on applications of ChatGPT in schools,” *Futur. Educ. Res*, vol. 1, no. 1, pp. 72-101, 2023, doi: 10.1002/fer3.10.

[4] L.-C. Lin, I.-C. Hung, Kinshuk, and N.-S. Chen, “The impact of student engagement on learning outcomes in a cyber-flipped course,” *Educ. Technol. Res. Dev*, vol. 67, pp. 1573-1591, 2019.

[5] W. Villegas-Ch and J. García-Ortiz, “Enhancing Learning Personalization in Educational Environments through Ontology-Based Knowledge Representation,” *Computers*, vol. 12, no. 10, 2023, doi: 10.3390/computers12100199.

[6] N. A. Alrehaili, M. A. Aslam, D. H. Alahmadi, D. A. Alrehaili, M. Asif, and M. S. A. Malik, “Ontology-Based Smart System to Automate Higher Education Activities,” *Complexity*, vol. 2021, 2021, doi: 10.1155/2021/5588381.

[7] Q. U. Ain, M. A. Chatti, K. G. C. Bakar, S. Joarder, and R. Alatrash, “Automatic Construction of Educational Knowledge Graphs: A Word Embedding-Based Approach,” *Inf*, vol. 14, no. 10, 2023, doi: 10.3390/info14100526.

[8] S. MacNeil, *Automatically Generating CS Learning Materials with Large Language Models*, vol. 1, no. 1. Association for Computing Machinery, 2022.

[9] B. Flanagan, G. Akçapinar, R. Majumdar, and H. Ogata, “Automatic generation of contents models for digital learning materials,” in *ICCE 2018 - 26th Int. Conf. Comput. Educ. Main Conf. Proc*, 2018, pp. 804–806.

[10] K. Zhuang, “The Knowledge Graph Construction in the Educational Domain : Take an Australian School Science Course as an Example The Knowledge Graph Construction in the Educational Domain : Take an Australian School Science Course as an Example.” 2023.

[11] G. Kurdi, J. Leo, B. Parsia, U. Sattler, and S. Al-Emari, “A Systematic Review of Automatic Question Generation for Educational Purposes,” *International Journal of Artificial Intelligence in Education*, vol. 30, no. 1, pp. 121–204, Mar. 2020, doi: 10.1007/s40593-019-00186-y.

[12] C. Diwan, S. Srinivasa, G. Suri, S. Agarwal, and P. Ram, “AI-based learning content generation and learning pathway augmentation to increase learner engagement,” *Comput. Educ. Artif. Intell*, vol. 4, no. February, p. 100110, 2022, doi: 10.1016/j.caeai.2022.100110.

[13] C. Pierrakeas, G. Solomou, and A. Kameas, “An ontology-based approach in learning programming languages,” *Proc*, pp. 393-398, 2012, doi: 10.1109/PCi.2012.78.

[14] N. A. Anindyaputri, R. A. Yuana, and P. Hatta, “Enhancing Students’ Ability in Learning Process of Programming Language using Adaptive Learning Systems: A Literature Review,” *Open Eng*, vol. 10, no. 1, pp. 820-829, 2020, doi: 10.1515/eng-2020-0092.

[15] F. D. Calmon, R. Kokku, and A. Vempaty, “Automatic learning curriculum generation,” *Google Patents*, 2019.

[16] T. Guber, “A translational approach to portable ontologies,” *Knowl. Acquis*, vol. 5, no. 2, pp. 199-229, 1993.

[17] K. Chen, Q. Huang, H. Palangi, P. Smolensky, K. Forbus, and J. Gao, “Mapping natural-language problems to formal-language solutions using structured neural representations,” in *International Conference on Machine Learning*, 2020, pp. 1566–1575.

[18] F. Baader, I. Horrocks, C. Lutz, and U. Sattler, *Introduction to description logic*. Cambridge University Press, 2017.

[19] V. Lama, A. Patel, N. C. Debnath, and S. Jain, “IRI_Debug: An Ontology Evaluation Tool,” *New Generation Computing*, vol. 42, no. 1, pp. 177-197, 2024, doi: 10.1007/s00354-024-00246-5.

[20] A. Ramírez-Noriega, “Towards the Automatic Construction of an Intelligent Tutoring System: Domain Module,” *Adv. Intell. Syst. Comput*, vol. 930, no. 3, pp. 293-302, 2019, doi: 10.1007/978-3-030-16181-1_28.

[21] Z. Xia, Y. Zhou, F. Y. Yan, and J. Jiang, “Automatic curriculum generation for learning adaptation in networking.” 2022.

[22] P. Brusilovsky, B. J. Ericson, C. Zilles, C. S. Horstmann, C. Servin, and F. Vahid, “The Future of Computing Education Materials,” *Comput. Sci. Curricula, Curricula Pract*, vol. 1, no. 1, pp. 1-8, 2023.

[23] N. C. Debnath and A. Patel, “Ontology Evaluation Tools: Current and Future Research,” *Recent Adv. Comput. Sci. Commun*, 2022, [Online]. Available: https://api.semanticscholar.org/CorpusID:248138690.

[24] T. Urazova, “Building a System for Automated Question Generation and Evaluation to Assist Students Learning UML Database Design,” University of British Columbia, 2022. [Online]. Available: https://open.library.ubc.ca/soa/cIRcle/collections/undergraduateresearch/52966/items/1.0413656

[25] S. Russell, “Automated Code Tracing Exercises for CS1,” presented at the Computing Education Practice 2022, Durham, United Kingdom: ACM, Jan. 2022, pp. 13–16. doi: https://doi.org/10.1145/3498343.3498347.

[26] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, “Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models,” presented at the International Computing Education Research, Lugano, Switzerland: ACM, Aug. 2022, pp. 27–43. doi: https://doi.org/10.1145/3501385.3543957.

[27] M. Sh. Murtazina and T. V. Avdeenko, “The Constructing of Cognitive Functions Ontology,” presented at the 14th International Symposium "Intelligent Systems, Moscow, Russia: Procedia Computer Science, 2021, pp. 595–602. doi: https://doi.org/10.1016/j.procs.2021.04.181.

[28] M. Alqaradaghi, G. Morse, and T. Kozsik, “Detecting Security Vulnerabilities with Static Analysis - A Case Study,” *Pollack Periodica*, vol. 17, no. 2, pp. 1–7, Sep. 2021, doi: 10.1556/606.2021.00454.

[29] K. Sterner, “Automated Checking of Programming Assignments Using Static Analysis,” Mälardalens University, Sweden, 2021. Accessed: Apr. 25, 2025. [Online]. Available: https://mdh.diva-portal.org/smash/record.jsf?pid=diva2%3A1526100&dswid=6624

[30] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, “BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection,” *Information and Software Technology*, vol. 136, p. 106576, 2021, doi: https://doi.org/10.1016/j.infsof.2021.106576.

[31] R. Zviel-Girshin, “The Good and Bad of AI Tools in Novice Programming Education,” *Education Sciences*, vol. 14, no. 10, 2024, doi: 10.3390/educsci14101089.

[32] S. Srikant and V. Aggarwal, “A System to Grade Computer Programming Skills using Machine Learning,” in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 1887–1896. doi: http://dx.doi.org/10.1145/2623330.2623377.

[33] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to Represent Programs with Graphs,” May 04, 2018. doi: https://doi.org/10.48550/arXiv.1711.00740.

[34] T. H. M. Le, H. Chen, and M. A. Babar, “Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges,” *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020, doi: 10.1145/3383458.

[35] S. Bhatia, P. Kohli, and R. Singh, “Neuro-Symbolic Program Corrector for Introductory Programming Assignments,” in *Proceedings of the 40th International Conference on Software Engineering*, in ICSE ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 60–70. doi: 10.1145/3180155.3180219.

[36] N. Emamdoost, “Better Program Analysis for Security via Data Flow Tracking and Symbolic Execution,” PhD Thesis, 2021. Accessed: Jun. 06, 2025. [Online]. Available: https://hdl.handle.net/11299/225000

[37] Z. Wang, L. Yu, S. Wang, and P. Liu, “Spotting Silent Buffer Overflows in Execution Trace through Graph Neural Network Assisted Data Flow Analysis,” Feb. 20, 2021, *ArXiv*. doi: https://doi.org/10.48550/arXiv.2102.10452.

[38] W. Hasselbring, M. Wojcieszak, and S. Dustdar, “Control Flow Versus Data Flow in Distributed Systems Integration: Revival of Flow-Based Programming for the Industrial Internet of Things,” *IEEE Internet Computing*, vol. 25, no. 4, pp. 5–12, 2021, doi: 10.1109/MIC.2021.3053712.

[39] D. Guo *et al.*, “GraphCodeBERT: Pre-training Code Representations with Data Flow,” in *ICLR 2021*, Vienna, Austria, May 2021. doi: https://doi.org/10.48550/arXiv.2009.08366.

[40] B. Steenhoek, H. Gao, and W. Le, “Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, in ICSE ’24. New York, NY, USA: Association for Computing Machinery, 2024. doi: 10.1145/3597503.3623345.

[41] J. Phillips, A. Sudarsanam, H. Samala, R. Kallam, J. Carver, and A. Dasu, “Methodology To Derive Context Adaptable Architectures for FPGAS,” *IET Computers & Digital Techniques*, vol. 3, no. 1, pp. 124–141, Jan. 2009, doi: 10.1049/iet-cdt:20070099.

[42] K. C. Swarna, N. S. Mathews, D. Vagavolu, and S. Chimalakonda, "On The Impact of Multiple Source Code Representations on Software Engineering Tasks — An Empirical Study," *Journal of Systems and Software*, vol. 210, p. 111941, 2024, doi: https://doi.org/10.1016/j.jss.2023.111941.

[43] A. S. Saimbhi, "Enhancing Software Vulnerability Detection Using Code Property Graphs and Convolutional Neural Networks," in *2025 International Conference on Computational, Communication and Information Technology (ICCCIT)*, Indore, India: IEEE, 2025, pp. 435–440. doi: 10.1109/ICCCIT62592.2025.10928033.

[44] G. Kurdi, J. Leo, B. Parsia, U. Sattler, and S. Al-Emari, "A Systematic Review of Automatic Question Generation for Educational Purposes," *International Journal of Artificial Intelligence in Education*, vol. 30, no. 1, pp. 121–204, Mar. 2020, doi: 10.1007/s40593-019-00186-y.

[45] N. Willert and J. Thiemann, "Template-Based Generator for Single-Choice Questions," *Technology, Knowledge and Learning*, vol. 29, no. 1, pp. 355–370, Mar. 2024, doi: 10.1007/s10758-023-09659-5.

[46] L. L. Shwe, S. Matayong, and S. Witosurapot, "Enabling Cognitive and Unified Similarity-Based Difficulty Ranking Mechanisms for AQG On Multimedia Content," *Expert Systems with Applications*, vol. 277, p. 127244, Jun. 2025, doi: 10.1016/j.eswa.2025.127244.

[47] B. Khoy, "Unlocking Cognitive Learning Objectives: A Comprehensive Evaluation of How Textbooks and Syllabi Align with Revised Bloom's Taxonomy Across Disciplines," *Curriculum Perspectives*, Jan. 2025, doi: 10.1007/s41297-024-00295-2.

[48] A. Luxton-Reilly *et al.*, "Developing Assessments to Determine Mastery of Programming Fundamentals," in *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, in ITiCSE-WGR '17. New York, NY, USA: Association for Computing Machinery, 2018, pp. 47–69. doi: 10.1145/3174781.3174784.

[49] G. Deena, K. Raja, and K. Kannan, "An Automatic Question Generation System using Rule-Based Approach in Bloom's Taxonomy," *Recent Advances in Computer Science and Communications*, vol. 14, no. 5, pp. 1477–1487, 2019, doi: 10.2174/2213275912666191113143335.

[50] 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, Apr. 2024, doi: 10.1080/03004279.2024.2332469.

[51] M. Shoaib, G. Husnain, N. Sayed, Y. Yasin Ghadi, M. Alajmi, and A. Qahmash, "Automated Generation of Multiple-Choice Questions for Computer Science Education Using Conditional Generative Adversarial Networks," *IEEE Access*, vol. 13, pp. 16697–16715, 2025, doi: 10.1109/ACCESS.2025.3530474.

[52] N. Liu, Z. Wang, R. Baraniuk, and A. Lan, "Open-ended Knowledge Tracing for Computer Science Education," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Y. Goldberg, Z. Kozareva, and Y. Zhang, Eds., Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, Dec. 2022, pp. 3849–3862. doi: 10.18653/v1/2022.emnlp-main.254.

[53] Y. Wu *et al.*, "Programming knowledge tracing based on heterogeneous graph representation," *Knowledge-Based Systems*, vol. 300, p. 112161, 2024, doi: https://doi.org/10.1016/j.knosys.2024.112161.

[54] O. H. T. Lu, A. Y. Q. Huang, D. C. L. Tsai, and S. J. H. Yang, "Expert-Authored and Machine-Generated Short-Answer Questions for Assessing Students Learning Performance," *Educational Technology & Society*, vol. 24, no. 3, pp. 159–173, 2021.

[55] M. Hassan and C. Zilles, "On Students' Usage of Tracing for Understanding Code," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, in SIGCSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 129–136. doi: 10.1145/3545945.3569741.

[56] M. Murata, N. Kato, M. Ohtsuki, and T. Kakeshita, "Fill-in-the-blank Questions for Object-Oriented Programming Education and Its Preliminary Evaluation," *International Journal of Learning Technologies and Learning Environments*, vol. 6, pp. 1–1, Jan. 2023, doi: 10.52731/ijltle.v6.i1.699.

[57] S. Kyaw, Nobuo Funabiki, and W.-C. Kao, "A Proposal of Code Amendment Problem in Java Programming Learning Assistant System," *International Journal of Information and Education Technology*, vol. 10, no. 10, pp. 751–756, 2020, doi: 10.18178/ijiet.2020.10.10.1453.

[58] T. Terroso and M. Pinto, "Programming for Non-Programmers: An Approach Using Creative Coding in Higher Education," in *Third International Computer Programming Education Conference (ICPEC 2022)*, A. Simões and J. C. Silva, Eds., in Open Access Series in Informatics (OASIcs), vol. 102. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, p. 13:1-13:8. doi: 10.4230/OASIcs.ICPEC.2022.13.

[59] I. Lauriola, A. Lavelli, and F. Aiolli, “An introduction to deep learning in natural language processing: Models, techniques, and tools,” *Neurocomputing*, vol. 470, pp. 443–456, 2022.

[60] Y. Kang, Z. Cai, C.-W. Tan, Q. Huang, and H. Liu, “Natural language processing (NLP) in management research: A literature review,” *Journal of Management Analytics*, vol. 7, no. 2, pp. 139–172, May 2020, doi: 10.1080/23270012.2020.1756939.

[61] J. Eisenstein, *Introduction to natural language processing*. MIT Press, 2019.

[62] H. Liu, R. Ning, Z. Teng, J. Liu, Q. Zhou, and Y. Zhang, “Evaluating the logical reasoning ability of chatgpt and gpt-4,” May 05, 2023, *ArXiv*. doi: 10.48550/arXiv.2304.03439.

[63] S. L. Blodgett, S. Barocas, H. Daumé III, and H. Wallach, “Language (Technology) is Power: A Critical Survey of ‘Bias’ in NLP,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, Jul. 2020, pp. 5454–5476. doi: 10.18653/v1/2020.acl-main.485.

[64] J. Wei *et al.*, “Emergent abilities of large language models,” *arXiv preprint arXiv:2206.07682*, 2022.

[65] M. Zhou, N. Duan, S. Liu, and H.-Y. Shum, “Progress in Neural NLP: Modeling, Learning, and Reasoning,” *Engineering*, vol. 6, no. 3, pp. 275–290, 2020, doi: 10.1016/j.eng.2019.12.014.

[66] H. Naveed *et al.*, “A Comprehensive Overview of Large Language Models,” Jul. 2023.

[67] M. Mitchell and D. C. Krakauer, “The debate over understanding in AI’s large language models,” *Proceedings of the National Academy of Sciences*, vol. 120, no. 13, p. e2215907120, Feb. 2023, doi: 10.1073/pnas.2215907120.

[68] D. Sykes *et al.*, “Comparison of rule-based and neural network models for negation detection in radiology reports,” *Natural Language Engineering*, vol. 27, no. 2, pp. 203–224, Mar. 2021, doi: 10.1017/S1351324920000509.

[69] L. Ouyang *et al.*, “Training language models to follow instructions with human feedback,” in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, New Orleans LA USA: Curran Associates Inc., Nov. 2022, pp. 27730–27744.

[70] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, “Challenges and Applications of Large Language Models,” Jul. 2023.

[71] M. Chen *et al.*, “Evaluating large language models trained on code,” Jul. 14, 2021, *arXiv*. doi: 10.48550/arXiv.2107.03374.

[72] A. Sottana, B. Liang, K. Zou, and Z. Yuan, “Evaluation Metrics in the Era of GPT-4: Reliably Evaluating Large Language Models on Sequence to Sequence Tasks,” Oct. 2023.

[73] Z. Guo *et al.*, “Evaluating Large Language Models: A Comprehensive Survey,” Nov. 25, 2023, *ArXiv*. doi: 10.48550/arXiv.2310.19736.

[74] A. Mohammadshahi *et al.*, “RQUGE: Reference-Free Metric for Evaluating Question Generation by Answering the Question,” in *Findings of the Association for Computational Linguistics: ACL 2023*, Toronto, Canada: Association for Computational Linguistics, 2023, pp. 6845–6867. [Online]. Available: https://aclanthology.org/2023.findings-acl.428/

[75] C. Kooli, “Chatbots in Education and Research: A Critical Examination of Ethical Implications and Solutions,” *Sustainability*, vol. 15, no. 7, p. 5614, Mar. 2023, doi: 10.3390/su15075614.

[76] D. Hupkes *et al.*, “A taxonomy and review of generalization research in NLP,” *Nature Machine Intelligence*, vol. 5, pp. 1161–1174, Oct. 2023, doi: 10.1038/s42256-023-00729-y.

[77] L. Moussiades and G. Zografos, “OpenAi’s GPT4 as coding assistant,” Sep. 22, 2023, *ArXiv*. doi: 10.48550/arXiv.2309.12732.

[78] J. A. Baktash and M. Dawodi, “Gpt-4: A Review on Advancements and Opportunities in Natural Language Processing,” May 04, 2023, *ArXiv*. doi: 10.48550/arXiv.2305.03195.

[79] A. Belfathi, N. Hernandez, and L. Monceaux, “Harnessing GPT-3.5-Turbo for Rhetorical Role Prediction in Legal Cases,” Oct. 26, 2023, *arXiv*. doi: 10.48550/arXiv.2310.17413.

[80] H. Touvron *et al.*, “Llama: Open and efficient foundation language models,” Feb. 27, 2023, *ArXiv*. doi: 10.48550/arXiv.2302.13971.

[81] A. Candel *et al.*, “h2oGPT: Democratizing Large Language Models,” Jun. 16, 2023, *ArXiv*. doi: 10.48550/arXiv.2306.08161.

[82] Hugging Face, “Hugging Face Models,” Hugging Face. Accessed: Jan. 12, 2025. [Online]. Available: https://huggingface.co/models

[83] *Vicuna: An Instruction-following LLaMA-based Model*. (2023). [Chinese-Vicuna]. Available: https://github.com/Facico/Chinese-Vicuna

[84] L. Caruccio, S. Cirillo, G. Polese, G. Solimando, S. Sundaramurthy, and G. Tortora, “Claude 2.0 large language model: Tackling a real-world classification problem with a new iterative prompt engineering approach,” *Intelligent Systems with Applications*, vol. 21, p. 200336, Mar. 2024, doi: 10.1016/j.iswa.2024.200336.

[85] Anthropic, “Claude 2 Anthropic.” [Online]. Available: https://www.anthropic.com/news/claude-2

[86] E. Portakal, “Claude 2 Parameters (Parameter Size, Context Window.),” TextCortex AI. [Online]. Available: https://textcortex.com/post/claude-2-parameters

[87] Y. Susanti, T. Tokunaga, H. Nishikawa, and H. Obari, “Evaluation of Automatically Generated English Vocabulary Questions,” *Research and Practice in Technology Enhanced Learning*, vol. 12, no. 1, p. 11, Mar. 2017, doi: 10.1186/s41039-017-0051-y.

[88] T. Song, Q. Tian, Y. Xiao, and S. Liu, “Automatic Generation of Multiple-Choice Questions for CS0 and CS1 Curricula Using Large Language Models,” in *Computer Science and Education. Computer Science and Technology*, W. Hong and G. Kanaparan, Eds., Singapore: Springer Nature Singapore, 2024, pp. 314–324.

[89] B. Abu-Salih and S. Alotaibi, “A systematic literature review of knowledge graph construction and application in education,” *Heliyon*, vol. 10, no. 3, p. 25383, 2024, doi: 10.1016/j.heliyon.2024.e25383.

[90] E. Rajabi and K. Etminani, “Knowledge-graph-based explainable AI: A systematic review,” *J. Inf. Sci*, 2022, doi: 10.1177/01655515221112844.

[91] L. N. Nongkhai, J. Wang, and T. Mendori, “Developing An Ontology of Multiple Programming Languages from The Perspective of Computational Thinking Education,” in *Proceeedings of the 19th International Conference on Cognition and Exploratory Learning in the Digital Age (CELDA 2022)*, Lisbon, Portugal: International Association for Development of the Information Society (IADIS), 2022, pp. 66–72. doi: 10.33965/celda2022_202207l009.

[92] W. Nie, K. Vita, and T. Masood, “An ontology for defining and characterizing demonstration environments,” *J. Intell. Manuf*, 2023, doi: 10.1007/s10845-023-02213-1.

[93] W. Yathongchai, J. Angskun, and C. C. Fung, “An Ontology Model for Developing a SQL Personalized Intelligent Tutoring System,” *Naresuan Univ. J. Sci. Technol*, vol. 25, no. 4, pp. 88-96, 2017.

[94] A. Fernández-Izquierdo and R. García-Castro, “Themis: A tool for validating ontologies through requirements,” in *Proc. Int. Conf. Softw. Eng. Knowl. Eng. SEKE*, 2019, pp. 573-578,.

[95] M. Poveda-Villalón, M. C. Suárez-Figueroa, and A. Gómez-Pérez, “Validating Ontologies with OOPS ! State of the Art,” *Knowl. Eng. Knowl. Manag*, pp. 267-281, 2012.

[96] J. Alshboul, “Ontology Generation and Ontology Data Set.” Accessed: Apr. 24, 2025. [Online]. Available: https://github.com/jalshboul/Python-Ontology-GLM

[97] Y. Ham and B. Myers, “Supporting Guided Inquiry with Cooperative Learning in Computer Organization,” in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, Minneapolis, USA: ACM, Feb. 2019, pp. 273–279. doi: https://doi.org/10.1145/3287324.3287355.

[98] R. S. J. d Baker, A. T. Corbett, and V. Aleven, “More Accurate Student Modeling through Contextual Estimation of Slip and Guess Probabilities in Bayesian Knowledge Tracing,” presented at the International Conference on Intelligent Tutoring Systems, in Lecture Notes in Computer Science, vol. 5091. Montreal, Canada: Springer Berlin Heidelberg, Jun. 2008, pp. 406–415. doi: https://doi.org/10.1007/978-3-540-69132-7_44.

[99] C.-Y. Chung and I.-H. Hsiao, “Investigating Patterns of Study Persistence on Self-Assessment Platform of Programming Problem-Solving,” in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, ACM, Feb. 2020, pp. 162–168. doi: https://doi.org/10.1145/3328778.3366827.

[100] C.-Y. Chung, C. Y. C. Edu, and I.-H. Hsiao, “From Detail to Context: Modeling Distributed Practice Intensity and Timing by Multiresolution Signal Analysis,” presented at the 14th International Conference on Educational Data Mining, Virtual: International Educational Data Mining Society, Jul. 2021. [Online]. Available: https://educationaldatamining.org/edm2021/

[101] P. Brusilovsky, M. Yudelson, and I.-H. Hsiao, “Problem Solving Examples as First Class Objects in Educational Digital Libraries: Three Obstacles to Overcome Problem Solving Examples as Interactive Learning Objects for

Educational Digital Libraries," *Journal of Educational Multimedia and Hypermedia*, vol. 18, no. 3, pp. 267–288, Jul. 2009.

[102] R. Cafolla, "Project MERLOT: Bringing Peer Review to Web-Based Educational Resources," *Journal of Information Technology for Teacher Education*, vol. 14, no. 2, Apr. 2006.

[103] H. K. M. Al-Chalabi, "Evaluation of a Multi-Parameter E-learning System using Web 3.0 Technologies," presented at the 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), Pitesti, Romania: IEEE, Jul. 2021, pp. 1–4. doi: https://doi.org/10.1109/ECAI52376.2021.9515191.

[104] H. K. M. Al-Chalabi and U. C. Apoki, "A Semantic Approach to Multi-parameter Personalisation of E-Learning Systems," presented at the International Conference on Modelling and Development of Intelligent Systems, in Communications in Computer and Information Science, vol. 1341. Sibiu, Romania: Springer International Publishing, 2021, pp. 381–393. doi: https://doi.org/10.1007/978-3-030-68527-0_24.

[105] P. Denny, A. Luxton-Reilly, and J. Hamer, "The PeerWise System of Student Contributed Assessment Questions," in *Proceedings of the tenth conference on Australasian computing education*, Wollongong, Australia, Jan. 2008, pp. 69–74. doi: https://dl.acm.org/doi/10.5555/1379249.1379255.

[106] N. Mulla and P. Gharpure, "Automatic Question Generation: A Review of Methodologies, Datasets, Evaluation Metrics, and Applications," *Progress in Artificial Intelligence*, vol. 12, no. 1, pp. 1–32, Jan. 2023, doi: https://doi.org/10.1007/s13748-023-00295-9.

[107] R. G. Golla, V. Tiwari, P. Chokhra, and H. Okada, "QuestGen AI." [Online]. Available: https://github.com/ramsrigouthamg/Questgen.ai

[108] J. Li, T. Tang, W. X. Zhao, J.-Y. Nie, and J.-R. Wen, "Pretrained Language Models for Text Generation: A Survey," May 13, 2022, *ArXiv*. doi: 10.48550/arXiv.2201.05273.

[109] X.-Q. Dao, "Performance Comparison of Large Language Models on VNHSGE English Dataset: OpenAI ChatGPT, Microsoft Bing Chat, and Google Bard," Jul. 20, 2023, *ArXiv*. doi: 10.48550/arXiv.2307.02288.

[110] A. Koubaa, "GPT-4 vs. GPT-3.5: A concise showdown," Apr. 07, 2023, *TechRxiv*. doi: 10.36227/techrxiv.22312330.v2.

[111] J. Alshboul, "Generator-Evaluator: Dataset-Codes," *GitHub Dataset*. GitHub, 2025. Accessed: Jun. 20, 2025. [Online]. Available: https://github.com/jalshboul/Generator-Evaluator

[112] United States Chess Federation, "Approximating formulas for the US Chess rating system." Apr. 2017. [Online]. Available: http://www.glicko.net/ratings/approx.pdf

[113] S. Maity and A. Deroy, "The Future of Learning in the Age of Generative AI: Automated Question Generation and Assessment with Large Language Models," Oct. 12, 2024, *ArXiv*. doi: 10.48550/arXiv.2410.09576.

[114] A. Tran, K. Angelikas, E. Rama, C. Okechukwu, D. H. Smith, and S. MacNeil, "Generating Multiple Choice Questions for Computing Courses Using Large Language Models," in *2023 IEEE Frontiers in Education Conference (FIE)*, Oct. 2023, pp. 1–8. doi: 10.1109/FIE58773.2023.10342898.

[115] J. Doughty *et al.*, "A Comparative Study of AI-Generated (GPT-4) and Human-crafted MCQs in Programming Education," in *Proceedings of the 26th Australasian Computing Education Conference*, in ACE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 114–123. doi: 10.1145/3636243.3636256.

[116] S. Baral, E. Worden, W.-C. Lim, Z. Luo, C. Santorelli, and A. Gurung, "Automated Assessment in Math Education: A Comparative Analysis of LLMs for Open-Ended Responses," in *Proceedings of the 17th International Conference on Educational Data Mining*, B. PaaÃŸen and C. D. Epp, Eds., Atlanta, Georgia, USA: International Educational Data Mining Society, Jul. 2024, pp. 732–737. doi: 10.5281/zenodo.12729932.

[117] P. Kargupta, I. Agarwal, D. H. Tur, and J. Han, "Instruct, Not Assist: LLM-based Multi-Turn Planning and Hierarchical Questioning for Socratic Code Debugging," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds., Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 9475–9495. doi: 10.18653/v1/2024.findings-emnlp.553.

[118] 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*, in SLE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 56–69. doi: 10.1145/3687997.3695644.

[119] G. Son, H. Ko, H. Lee, Y. Kim, and S. Hong, “Multilingual Challenges in Automated Evaluators: A Case Study on English and Korean,” OpenReview. Accessed: May 16, 2025. [Online]. Available: https://openreview.net/forum?id=8NIUi6Ha1f

[120] M. Hidvégi, G. Mezei, and S. Bácsi, “The Challenges of Visualizing DMLA Models,” *Pollack Periodica*, vol. 16, no. 3, pp. 13–19, 2021, doi: 10.1556/606.2021.00345.

[121] 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*, in SIGCSE ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 704. doi: 10.1145/3017680.3022440.

[122] S. Rakangor and Y. Ghodasara, “Literature Review of Automatic Question Generation Systems,” *International Journal of Scientific and Research Publications*, vol. 5, no. 1, 2015, Accessed: Jan. 01, 2025. [Online]. Available: https://www.ijsrp.org/research-paper-0115/ijsrp-p3757.pdf

[123] Paul Jansen, “The TIOBE Programming Community Index,” Tiobe.com. Accessed: May 16, 2025. [Online]. Available: https://www.tiobe.com/tiobe-index/

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

[125] K. A. Mills, J. Cope, L. Scholes, and L. Rowe, “Coding and Computational Thinking Across the Curriculum: A Review of Educational Outcomes,” *Review of Educational Research*, vol. 95, no. 3, pp. 581–618, 2025, doi: 10.3102/00346543241241327.

[126] J. Savelka, A. Agarwal, C. Bogart, and M. Sakr, “Large Language Models (GPT) Struggle to Answer Multiple-Choice Questions about Code,” Mar. 09, 2023. doi: https://doi.org/10.48550/arXiv.2303.08033.

[127] P. V. L. Pham, A. V. Duc, N. M. Hoang, X. L. Do, and A. T. Luu, “ChatGPT as a Math Questioner? Evaluating ChatGPT on Generating Pre-university Math Questions,” in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, in SAC ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 65–73. doi: 10.1145/3605098.3636030.

[128] S. Ito, “Semantical Equivalence of the Control Flow Graph and the Program Dependence Graph,” Mar. 08, 2018. doi: https://doi.org/10.48550/arXiv.1803.02976 Focus to learn more.

[129] J. Prather *et al.*, “‘It’s Weird That it Knows What I Want’: Usability and Interactions with Copilot for Novice Programmers,” *ACM Trans. Comput.-Hum. Interact.*, vol. 31, no. 1, Nov. 2023, doi: 10.1145/3617367.

[130] Q. Zhang, C. Fang, Y. Shang, T. Zhang, S. Yu, and Z. Chen, “No Man is an Island: Towards Fully Automatic Programming by Code Search, Code Generation and Program Repair,” Sep. 05, 2024. doi: https://doi.org/10.48550/arXiv.2409.03267.

[131] Q. Zhu and W. Zhang, “Code Generation Based on Deep Learning: a Brief Review,” Jul. 04, 2021. doi: arXiv:2106.08253v4.