UNIVERSITY OF MISKOLC


Faculty of Mechanical Engineering and Informatics

PhD Dissertation Booklet

**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

Table of Contents

# 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 [1], [P2]. Figure 1.1 illustrates the conceptual framework of AQG from source code. The system takes source code as input, processes it through computational analysis and generation techniques, and automatically produces relevant questions for educational or assessment purposes. Figure 1.2 illustrates the four-component architecture of Intelligent Tutoring Systems (ITS) as discussed by the review article [2]. This dissertation focuses specifically on the Domain Model component through AQG for programming education. This work contributes to the foundational knowledge representation layer by developing methods to automatically generate contextually appropriate programming questions that can be integrated into the broader tutoring system architecture.

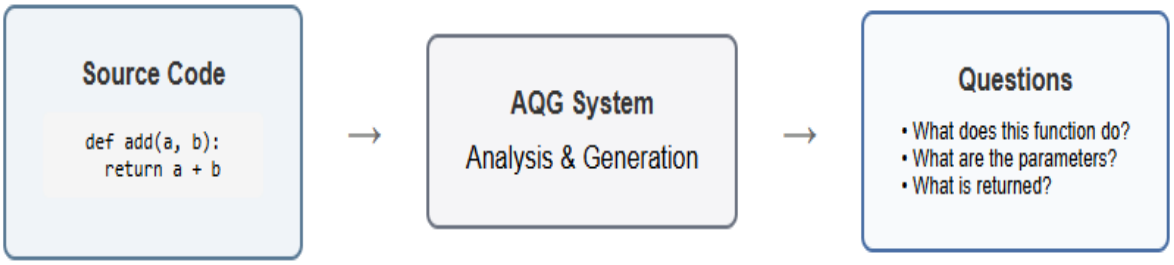


*Figure 1.1 Conceptual framework of AQG from source code*

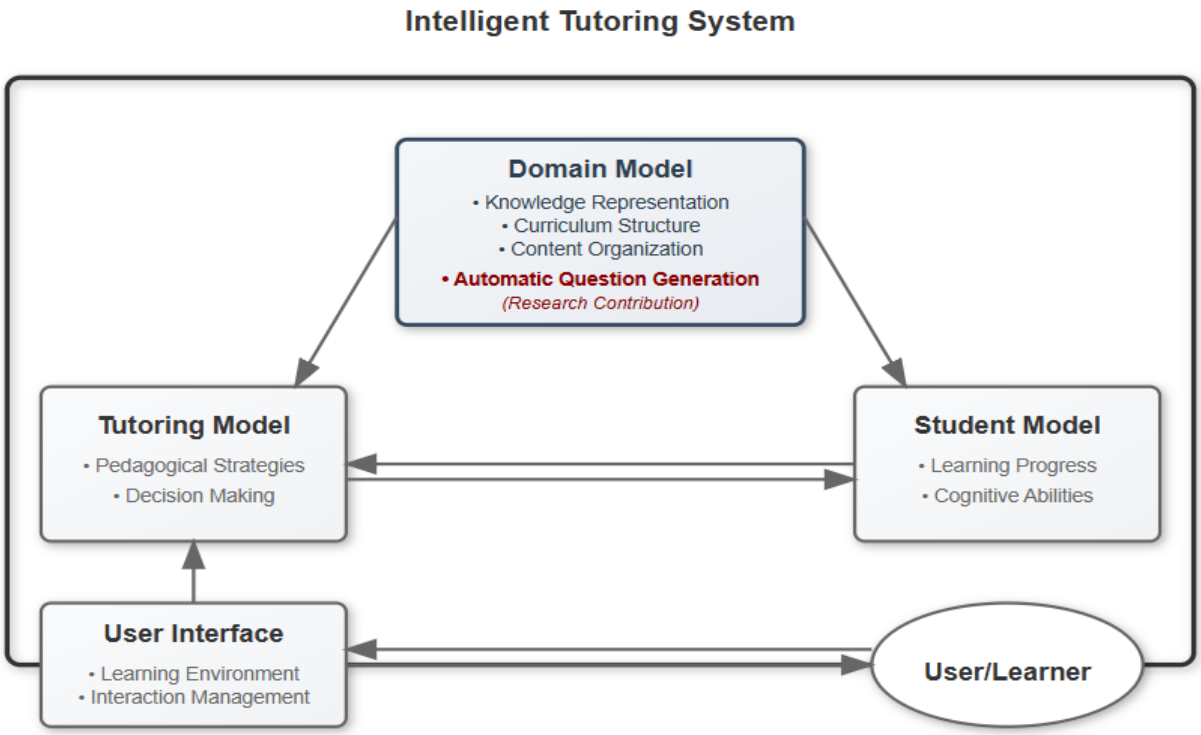


*Figure 1.2 The four-component ITS architecture*

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 [3]. 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. Although recent AQG studies have primarily focused on generating questions from natural language texts and, to a lesser extent, visual data [1], [4], [5], AQG from source code remains underexplored despite its potential to transform programming education. Academic programming textbooks typically include text, images, and code examples, yet most AQG systems rely heavily on NLP techniques for text-based question generation (QG), with limited exploration of visual content [1], [P3]. The review paper [P2] advocates for developing QG methods tailored to programming topics, along with appropriate evaluation criteria. 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 [6], difficult to standardize across diverse learners and languages [6], [7], and often fall short of covering the full spectrum of cognitive skills outlined in Bloom's Taxonomy [8]. Moreover, they tend to lack scalability, particularly in large or multi-language educational settings where hundreds of students may require tailored assessment materials [7]. 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 [9]. This dissertation responds to that demand by presenting a unified exploration of five distinct yet complementary approaches: ontology-driven QG [10], [11], hybrid semantic-to-question modeling [9], template-based multi-language QG via static code analysis [12], evaluation of large language models (LLMs) for QG from source code [13], and a comprehensive multi-language assessment system powered by Control Flow Graphs (CFGs) and Program Dependence Graphs (PDGs). Collectively, these approaches constitute the novel contributions of this work. I extend beyond traditional template or ontology-based systems by incorporating formal semantic graph representations, namely CFGs and PDGs, to anchor QG in actual program structure and behavior. CFGs model possible execution paths and dependencies across program blocks [14], while PDGs capture both control and data dependencies among statements [15], providing a richer semantic foundation for QG. Each approach contributes to a shared objective: to automate programming QG in a pedagogically grounded, cognitively stratified (Organizing learning or assessment tasks by levels of thinking, from simple recall to complex problem-solving), and linguistically inclusive way [7], [16].

### **1.2** Problem Statement

The global expansion of computer science (CS) education has intensified the need for scalable, high-quality assessment tools that can effectively serve diverse learners across various programming languages [3], [7]. 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 [6], [8], [17]. AQG has emerged as a promising approach for scalable assessment across educational contexts [1], [4], [5]. 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 [4], [18], [19]. Similarly, visual QG has seen growing attention, particularly for generating questions from images and, more recently, educational diagrams, leveraging advancements in multimodal learning [20], [21]. These areas have established robust evaluation practices and benchmarks, fueling rapid progress and adoption [22], [23]. In contrast, QG from source code remains significantly underrepresented despite its critical potential in programming education [9], [12], [18]. Generating meaningful and pedagogically aligned questions directly from source code presents unique challenges, including understanding code semantics [14], [15], aligning questions with relevant programming concepts [9], [12], and ensuring cognitive coverage across difficulty levels [8], [17], [24]. The lack of standardized datasets and well-defined evaluation metrics further impedes systematic advancements in this domain [13], [22], [23].

Most existing AQG research has overlooked this research gap in programming education assessment, and only a few recent studies have begun exploring it, often in isolated or single-language contexts [9], [12], [18], leaving a substantial gap in the scalable assessment needs of programming education. To clarify, generating programming questions directly from raw, multi-language source code requires integrated semantic parsing (AST/CFG/PDG), multi-language normalization, deliberate Bloom-level coverage, diverse code-centric question types, and multi-metric evaluation. These requirements are largely absent in existing primarily text-focused or single-language ontology/LLM studies, leaving the domain underdeveloped and limiting scalable programming assessment. 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 [3], [7], [17], [25]. Advancing AQG from code requires not only robust generation methods that capture the semantics of source code [14], [15], but also the development of principled evaluation frameworks tailored to the unique requirements of programming education [13], [22], [23]. This dissertation aims to address these gaps to advance scalable, high-quality, and pedagogically aligned AQG systems that support equitable programming education worldwide.

### **1.3** Research Objectives

This dissertation seeks to address the limitations of current programming assessment methods by pursuing the following core objectives:

1. To design and implement models that automatically generate programming questions directly from source code.
2. To ensure systematic alignment of generated questions with cognitive learning frameworks, particularly Bloom's Taxonomy.
3. To support multiple programming languages (Python, Java, C++, and C) within a unified, multi-language assessment context.
4. To evaluate both the technical quality and the pedagogical value of generated questions through automated metrics and expert review.

Together, these objectives establish the foundation of this dissertation's contribution to advancing programming education assessment through AI-enhanced, source code–driven QG and evaluation.

## **2 Thesis 1:** Ontology-Based Automatic Generation of Learning Materials for Python Programming

### **2.1** Introduction

*I developed an ontology-based system that automatically generates programming-related assessment questions directly from source code. By leveraging structured domain knowledge, the system semantically interprets programming constructs to support concept-aware question generation, without relying on adaptive learning mechanisms* **[P1, P2]**.

The objectives of this research are to design an ontology-based framework that models Python programming concepts and their interconnections, and develop a system for automatically generating Python programming learning materials (specifically quizzes) that align with the modeled concepts and relationships. It supports beginner, intermediate, and advanced difficulty levels.

### **2.2** Methodology

Algorithm 2.1 automatically generates MCQs quizzes aligned with Python programming concepts using a domain-specific ontology. It aims to deliver personalized 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 MCQs bank. This bank is then saved in a comma separated values (CSV) format for efficient retrieval and further processing. When a learner requests a quiz, the system loads

the MCQs 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.

Algorithm 2.1: 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 ← ∅
    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
```

# 3 Thesis 2: A Hybrid Approach for Automatic Question Generation from Program Codes

## 3.1 Introduction

*I developed a hybrid system that combines static code analysis, ontology, and natural language processing using word embeddings to generate programming-related questions from source code* **[P3]**.

This thesis 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 thesis 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 AQG 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 thesis 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 for AQG.

## 3.2 Methodology

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. 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" [26]. This model will generate Boolean, short-answer, and open-ended questions. Figure 3.1 shows the framework data flow and its components. 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.

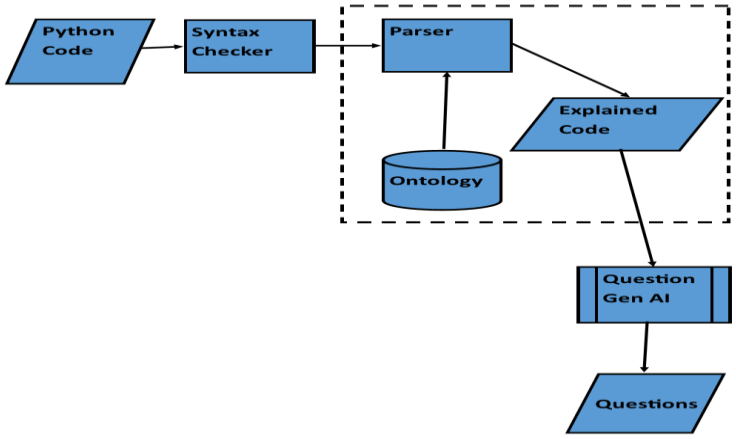


*Figure 3.1 Proposed framework architecture*

Over time, there is a growing demand for QG, 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 3.2 shows the whole procedure for translating code into text. In Figure 3.2, 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 QG AI model to generate reasonable questions. An explanation of the QG AI model is provided in the subsequent section.

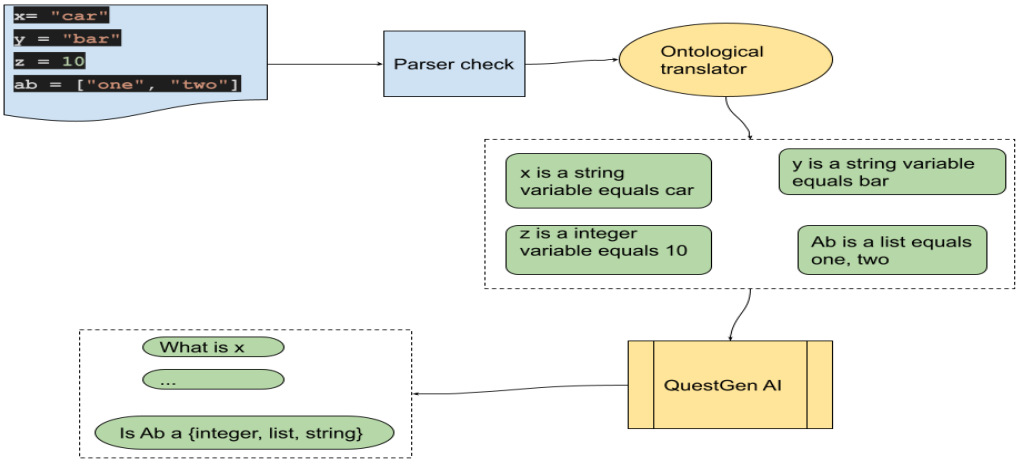


*Figure 3.2 Question-generation process*

Algorithm 3.1 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 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 Boolean, short-answer, and open-ended questions.

Algorithm 3.1: Hybrid Approach for QG 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 Thesis 3: Evaluating Large Language Models for Generating Programming Questions from Source Code

### 4.1 Introduction

*I developed a systematic evaluation framework to assess the QG capabilities of LLMs, using automatic evaluation metrics and complemented by human-centered evaluation metrics for the top-performer LLM. The findings provide insights into their strengths and limitations in generating programming-related assessment questions for potential educational use in the programming domain* **[P4]**.

This thesis 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 LLMs for code-related QG tasks. The thesis evaluates a diverse set of state-of-the-art LLMs. Theses 1 and 2 presented two distinct approaches for AQG from Python source code. Thesis 1 discussed an ontology-driven approach which allowed the structured representation of knowledge that would yield MCQs automatically from Python programs. Thesis 2 extended Thesis 1 by providing a hybrid approach, the ontology combined with the QuestGen AI model, to make the generation process dynamic and grab semantic understanding better. Though they both made headway, the two approaches suffer mainly in their limited scope in one aspect. No systematic evaluation metric is provided to benchmark the quality of the questions generated from source codes across the different dimensions. Hence the evaluation was very much a subjective measure that limits comparisons of results systematically with other AQG methods. Thesis 3 goes on to cover this gap by extending AQG research into a multi-language context including Java, C++, and Python. With a

broader scope, the performance of LLMs in forming questions from codes rooted in different source code paradigms with individual syntaxes, semantics, and idiomatic usages could be evaluated. A structured evaluation framework established by this thesis would assess AQG systems in terms of comprehensiveness, reliability, and reproducibility in model, language, and approach comparisons. Thus, Thesis 3 naturally follows from the methodological foundations laid in Theses 1 and 2 and directly addresses their limitation in evaluations-driven framework for AQG from source code. The primary objectives of this thesis 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 QG from program codes.
3. To empirically evaluate and rank the selected LLMs based on their performance in QG from program codes.

**4.2** Methodology

The methodology explains how the evaluation and comparison are made regarding the proficiency of various LLMs to create questions from the given source code. This section outlines all the events leading to data collection and preparation, model selection, evaluation metric selection, experiment execution, and ranking of the models. In this context, a comprehensive and impartial exercise is carried out to identify the models best suited for relevant QG tasks concerning programming code. The languages chosen for the experiment were Python, C++, and Java. These languages were focused on during the research, with the possibility of applying such methods to other structurally similar programming languages. The sequence selected aids in rendering clear views into the strengths and weaknesses of each of the models, thereby allowing a deeper understanding of questions pertaining to the future of this research. Previous studies have undertaken related efforts, like [27], [28], and [29]. Algorithm 4.1 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 based on dimensions like 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 are then summarized, including average scores, win rates, Elo ratings, repetition rates, and comparison matrices.

The generated questions were assessed for their quality to analyze differences in performance regarding the selected LLMs. Each question gets evaluated on a scale from 1 to 10 based on the evaluation metric by GPT-4-0314 as a judge. This study used objective and subjective evaluation modes, touching on the primary indicators. Relevance means how closely the generated questions match the source code. Clarity and coherence measure questions' phrasing and how logic is structured in them. Conciseness assesses whether the questions were brief by examining their length and checking for unnecessary detail or verbosity. Coverage involves how well each question covered the entire scope of the input script. It also involved whether the questions reflected different sections or key components of the code, and not just focused narrowly on isolated elements. In addition to automated scoring, human reviewers were involved to provide a pedagogical perspective on the top-performing LLM. Their insights helped validate the results and brought attention to the educational value of the questions. Human feedback added important context about classroom relevance, teaching goals, and practical usefulness, which are things that automated systems alone cannot fully capture. Evaluators kept in mind relevance and educational value when making their judgments. The approach encompassed a mix of different input data sets, multiple LLMs, stringent evaluation criteria, and automated and human judgment. The results and examples, from inputs to generated questions, are discussed in the next section. Parts of this output and the evaluation deconstruction are illustrated in

Algorithm 4.1: Multi-Model Code QG 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)
Initialize scores_by_model, reps_by_model, results as empty.
For each sample in D do:
    For each model_name in MODELS do:
        prompt ← build_generation_prompt(sample.code, sample.language)
        questions ← LLM(model_name).generate_questions(prompt)
        metrics ← evaluate_questions(questions, EVAL_MODEL)
        score ← average_scores(metrics)
        repetition ← repetition_rate(questions) // optional
        Store (model_name, sample, metrics) in results
        Append score to scores_by_model[model_name]
        Append repetition to reps_by_model[model_name]
    End For
End For
wins, comparisons ← build_win_matrix(scores_by_model)
win_rate ← win_rates(wins, comparisons)
elo ← elo_ratings(scores_by_model)
repetition ← aggregate_repetition(reps_by_model)
Construct SMPM as {ranking(scores_by_model), win_rate, elo, repetition, wins, comparisons}
```

The model average score is established by summing the scores of each criterion across all questions, and higher scores in each criterion indicate better accuracy in script-to-question generation. The rankings show that GPT-4-0314 obtained the first rank confirming its effectiveness in generating relevant, high-quality questions. Moreover, it was analytically carried out on an average win rate account of all other models to get an all-round perspective on the performance of LLMs under evaluation. The term win rate refers to a cumulative score for every model and helps determine the best-performing model among them. 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 identification of 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 following Equations (5.1) and (5.2), would calculate the New Rating and the Predicted Rating, respectively [30]. This technique is used here for the AI evaluation domain; it is derived from tournaments in sports, where it is often used.

$$\text{New Rating} = \text{Old Rating} + \text{K} \times (\text{W} - \text{P}) \tag{4.1}$$

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

$$\text{P} = \frac{1}{1 + 10^{\frac{(\text{Mo}-\text{Mp})}{\text{score point}}}} \tag{4.2}$$

Where P stands for the expected outcome for a given model, Mo for model opponent, and Mp for model player. The constants relating to 1 and 10 are customized; these traditional constants have been customized in the context to mean that the score point is 400. The two equations constitute the basis of the Elo rating methodology created initially by Arpad Elo [18] to enable fair and dynamic ranking of chess players based on match outcomes. Because of its simplicity and efficiency in tracking relative skill levels, the Elo rating system gradually found acceptance in areas other than chess, like online games, sporting events, and AI benchmarking. The second equation calculates the expected probability of one player winning against the other depending on their rating difference, and the first

updates the player's rating after every game depending on the actual and expected result. The combination of both ensures that the rating system accommodates rating adjustments to reward the unexpected win and penalize against the loss when a rating would become obsolete in view of actual performance. This means that the average win rate measure provides a clear and quantitative indication of the relative strength of the models and competitive standing in question generation.

## **5 Thesis 4:** Template-Based Question Generation from Code Using Static Code Analysis

### **5.1** Introduction

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

The methodology presented in Thesis 4 represents a significant departure from the approaches detailed in Theses 1, 2, and 3. Thesis 1 was limited to QG using engineered ontologies specific to providing support for only Python via a reasoning engine and conceptual hierarchies. Thesis 2 blended the hybrid model of ontology and NLP (QuestGen) approaches, translating the Python code into text, prior to the generation of the question. Then, in Thesis 3, custom evaluation metrics were framed for benchmarking evaluation of LLM-based systems, among them GPT-4, LLaMA, and Falcon. LLMs, introduced in Thesis 3, are highly effective for QG from source code; however, they demand substantial financial and computational resources. This thesis presents a multi-language code question generator capable of automatically producing assessment questions for Python, C++, Java, and C codes. It focuses on QG from source code 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. This template-based approach serves as a lightweight baseline for the future version alternative to the LLMs discussed in Thesis 3, offering lower computational requirements, greater interpretability, and faster processing for large-scale deployment. 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 (AQG from source code).
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 source codes.

### **5.2** Methodology

This thesis 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 [31]. While the paper [32] 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 5.1 shows the proposed methodology for a multi-language question generator from source code. The research methodology behind the multi-language question generator involves several interconnected components that work together to analyze code snippets and generate relevant questions. A detailed explanation of the methodology follows. Parsing is the process of checking the structure of the code and identifying elements like

keywords and variables. After parsing, the system extracts various structural elements from the code. After parsing, the system extracts various structural elements from the code. The QG process uses templates customized for different code elements and difficulty levels, as shown in Figure 5.2.
The templates are designed based on principles from cognitive science and educational theory, as shown in Figure 5.2. After generating candidate questions, the system applies several post-processing steps like de-duplication, shuffling, and limiting the number of questions to prevent overwhelming the user, while maintaining a balance of difficulty levels. 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 source code. The methodology involves a structured approach to assess the quality of the generated questions across several key dimensions (Bloom's Taxonomy, Difficulty Distribution, Linguistic Complexity, Code Coverage, Precision, Recall, Novelty, Educational Alignment, Cognitive Diversity, Question Quality Score).
Algorithm 5.1 shows a multi-language template-based QG 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 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 QG 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. It then 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, QG 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.
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.

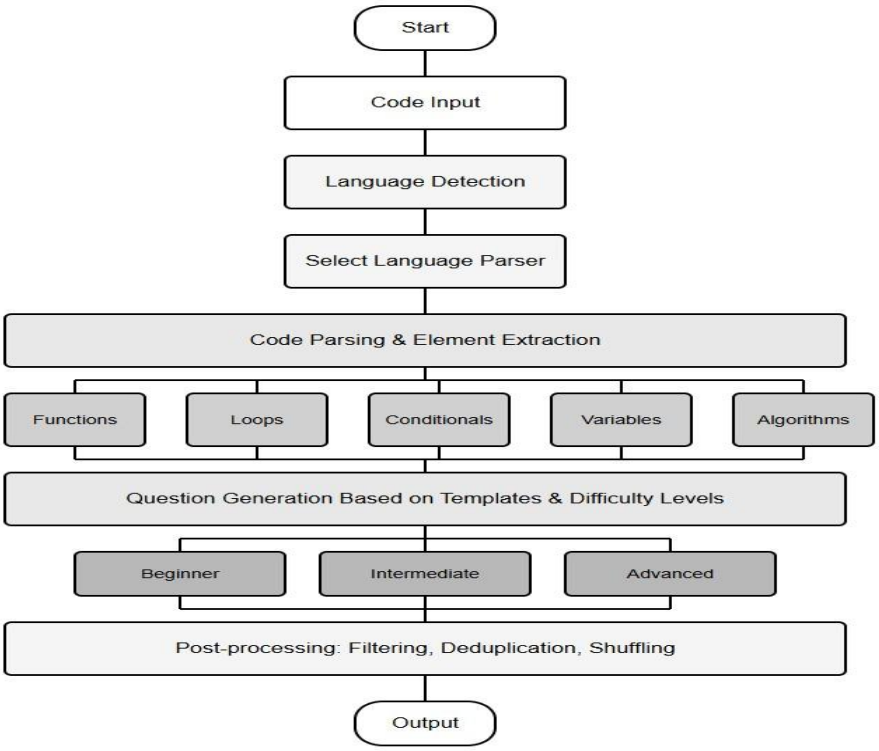


*Figure 5.1 Methodology for multi-language question generation from source code*

```
'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 5.2 Sample of templates used for question generation from source code*

Algorithm 5.1: Multi-Language Template-Based QG 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 Thesis 5: Multi-Language Static-Analysis System for Automatic Question Generation from Source Code

### 6.1 Introduction

*I developed a modular static analysis framework for AQG across multiple programming languages. The system integrates language-specific analyzers within a unified architecture designed to support consistency in QG across the four programming languages (C, C++, Java, and Python)* **[P6]**.
The graph-based pipelines in this thesis are meant to complement not compete with the approach of early LLM methods discussed in Thesis 3 and of the template-based static baseline discussed in Thesis 4. Thesis 5 has given a lightweight and reproducible baseline across languages but also revealed some pitfalls of regex parsing, including low precision, limited novelty, and a cap on structural depth. In this Thesis, that layer is replaced by language-specific parsers (Python AST, javalang, and Clang/LLVM) that are integrated through a normalization interface to ensure consistent treatment of functions, methods, loops, conditionals, and variables across Python, Java, C++, and C. Building on such normalized elements, CFG and PDG construction adds structural insights, such as control paths, branching, and complexity, alongside semantic insights such as data dependencies and variable lifecycles. The force-balanced generation mechanism then adjusts in real time from course to emphasizing under-represented Bloom levels, question types, and algorithm families to achieve more well-rounded coverage rather than chance distribution across all levels of variety in the methodology. This generates improved precision, a richer language, greater novelty, and broader cognitive diversity, while remaining interpretable, deterministic, and free per item. LLMs sometimes fail to deliver due to budgetary, privacy, or accreditation constraints. The result is an explainable and adaptable layer that can also support future hybrid pipelines, such as using curated CFG/PDG summaries to guide LLMs in producing more creative, higher-order variations. In practice, this clarifies when each method is best suited: LLMs excel in breadth and stylistic variety, while graph fusion offers transparent, coverage-controlled, and semantically grounded assessment. The research objectives of this thesis are:

1. To design and implement three automated pipelines (CFG-based, PDG-based, and CFG-PDG Synergetic) for QG from source code, 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.

### 6.2 Methodology

This thesis introduces a multi-language generator and evaluator system that takes source code as input and is capable of generating coding questions in various programming languages, including 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 [31]. 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 reasonably balanced manner. This section presents a comprehensive description of every element within the pipeline and interconnected characteristics and functions of the general system. Figure 6.1 shows the comprehensive pipeline for multi-language question generator and evaluator system. The objective of building a multi-language 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. The pipeline shown in Figure 6.1 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.
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.
4. Advanced Code Analysis: CFG identifies loops, execution paths, and branching conditionals. PDG captures variable relationships and data dependencies.
5. Force-Balanced Generation: It takes 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.
7. Output Generation: It generates structured questions.

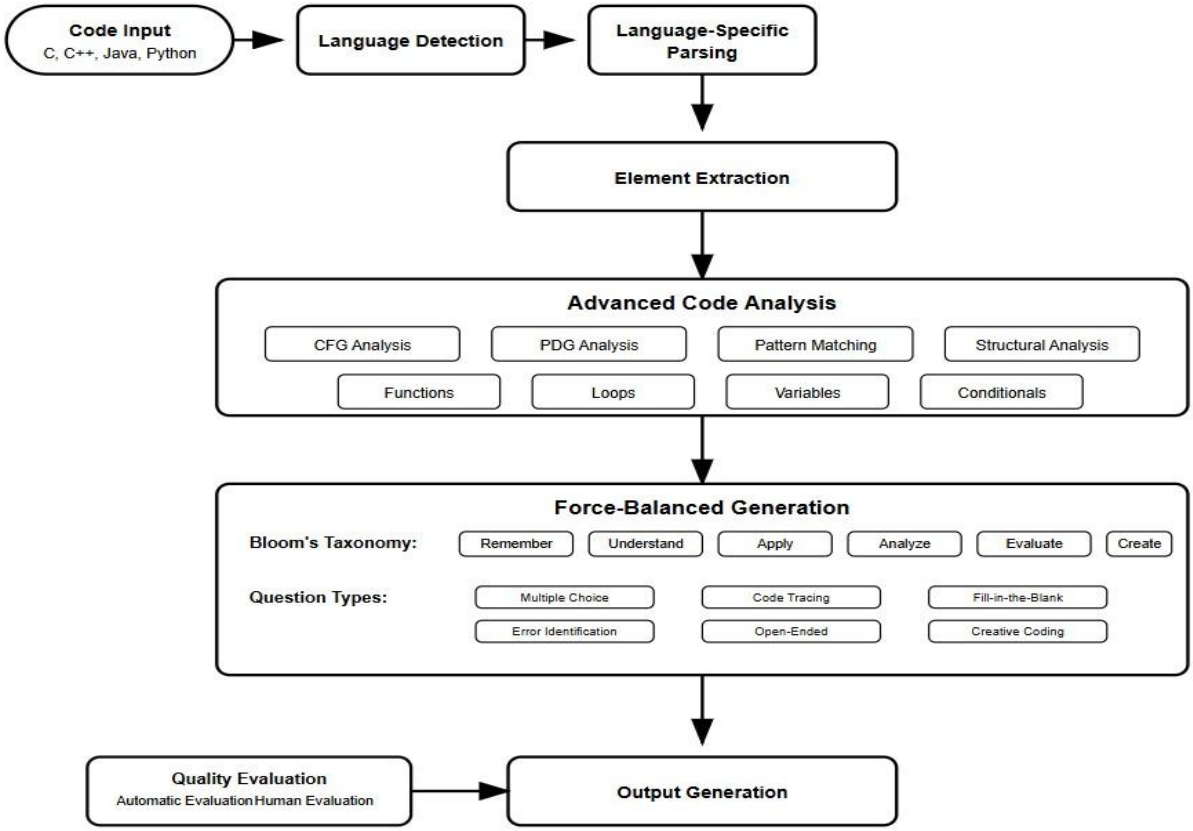


*Figure 6.1 Comprehensive pipeline for multi-language question generator and evaluator system*

Algorithm 6.1 shows the CFG pipeline algorithm for code QG 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 questions like tracing, MCQ, and basic error-identification questions based on flow paths. Algorithm 6.2 shows the PDG pipeline algorithm for code QG 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 questions like dependency, comprehension, and advanced error-identification questions.
Algorithm 6.3 shows the CFG-PDG pipeline algorithm for code QG and evaluation. Its main objective is to generate advanced, diverse questions using a synergistic integration of CFG and PDG.

Algorithm 6.1: 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 (e.g., tracing, MCQ, and error-identification questions).

8: Evaluate QS using quality and diversity metrics.

Algorithm 6.2: 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 (e.g., dependency, error identification, and comprehension questions).

8: Evaluate QS using quality and diversity metrics.

Algorithm 6.3: 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 (e.g., tracing, dependency, error identification, creative coding, and MCQs).

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

It parses and simultaneously extracts CFG and PDG representations. Next, it integrates structural (CFG) and semantic (PDG) information. Then, it identifies algorithm types. Finally, it generates a reasonably balanced set of questions, including creative coding and higher-order Bloom questions.

The same automatic evaluation metrics as the baseline model (Thesis 4 Automatic 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]. 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 (relevance, difficulty appropriateness, clarity, educational value, and cognitive level match).

## 7 Conclusion

### **7.1** 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 dissertation.

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 Generating Programming Questions from Code: 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 LLMs 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 ML-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 Question Generation from Source Code: 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. 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. Finally, a promising extension of

this work lies in integrating LLMs with the CFG-PDG framework. The modular design of the current system already provides clear entry points for such hybridization, where LLMs can be guided by structural program representations rather than generating questions in isolation. By using CFG and PDG graphs as guardrails, LLMs could enrich QG with greater semantic variety and higher-order reasoning while maintaining alignment with Bloom's taxonomy and algorithmic correctness.

## **7.2** 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 Generating Programming Questions from Code," Pollack Periodica: An International Journal for Engineering and Information Sciences, Status: Accepted/Minor Revision, doi: 10.1556/606.2025.01471. Quartile: **Q3**.

**[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, Status: Under Review. Quartile: **Q3**.

**[P6]** J. Alshboul and E. Baksa-Varga, "Multi-Language Static-Analysis System for Automatic Question Generation from Source Code," Status: To Be Submitted.

### 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.

## 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] M. Zerkouk, M. Mihoubi, and B. Chikhaoui, "A Comprehensive Review of AI-based Intelligent Tutoring Systems: Applications and Challenges," Jul. 25, 2025, *arXiv*. doi: 10.48550/arXiv.2507.18882.

[3] M. Vinueza-Morales, J. Rodas-Silva, C. Vidal-Silva, J. Córdova-Morán, and E. Cevallos-Ayón, "Teaching programming in higher education: a bibliometric analysis of trends, technologies, and pedagogical approaches," *Frontiers in Education*, vol. 10, Mar. 2025, doi: 10.3389/feduc.2025.1525917.

[4] S. Al Faraby, A. Adiwijaya, and A. Romadhony, "Review on Neural Question Generation for Education Purposes," *International Journal of Artificial Intelligence in Education*, vol. 34, no. 3, pp. 1008–1045, Sep. 2024, doi: 10.1007/s40593-023-00374-x.

[5] 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.

[6] R. Queirós, J. C. Paiva, and J. P. Leal, "Programming Exercises Interoperability: The Case of a Non-Picky Consumer," in *10th Symposium on Languages, Applications and Technologies (SLATE 2021)*, R. Queirós, M. Pinto, A. Simões, F. Portela, and M. J. Pereira, Eds., in Open Access Series in Informatics (OASIcs), vol. 94. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, p. 5:1-5:9. doi: 10.4230/OASIcs.SLATE.2021.5.

[7] I. Mekterović, L. Brkić, and M. Horvat, "Scaling Automated Programming Assessment Systems," *Electronics*, vol. 12, no. 4, 2023, doi: 10.3390/electronics12040942.

[8] H. S. Wankhede and A. W. Kiwelekar, "Qualitative Assessment of Software Engineering Examination Questions with Bloom's Taxonomy," *Indian Journal of Science and Technology*, vol. 9, no. 6, Mar. 2016, doi: 10.17485/ijst/2016/v9i6/85012.

[9] L. J. Tamang, R. Banjade, J. Chapagain, and V. Rus, "Automatic Question Generation for Scaffolding Self-explanations for Code Comprehension," in *Artificial Intelligence in Education*, M. M. Rodrigo, N. Matsuda, A. I. Cristea, and V. Dimitrova, Eds., Cham: Springer International Publishing, 2022, pp. 743–748.

[10] O. Sitthisak, L. Gilbert, and D. Albert, “Ontology-Driven Automatic Generation of Questions from Competency Models,” in *The 9th International Conference on Computing and InformationTechnology (IC2IT2013)*, P. Meesad, H. Unger, and S. Boonkrong, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 145–154.

[11] S. Alkhuzaey, F. Grasso, T. R. Payne, and V. Tamma, “Evaluating the Fitness of Ontologies for the Task of Question Generation,” Apr. 08, 2025, *arXiv*. doi: 10.48550/arXiv.2504.07994.

[12] O. Sychev and D. Shashkov, “Mass Generation of Programming Learning Problems from Public Code Repositories,” *Big Data and Cognitive Computing*, vol. 9, no. 3, 2025, doi: 10.3390/bdcc9030057.

[13] E. Logacheva, A. Hellas, J. Prather, S. Sarsa, and J. Leinonen, “Evaluating Contextually Personalized Programming Exercises Created with Generative AI,” in *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1*, in ICER ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 95–113. doi: 10.1145/3632620.3671103.

[14] K. Zhu, Y. Lu, H. Huang, L. Yu, and J. Zhao, “Constructing More Complete Control Flow Graphs Utilizing Directed Gray-Box Fuzzing,” *Applied Sciences*, vol. 11, no. 3, 2021, doi: 10.3390/app11031351.

[15] Y. Yan, N. Cooper, K. Moran, G. Bavota, D. Poshyvanyk, and S. Rich, “Enhancing Code Understanding for Impact Analysis by Combining Transformers and Program Dependence Graphs,” *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024, doi: 10.1145/3643770.

[16] S. K. Patil and M. M. Shreyas, “A Comparative Study of Question Bank Classification based on Revised Bloom’s Taxonomy using SVM and K-NN,” in *2017 2nd International Conference On Emerging Computation and Information Technologies (ICECIT)*, 2017, pp. 1–7. doi: 10.1109/ICECIT.2017.8453305.

[17] Z. Ullah, A. Lajis, M. Jamjoom, A. Altalhi, and F. Saleem, “Bloom’s taxonomy: A beneficial tool for learning and assessing students’ competency levels in computer programming using empirical analysis,” *Computer Applications in Engineering Education*, vol. 28, no. 6, pp. 1628–1640, 2020, doi: https://doi.org/10.1002/cae.22339.

[18] 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.

[19] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, “A Comprehensive Overview of Large Language Models,” *ACM Trans. Intell. Syst. Technol.*, vol. 16, no. 5, Aug. 2025, doi: 10.1145/3744746.

[20] A. Ushio, F. Alva-Manchego, and J. Camacho-Collados, “A Practical Toolkit for Multilingual Question and Answer Generation,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, D. Bollegala, R. Huang, and A. Ritter, Eds., Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 86–94. doi: 10.18653/v1/2023.acl-demo.8.

[21] C. Cheng, Z. Huang, G. Zhao, Y. Guo, X. Lin, J. Wu, X. Li, and S. Wang, “From Objectives to Questions: A Planning-based Framework for Educational Mathematical Question Generation,” in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds., Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 12836–12856. doi: 10.18653/v1/2025.acl-long.628.

[22] B. Nguyen, M. Yu, Y. Huang, and M. Jiang, “Reference-based Metrics Disprove Themselves in Question Generation,” 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. 13651–13666. doi: 10.18653/v1/2024.findings-emnlp.798.

[23] C. Zhou, M. Wang, T. Zhang, Q. Zhu, J. Li, and H. Huang, “From Answers to Questions: EQGBench for Evaluating LLMs’ Educational Question Generation,” Aug. 05, 2025, *arXiv*. doi: 10.48550/arXiv.2508.10005.

[24] D. Gnanasekaran, R. Kothandaraman, and K. Kaliyan, “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, 2021, doi: 10.2174/2213275912666191113143335.

[25] E. Kasneci *et al.*, “ChatGPT for good? On opportunities and challenges of large language models for education,” *Learning and Individual Differences*, vol. 103, p. 102274, 2023, doi: https://doi.org/10.1016/j.lindif.2023.102274.

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

[27] 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.

[28] 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.

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

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

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

[32] 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.