Research Project

Marc Bessa Hoffmann

# Automatically Generating Programming Exercises with Open-Source LLMs: Integrating Lecture Slides and Learning Objectives

May 6, 2025

supervised by:
Prof. Dr. Sibylle Schupp
Daniel Rashedi

Hamburg University of Technology (TUHH)
*Technische Universität Hamburg*
Institute for Software Systems
21073 Hamburg

# Statutory Declaration

I herewith declare that I have composed the present thesis myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The thesis in the same or similar form has not been submitted to any examination body and has not been published. This thesis was not yet, even in part, used in another examination or as a course performance.

Hamburg, May 6, 2025

Marc Bessa Hoffmann

Text source: Goethe University, Department of Economics

## Abstract

Automatic question generation using Large Language Models (LLMs) is an emerging area in educational technology that aims to reduce manual effort while improving learning outcomes. However, generating questions that align with clearly defined learning objectives (LOs) while making use of contextual information from lecture slides remains a nontrivial challenge. In the domain of programming education, existing approaches rely either on external contextual input or focus on learning objectives, but do not combine both. Moreover, current solutions are still largely limited to the generation of multiple-choice questions (MCQ). This work presents a new system for generating programming exercises aimed at university-level students. The system combines LOs classified according to Bloom's taxonomy with key concepts extracted from lecture slides. Leveraging open-source LLMs, it produces diverse open-ended tasks, including implementation, explanation, and comparison exercises. We evaluated 60 exercises generated from predefined LOs and real-world lecture slides from a Haskell programming course. Each exercise was analyzed using an automated evaluation pipeline assessing grammar, readability, code validity, and semantic similarity to lecture content. The results show that all exercises were grammatically correct and readable at the university level. For exercises containing Haskell code, 70% of the snippets were compiled without errors. In addition, we manually reviewed selected exercises and confirmed that most align with the defined learning objectives and the lecture slides.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Code listing

# 1 Introduction

Large language models (LLMs) have become a transformative force across a wide range of domains, including education [1]. In educational settings, LLMs are increasingly being used to help teachers generate instructional materials or questions and exercises. This work focuses specifically on the field of automatic question generation, where recent research has demonstrated promising results in tasks such as reading comprehension [2], computer programming [3], and solving mathematical problems [4]. In particular, we focus on the generation of programming exercises intended for use in university-level education.

One of the core challenges in generating exercises is to ensure that they are meaningfully aligned with the intended learning objectives of a course. Learning objectives define the specific skills, knowledge, or competencies that students are expected to achieve by the end of an exercise. Ideally, these objectives are structured according to established educational frameworks. A popular framework is Bloom's taxonomy, which categorizes cognitive learning goals into six levels ranging from simple recall to higher-order skills [5]. At the same time, generated exercises should reflect the actual content of the lecture material taught in a course. However, generating exercises that align with Bloom-based learning objectives while considering lecture content remains a nontrivial task. This is particularly challenging when using general-purpose large language models. These models are not explicitly optimized for educational purposes or for the interpretation of structured teaching materials.

In the domain of programming education, existing solutions typically address only parts of the challenge. Some methods focus on generating questions based on educational materials [6]. Others attempt to guide the generation process using Bloom's taxonomy, categorizing the output into cognitive levels [3, 7]. To our knowledge, no existing approach combines both contextual input from lecture slides and formalized learning objectives. In addition, most systems are limited to generating multiple-choice questions (MCQ). Especially in the context of programming, open-ended tasks such as implementation, explanation, or comparison often represent more meaningful and authentic exercises. Furthermore, we observed that most existing approaches rely on commercial models like GPT-3 or GPT-4, which limits transparency and reproducibility.

To address these limitations, we present an exercise generation framework for computer science education that leverages open-source LLMs. Our system combines two sources of guidance: (1) learning objectives categorized by Bloom's taxonomy and (2) lecture slides that contain the actual teaching material of a course. The framework takes as input a learning objective, a topic, and a set of lecture slides. The learning objective is automatically classified according to Bloom's taxonomy. Based on the assigned Bloom level, the system selects predefined example assignments that reflect the corresponding cognitive demands. These examples serve as structural guidance for the exercise generation process. In parallel, a vector-based similarity search is performed between the learning objective and the lecture slides to retrieve the most relevant content. This mechanism follows the Retrieval-Augmented Generation (RAG) approach, where the external context grounds the model output [8]. The combined input enables the generation of programming exercises that are both grounded in course content and aligned with the intended learning outcomes. Depending on the cognitive Bloom level, the framework can generate a variety of task types including syntax recall, implementation tasks, conceptual explanation, and comparative evaluation.

To assess the quality of the generated exercises, we applied a fully automated five-stage evaluation pipeline. This pipeline verified (1) grammatical correctness using rule-based grammar analysis, (2) readability based on Flesch Reading Ease, (3) compilability of embedded Haskell code using the Glasgow Haskell Compiler (GHC), (4) semantic similarity between exercises and the related lecture slides using sentence embeddings, and (5) alignment with the intended Bloom level through LLM-based classification. The results show that all exercises were grammatically correct, apart from minor false positives caused by embedded code elements. The readability of the texts matched expectations for university-level material. For exercises containing code, 70% of the Haskell snippets were successfully compiled. In addition, we manually reviewed selected examples and confirmed that most exercises align well with both the intended learning objectives and the course content.

This paper is structured as follows: First, we provide essential background information by introducing fundamental concepts related to LLMs, embeddings, prompt engineering, and Bloom's taxonomy. After that, we review related work in the field. Building upon this foundation, we then describe our own approach for generating programming exercises. Finally, we present and discuss the evaluation results.

# 2 Background

In this chapter, we provide the necessary background for this work. We begin with an introduction to neural networks. After that, we continue with large language models (LLMs), which are the core technology used to generate the programming exercises in this work. Finally, we present Bloom's taxonomy, a framework used to categorize learning objectives into distinct cognitive levels.

## 2.1 Neural Networks

Neural networks are a concept inspired by the workings of the human brain. In biological systems, neurons are activated when they reach a certain activation threshold. Similarly, a mathematical neuron is an abstraction that activates when a numerical threshold is reached. The mathematical idea behind neurons has existed for quite some time and was first introduced in 1943 by McCulloch and Pitts [9]. They proposed a formal model of a neuron that computes an affine linear combination of its inputs and the corresponding weights to produce an output $z$. Let us assume a neuron with $n$ inputs, where the input vector is defined as:

$$\mathbf{x} = (x_1, x_2, \ldots, x_n)^\top$$

with corresponding weights for each input:

$$\mathbf{w} = (w_1, w_2, \ldots, w_n)^\top.$$

The neuron then computes the weighted sum of its inputs:

$$z = \mathbf{w}^\top \mathbf{x} + b = \sum_{i=1}^{n} w_i x_i + b,$$

where $b$ is the bias term that allows the model to shift the activation threshold. A higher bias value $b$ increases the likelihood that the neuron activates, even when the weighted sum of the inputs is small.

However, this is not the final output of the neuron, as each neuron applies an activation function to the result. The selection of the activation function depends on the task and has a direct impact on the output of the model. If the output needs to represent a probability for a given category, the sigmoid function can be used to produce values between 0 and 1. For example, an output of 0.80 indicates an 80% confidence that an image contains a cat. In other cases, a step function is appropriate for binary classification. Determining whether an image contains a cat or not would be a suitable case here.

A neural network consists of multiple individual neurons organized in layers. Each neural network has an input layer and an output layer. The layers in between are referred to as hidden layers. An example of a neural network is shown in Figure 2.1.

The input layer consists of two neurons (highlighted in red), followed by two hidden layers with 3 and 2 neurons (purple). The final output layer has one neuron (green).

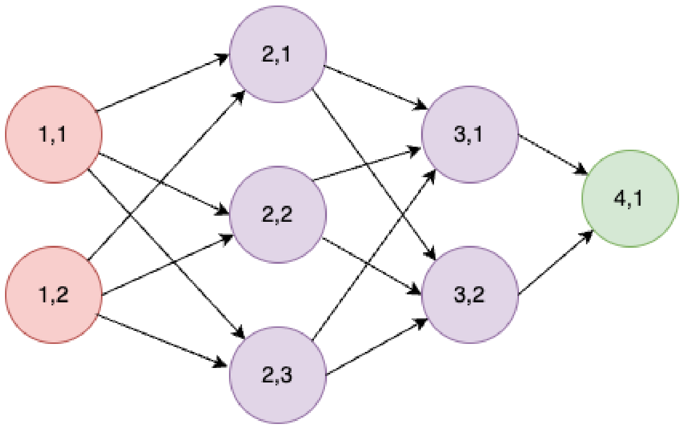


Figure 2.1: An Example of a Neural Network.

In a neural network, each neuron computes a weighted sum of its inputs followed by an activation function. This operation allows a single neuron to represent a simple linear separation in the input space. When multiple neurons and layers are combined, the network can learn to represent more complex, non-linear decision regions. The shape and complexity of these regions depend on the structure of the network and the task to which it is trained to solve.

Building on the concept of neural networks, the following section introduces large language models and their role in this work.

## 2.2 Large Language Models

This section introduces large language models (LLMs), followed by an overview of embeddings and prompt engineering.

Large Language Models (LLMs) have recently been developed in the fields of artificial intelligence (AI) and natural language processing (NLP). A key breakthrough was the introduction of the Transformer architecture in the paper *"Attention is All You Need"* by Vaswani et al. [10]. Earlier models like Recurrent Neural Networks (RNNs) or Long Short-Term Memory Networks (LSTMs) struggled with long sequences as they process inputs sequentially and lose earlier information over time. Transformers overcome this problem by introducing self-attention. This mechanism allows each word in a sequence to attend to itself and all other words, regardless of their position. As a result, even words that appear later in the sequence can retain strong contextual connections to earlier ones. This advancement laid the foundation for models such as BERT [11] and GPT-3 [12]. These models are capable not only of understanding natural language but also of generating coherent and contextually appropriate text. Applications like ChatGPT have demonstrated strong performance in dialogue systems. However, the use of LLMs extends far beyond chatbots. They are also applied in areas such as machine translation, text summarization, question answering, content generation, and code completion [13].

LLMs are typically trained on large datasets that include text from books, websites, programming code, and other human-generated content [14]. During training, they learn statistical patterns and semantic relationships between words and concepts. The nature of the training data plays a crucial role in shaping the behavior of the model, influencing its vocabulary, writing style, and domain knowledge. Since this work focuses on generating programming exercises, it is beneficial to use models that have been trained on code. Such models are more likely to produce syntactically correct output and to generate tasks that reflect realistic programming logic.

### Decoding Parameters

Although the behavior of the model is learned during training, its output can still be influenced at inference time through decoding parameters. Before looking at specific parameters, it is important to note that LLMs do not operate on complete words. Instead, they process smaller units called tokens, which may represent entire words or subwords.

The decoding parameters influence how the model selects the next token and impact the level of randomness, diversity, and repetition in the generated output [15, 16]. The following decoding parameters are commonly used to control text generation:

- **Temperature:** Controls the randomness of the predictions. Lower values lead to more deterministic and focused output, whereas higher values introduce more diversity and creativity. In the context of exercise generation, a higher temperature may result in more creative exercises.

- **Top-p:** Restricts the sample pool to the smallest set of tokens whose cumulative probability exceeds a defined threshold $p$. For example, with top_p = 0.9, only the most likely tokens of 90% are considered. This influences whether the generated exercises follow standard phrasing or introduce more varied formulations.

- **Frequency penalty:** Penalizes the repeated use of tokens that occur frequently. A higher penalty reduces redundancy and helps avoid generating exercises with repetitive structures or wording.

- **Context length (`num_ctx`):** Defines the maximum number of tokens the model can process in a single pass, including both input and output. A higher context window may be necessary when the prompt contains large amount of information like lecture slides.

Adjusting these parameters allows us to control the style, variability, and length of the generated exercises. In addition to that, the generation process can also be influenced by the external context provided to the model, which we discuss in the next section.

### 2.2.1 Embeddings

Embeddings play a central role in large-language models (LLMs), as they provide the mathematical foundation for how these models understand and process text. LLMs convert words, sentences, or even entire documents into high-dimensional vectors known as embeddings. These numerical representations capture the semantic meaning of the text and enable the model to perform reasoning using purely mathematical operations.

Let us consider the following example with three sentences:

- Sentence A: *"Loops are used to repeat instructions."*
- Sentence B: *"A function returns a value."*
- Learning Objective: *"Understand how loops work in programming."*

To obtain vector representations of text, embedding models are used. These models transform input sequences into dense numerical vectors that reflect semantic similarity. Many embedding models are based on transformer architectures and are closely related to large language models (LLM). Well-known examples include BERT [11] or Sentence-BERT [17]. Depending on the specific architecture, embeddings are typically represented in a 384, 768, or 1024 dimensional space. The resulting vectors for the example above might look as follows:

$$\text{Embedding}_{\text{A}} = \begin{bmatrix} 0.21 \\ 0.53 \\ -0.13 \\ \vdots \\ 0.09 \end{bmatrix}, \quad \text{Embedding}_{\text{B}} = \begin{bmatrix} 0.11 \\ 0.34 \\ 0.18 \\ \vdots \\ 0.02 \end{bmatrix}, \quad \text{Embedding}_{\text{LO}} = \begin{bmatrix} 0.24 \\ 0.51 \\ -0.15 \\ \vdots \\ 0.11 \end{bmatrix} \tag{2.1}$$

Now that we have their mathematical representation, we can measure the similarity between the vectors using mathematical formulas. One commonly used method is cosine similarity [18].

$$\cos(\theta) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \cdot \|\mathbf{v}\|} \tag{2.2}$$

Where:

- $\mathbf{u} \cdot \mathbf{v}$ is the dot product of the vectors
- $\|\mathbf{u}\|$ and $\|\mathbf{v}\|$ are their Euclidean norms

The cosine similarity ranges from −1 to 1. A value close to 1 indicates that the vectors point in the same direction and are therefore highly similar in meaning. A value of 0 means that the vectors are orthogonal and have no semantic similarity. Values near −1 indicate that the vectors point in opposite directions, representing dissimilarity or even a negative correlation in meaning. A possible outcome for the example above could be the following:

$$\cos(\mathrm{LO}, \mathrm{A}) = 0.92, \quad \cos(\mathrm{LO}, \mathrm{B}) = 0.34$$

This indicates that Sentence A is much more semantically related to the learning objective than Sentence B. If we project the embeddings into a 2D space, we would see that the vector of the learning objective lies closer to Document A than to Document B. This proximity supports the numerical result and provides intuitive insight into semantic similarity.

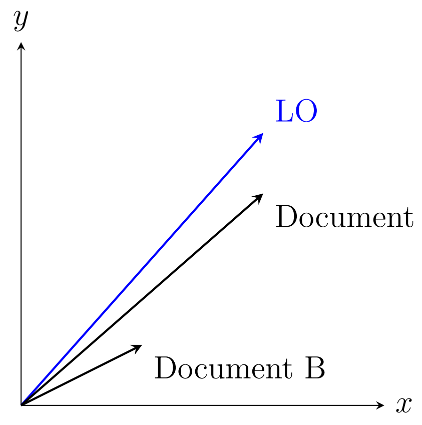


Figure 2.2: 2D Projection of Embedded Vectors.

This principle will later be applied in our system, where both learning objectives and lecture slides are embedded and compared using cosine similarity to retrieve the most relevant slides.

### 2.2.2 Prompt Engineering

Our work is based on guiding the behavior of LLMs through specific instructions, and therefore it is crucial to understand the role of prompts. This subsection gives an introduction to prompts and an overview of prompt engineering techniques.

In recent years, increasingly advanced techniques have been developed in the field of prompt engineering to guide and control the responses of LLM more effectively [19]. A prompt is the input text given to a large language model (LLM) to generate a response. It provides the initial context and instructions for the model's behavior. According to OpenAI, prompts can typically be categorized into two types based on their role within an application [20]:

- **User Prompt:** The input provided directly by the end user. For example, in a chatbot application, a user prompt could be *“Generate me a exercise about Haskell programming.”*
- **System Prompt:** A predefined instruction embedded in the application to consistently influence the behaviour of the model. The system prompts are not visible to end users and are configured by developers to control tone, format, or style. An example could be *“You are a helpful and concise assistant who answers in bullet points.”*

The core idea of prompt engineering is to design prompts that guide the model toward producing the intended output. In recent years, several prompt engineering techniques have been researched:

When giving instructions, it is important to note that LLMs tend to respond in a general way if the prompt is vague. For example, the previously mentioned user prompt *“Generate me a exercise about Haskell programming”* is quite unspecific and may result in a broad or unfocused output. Therefore, it is essential to formulate prompts clearly and precisely. A more effective version could be: *“Generate a university-level exercise in Haskell focusing on conditionals”*. Another approach is to instruct the model to adopt a specific role, a technique known as *role-prompting*. In the context of exercise generation, we might ask the model to act as an assistant specialized in generating programming exercises. Such role prompts are often part of the system prompt and are defined by developers to ensure consistent and domain-specific behavior.

Furthermore, it can be beneficial to provide one or more examples within the prompt to illustrate the expected output. This helps the model better understand the desired structure and content of the response.

- **Zero-shot prompting:** Prompting the model without any examples; only a task description is provided.
- **One-shot prompting:** The prompt includes a single example to illustrate the desired output.
- **Few-shot prompting:** The prompt includes several examples to help the model generalize patterns and generate consistent results.

If we again consider the domain of exercise generation, a prompt could include one or more examples of how the output should be structured. This allows the model to orient itself based on these examples and generate exercises that follow a similar style and level of complexity.

More advanced techniques have been proposed to improve the reasoning capabilities and consistency of LLMs in complex tasks. Among these, *Chain-of-Thought prompting* [21] stands out for its ability to guide the model through intermediate reasoning steps by adding simple cues like *“Let’s think step by step.”* Other techniques, such as *self-consistency*, further enhance the reliability of answers by aggregating multiple reasoning paths [22].

In this work, we apply established prompting principles and techniques to construct prompts that enable the generation of meaningful and structured programming exercises.

## 2.3 Bloom's Taxonomy

When generating educational exercises, it is essential to consider that different types of exercises serve different purposes. Some exercises focus on recalling facts, while others require critical thinking or problem solving. To address these differences, a structured framework is necessary. Bloom's Taxonomy is a classification system for learning objectives developed by Benjamin Bloom et al. in 1956. The taxonomy was revised in 2001, updating some of the original terms and refining the categorization of learning skills [23]. It defines six different levels that categorize the cognitive processes involved in learning. The six learning levels consist of the following.

- **Remember** – Retrieving, recognizing, and recalling relevant knowledge from long-term memory.
- **Understand** – Constructing meaning from oral, written, and graphic messages through interpreting, exemplifying, classifying, summarizing, inferring, comparing, and explaining.
- **Apply** – Carrying out or using a procedure for executing or implementing.
- **Analyze** – Breaking material into constituent parts and determining how the parts relate to one another and to an overall structure or purpose through differentiating, organizing, and attributing.
- **Evaluate** – Making judgments based on criteria and standards through checking and critiquing.
- **Create** – Putting elements together to form a coherent or functional whole; reorganizing elements into a new pattern or structure through generating, planning, or producing.

Table 2.1 provides an overview of the six Bloom taxonomy levels along with representative keywords and example learning objectives associated with each level.

In this project, Bloom's taxonomy is used to classify learning objectives into one of the six cognitive levels. This classification defines the intended complexity that the generated exercise should address.

| **Bloom's Level** | **Key Verbs (keywords)** | **Example Learning Objective** |
|---|---|---|
| Create | design, formulate, build, invent, create, compose, generate, derive, modify, develop. | *By the end of this lesson, the student will be able to design an original homework problem dealing with the principle of conservation of energy.* |
| Evaluate | choose, support, relate, determine, defend, judge, grade, compare, contrast, argue, justify, support, convince, select, evaluate. | *By the end of this lesson, the student will be able to determine whether using conservation of energy or conservation of momentum would be more appropriate for solving a dynamics problem.* |
| Analyze | classify, break down, categorize, analyze, diagram, illustrate, criticize, simplify, associate. | *By the end of this lesson, the student will be able to differentiate between potential and kinetic energy.* |
| Apply | calculate, predict, apply, solve, illustrate, use, demonstrate, determine, model, perform, present. | *By the end of this lesson, the student will be able to calculate the kinetic energy of a projectile.* |
| Understand | describe, explain, paraphrase, restate, give original examples of, summarize, contrast, interpret, discuss. | *By the end of this lesson, the student will be able to describe Newton's three laws of motion in her/his own words.* |
| Remember | list, recite, outline, define, name, match, quote, recall, identify, label, recognize. | *By the end of this lesson, the student will be able to recite Newton's three laws of motion.* |

Table 2.1: Bloom's Taxonomy Levels with Keywords and Example Learning Objectives [24].

# 3 Related Work

The early approaches to automatic question generation were predominantly rule-based or related to syntactic and semantic transformation techniques [25]. In recent years, research has increasingly shifted toward leveraging large language models (LLMs). This chapter presents related work that connects to our research work and outlines how our approach differs from existing methods.

A major aspect in generating educational exercises is the level of cognitive capacity that the exercise aims at. In the work by Scaria et al. [7], Bloom's taxonomy was used to generate general educational questions using both open-source and GPT-based models. Another approach employs a fine-tuned BERT model to classify learning objectives according to Bloom levels [3]. In our work, we also incorporate Bloom's taxonomy to address the cognitive dimension of generated questions. However, instead of using machine learning-based classifiers, we apply a keyword-matching approach to associate learning objectives with appropriate Bloom levels.

Another important aspect to consider in question generation is the use of additional context to improve the quality and relevance of generated questions. Retrieval-Augmented Generation (RAG) is an approach that was first introduced by Lewis et al. [8]. The core idea is to retrieve an external context to ground the model's output and reduce the risk of hallucination or generative responses. In the educational domain, Maity et al. [26] successfully applied RAG to generate questions from course materials. Their approach shows that contextually grounded question generation can lead to more accurate and pedagogically relevant results. In our work, we incorporate RAG into our pipeline by using lecture slides as a contextual basis for exercise generation.

Although context and cognitive level are essential factors, the format of the question itself also has a significant impact on how learners interact with and reflect on the material. Despite growing interest in the generation of LLM-based questions, most of existing research in the educational domain has focused primarily on multiple choice questions (MCQs) [27, 3]. In contrast, our approach is not limited to MCQs. We aim to support the generation of questions across various formats, including open-ended and coding-related exercises. To guide the model in generating such diverse question types, we provide example tasks that resemble real university-level exercises.

Among the works that we identified, only two approaches generate questions in the context of programming education. The work of Doughty et al. [3] compares GPT-4-generated multiple-choice questions with those authored by humans in the domain of programming education. EvalQuiz [6] explores the automatic generation of self-assessment quizzes from lecture slides using GPT-4.

Our approach extends previous work by the combination of:

- We go beyond MCQs and aim to generate diverse open-ended programming exercises.
- We incorporate a structured educational context by including lecture slides.

- We integrate learning objectives and apply Bloom’s taxonomy to classify the cognitive level.
- We rely on open-source models to ensure transparency.

To the best of our knowledge, no existing work combines these components into a unified pipeline to generate programming exercises.

# 4 Approach

This chapter outlines our approach to generate programming exercises that are aligned with two key elements: (1) learning objectives categorized by Bloom's taxonomy and (2) the specific content of the lecture slides. The goal of this chapter is to present a structured pipeline that transforms pedagogical input into targeted programming exercises using Large Language Models (LLMs). An overview of the entire pipeline is shown in Figure 4.1.

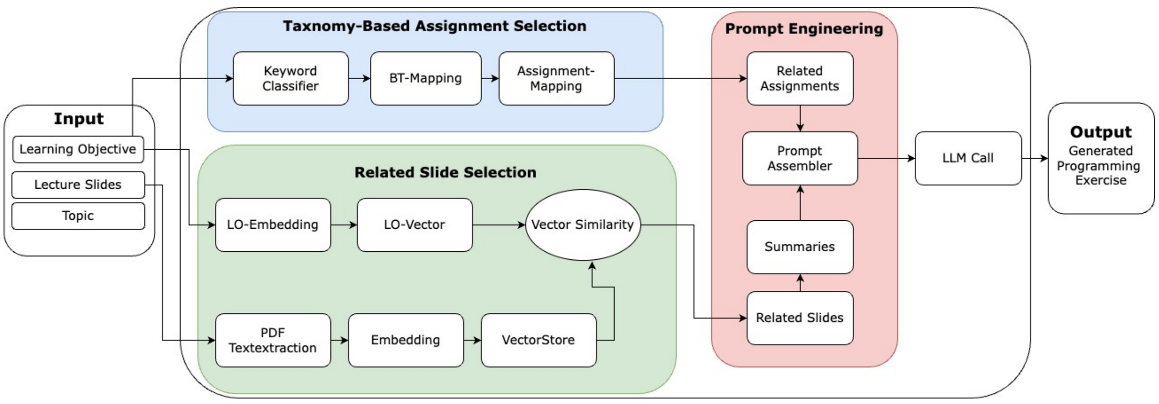


Figure 4.1: Exercise Generation Pipeline.

A structured input serves as the starting point for the generation process. This input consists of three parameters provided by the user:

1. A title or topic: Defines the thematic scope and focus of the exercise.
2. A learning objective: Describes the educational outcome that students should achieve by solving the exercise.
3. Lecture slides: Provide the contextual foundation to ensure that the generated exercise follows the notation, terminology, and concepts used in the course.

In addition to a learning objective and lecture slides, the inclusion of a user-defined topic is motivated by the need to further narrow the focus of the generation process. This topic can specify a subtopic (e.g., conditionals) or a programming language (e.g., Haskell).

The user inputs are processed through three main stages of the pipeline:

- **Taxonomy-Based Assignment Selection:** The learning objective is classified into one of Bloom's cognitive levels using keyword matching. For each level, predefined example assignments illustrate how programming exercises should look at the corresponding level of complexity.

- **Related Slide Selection:** Based on the learning objective and the lecture slides, this stage retrieves the most relevant slides using a retrieval-augmented approach. The selected slides are then summarized using an LLM to extract key concepts and code examples.
- **Prompt Engineering:** In the final stage, both the example assignments and the summarized slides are combined into a structured prompt that serves as input for generating the final programming exercise.

The following sections provide a detailed explanation of each pipeline component and its role in the exercise generation process.

## 4.1 Taxonomy-Based Assignment Selection

The taxonomy-based assignment selection process consists of two stages. In the first stage, the learning objective is classified according to Bloom's taxonomy using keyword matching. In the second stage, the resulting Bloom level is mapped to the corresponding assignment sheets. Each assignment sheet defines how a coding exercise should be structured for the designated learning level. In the following subsections, we describe both stages in detail.

### 4.1.1 Classification of Learning Objectives

In order to generate an exercise that targets an appropriate cognitive level, it is necessary to first determine the intended complexity of the learning objective. This subsection describes how a given learning objective can be mapped to a Bloom level using keyword-based classification.

The core idea behind the learning objective classification process is to analyze the words present in a given learning objective. If one of these words matches a keyword from Bloom's taxonomy (see Table 2.1), the objective is assigned to the corresponding Bloom learning level. To demonstrate the classification process, we use the following learning objective as an example:

*"Students should have the ability to describe QuickSort."*

The first step in processing the learning objective is to split the sentence into tokens. A token is a single unit of text, typically a word or punctuation mark. Tokenization is the process of segmenting a sentence into these individual units. Applied to the example, tokenization produces the following sequence of tokens:

```
['Students', 'should', 'have', 'the', 'ability', 'to',
'describe', 'QuickSort', '.']
```

Next, each token is enriched with grammatical context through part-of-speech (POS) tagging, which assigns a syntactic category to each word (e.g., `VB` for base-form verbs, `NNS` for plural nouns). For the tokens above, the POS tagging yields the following:

```
[('Students', 'NNS'), ('should', 'MD'), ('have', 'VB'),
('the', 'DT'), ('ability', 'NN'), ('to', 'TO'),
('describe', 'VB'), ('QuickSort', 'NNP'), ('.', '.')]
```

To match keywords from the Bloom taxonomy table, we focus specifically on verbs. In this example, the verbs *have* and *describe* are identified, as their tokens are tagged as `VB`. The verb *describe* already appears in its base form. However, a verb like *describing* would first be stemmed to *describe* before the matching process. From Table 2.1, we observe that *describe* is listed under the `Understand` category and can therefore be used to classify the learning objective accordingly.

Several libraries can be used to perform the natural language processing steps described above. One widely used tool is the Natural Language Toolkit (NLTK), an open-source Python library designed for the analysis and processing of textual data [28].

A possible algorithm for classifying learning objectives based on Bloom's taxonomy is shown in Algorithm 1. The functions `tokenize`, `posTag`, and `stem` are provided by the NLTK library.

**Algorithm 1** Classification of Learning Objective by Bloom Level

```
1: function CLASSIFYBLOOMLEVEL(learningObjective)
2:     Input: learningObjective (string)
3:     Output: matchedLevels (set of Bloom levels)
4:     tokens ← TOKENIZE(learningObjective)
5:     tagged ← POSTAG(tokens)
6:     verbs ← EXTRACTVERBS(tagged)
7:     stems ← STEM(verbs)
8:     matchedLevels ← {}
9:     for all level in BloomLevels do
10:        if ANYMATCH(stems, bloomVerbs[level]) then
11:            MATCHEDLEVELS.ADD(level)
12:        end if
13:    end for
14:    return matchedLevels
15: end function
```

The final output is a set of matched Bloom levels. This results from the fact that a single learning objective can contain multiple verbs, each linked to a different cognitive process. Several strategies can be applied to solve this problem. One approach is to select the level associated with the first matched verb. Alternatively, the instructor can choose the most appropriate level from the identified candidates. Both approaches result in a distinct selected level that then serves as input to the second stage of taxonomy-based assignment selection.

The following subsection builds upon this result by selecting example assignments aligned with the chosen Bloom level.

### 4.1.2 Assignment Selection

After identifying the Bloom level of a learning objective, the next step is to select assignment types that reflect the corresponding cognitive complexity. This subsection explains how predefined example assignments are used to guide the generation of exercises that are pedagogically appropriate for each level.

The taxonomy-based assignment selection follows a simple idea: once the Bloom level of a learning objective has been identified, the corresponding cognitive depth of the required exercise is also known. To support this, we provide example assignments that illustrate what an exercise for the identified cognitive level might look like. This is particularly helpful for the prompt and the language model, as it offers additional context and shows how such exercises are typically structured. The idea is to use few-shot prompting, where the model is presented with several examples of what the expected output should look like. In our context of generating programming exercises, we aim to define suitable coding tasks for each Bloom level. The following list outlines what we consider appropriate assignment types per level for programming exercises.

- **Remember**: The exercises focus on recalling facts. Examples include naming keywords or listing built-in functions.
- **Understand**: Students interpret or explain concepts. Typical exercises are used to describe the behavior of the function, identify general types, or explain type errors.
- **Apply**: This level involves the practical use of knowledge. Students may write functions, modify code, or translate between representations like list comprehensions and higher-order functions.
- **Analyze**: Exercises require breaking down code or comparing implementations. Students examine structure or explain causes of run-time behavior and type errors.
- **Evaluate**: Students assess correctness or performance. They might compare two implementations or decide whether the code is compileable.
- **Create**: Students build new solutions. This includes designing data structures or creating complete implementations with type signatures and examples.

For each Bloom level, we have defined two to three example assignments that reflect the respective cognitive requirements. One possible way to structure and reuse these assignments is to represent them in a JSON-based format. The following listing 4.1 shows a possible representation of a Haskell coding exercise for the `Understand` level.

```
{
    "exercise_type": "Identify general types of expressions",
    "exercise_text": "What are the most general types of the
  following expressions? Answer using Haskell type notation.",
    "subtasks": {
        "i": '"404"',
        "ii": '([404,4], ["HTTP", "Tires"])',
        "iii": '[[], ""]',
        "iv": "[filter not]",
    },
},
```

Code listing 4.1: Example JSON Representation of a Coding Assignment.

In summary, a given learning objective is first classified by matching its keywords to Bloom's taxonomy levels. Once the appropriate level is identified, the corresponding example coding assignments are selected. While these examples form one part of the contextual input for exercise generation, relevant lecture content must also be incorporated.

The next section outlines the second component of the pipeline, in which relevant slides are identified and retrieved based on the learning objective.

## 4.2 Content Retrieval

In this section, we describe the second main component of the pipeline: the retrieval of lecture slides relevant to a given learning objective. If we consider the quicksort-related learning objective from previous sections, the related lecture slides specifically address topics or code snippets around quicksort. For the final prompt of exercise generation, these slides can provide additional context on notation and structures from lectures that inform the exercise generation process. The general retrieval process of the relevant lecture slides is illustrated in Figure 4.2.

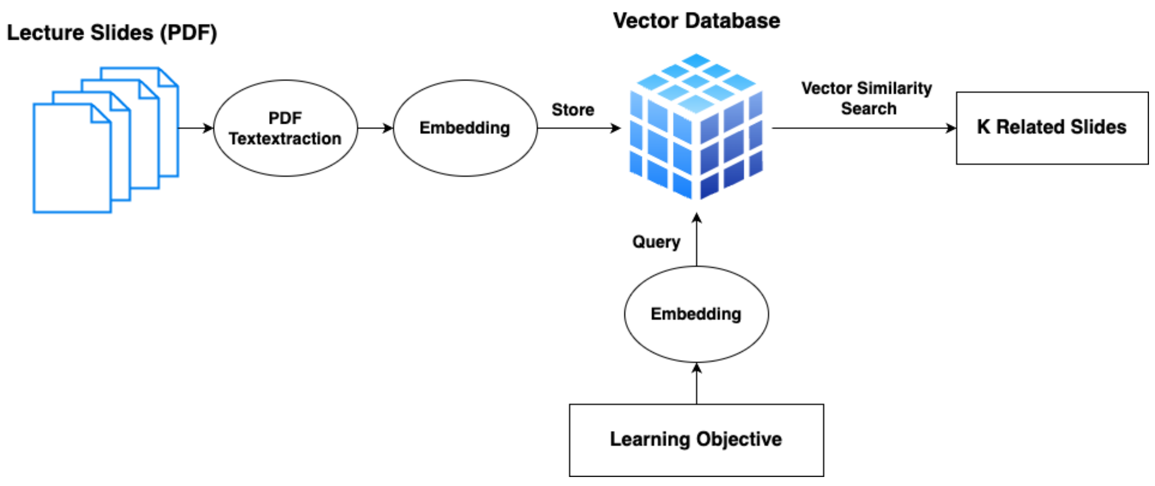


Figure 4.2: Semantic Retrieval Pipeline for Identifying Relevant Lecture Slides.

The pipeline for retrieving relevant lecture slides consists of three main steps. First, the textual content of each slide is extracted. In the second step, both the slide text and the learning objective are converted to semantic vector representations using an embedding model. Finally, a vector similarity search is performed to identify those slides whose embeddings are most similar to that of the learning objective. In the following paragraphs, we will examine each step in detail.

**Extracting Text from Lecture Slides** Initially, textual content is extracted from each lecture slide. We assume that slides are provided in PDF format as this is the prevalent standard. PDFs are primarily designed for visual representation, not structured data. Consequently, extracting text often leads to unstructured output. Although optical character recognition (OCR) is effective for scanned documents or nonselectable texts, our method is limited to direct text extraction techniques. This decision is grounded in the assumption that programming-related lecture slides generally contain selectable text and code snippets. Both are easily retrievable with standard PDF parsing libraries such as PyMuPDF or PyPDF.

**Embedding and Storing Text in Vector Database** Following text extraction, the content of each lecture slide is embedded in a semantic vector space. For every lecture slide, we generate an embedding vector that captures its semantic meaning. Various open-source embedding models are available. The models provided by Hugging Face, such as `sentence transformers / all MiniLM-L6-v2`, offer a good balance between computational efficiency and embedding quality. These semantic embeddings can then be stored in a vector database. FAISS (Facebook AI Similarity Search) is a popular and efficient choice among open-source solutions [29]. Slides are treated individually as discrete chunks. Each slide typically contains enough contextual information to stand alone.

**Performing Similarity Search** The final step performs a semantic similarity search. Given a learning objective, we embed it in the same vector space as lecture slides. We query the vector database to find slides with embeddings that are most similar to that of the learning objective. Typically, cosine similarity is used as the metric for this purpose. For example, the learning objective *"Students should be able to describe the QuickSort algorithm"* will match slides containing relevant terminology and concepts related to quicksort.

The following algorithm illustrates a possible implementation of the described steps, including extraction, embedding, storage, and similarity search.

**Algorithm 2** Related Slide Selection Based on Learning Objective

```
1: function selectRelatedSlides(lectureSlides, learningObjective)
2:     Input: lectureSlides (PDF), learningObjective (string)
3:     Output: relatedSlides (list of slide texts)
4:     slideTexts ← extractTextFromPDF(lectureSlides)
5:     for all slideText in slideTexts do
6:         vector ← embed(slideText)
7:         storeInVectorDB(vector)
8:     end for
9:     queryVector ← embed(learningObjective)
10:    relatedSlides ← vectorSimilaritySearch(queryVector, $k = 2$)
11:    return relatedSlides
12: end function
```

### Summarization of Retrieved Lecture Slides

Once we have identified the relevant lecture slides, we further process these slides employing a large language model (LLM) to summarize their content. This summarization step is used to extract key points from the slide content. In our case, we focus especially on extracting code-related content. To effectively guide the model, we apply common prompt engineering techniques such as role prompting. In this case, the model is instructed to act as a summarization assistant that receives a single slide and produces a concise summary that focuses on technical content and code snippets.

The final output of this stage consists of structured JSON objects representing the relevant lecture slides. These JSON objects serve as contextual input for the final prompt used to generate the programming exercise, which we describe in the next section.

## 4.3 Prompt Engineering

In this section, we describe the implementation of the final component of the pipeline. The results of the previous sections provide the foundational context for the prompt engineering phase.

- Assignment example sheets that illustrate programming tasks appropriate for the given Bloom level
- Summaries of lecture slides identified as relevant to the learning objective

Figure 4.3 shows the final version of the structured prompt used to generate programming exercises. The design of this prompt is based on best practices in prompt engineering and was inspired by the structure proposed in [27]. The prompt is composed of five major components:

**Role Prompting:** The system is explicitly instructed to act as a *learning engineer support bot*. This role-based instruction encourages the language model to adopt a pedagogically grounded perspective when generating exercises.

**Bloom's Taxonomy:** To guide cognitive alignment, the six levels of Bloom's taxonomy are defined within the prompt. By including these descriptions, we aim to guide the LLM toward generating exercises at the appropriate complexity level.

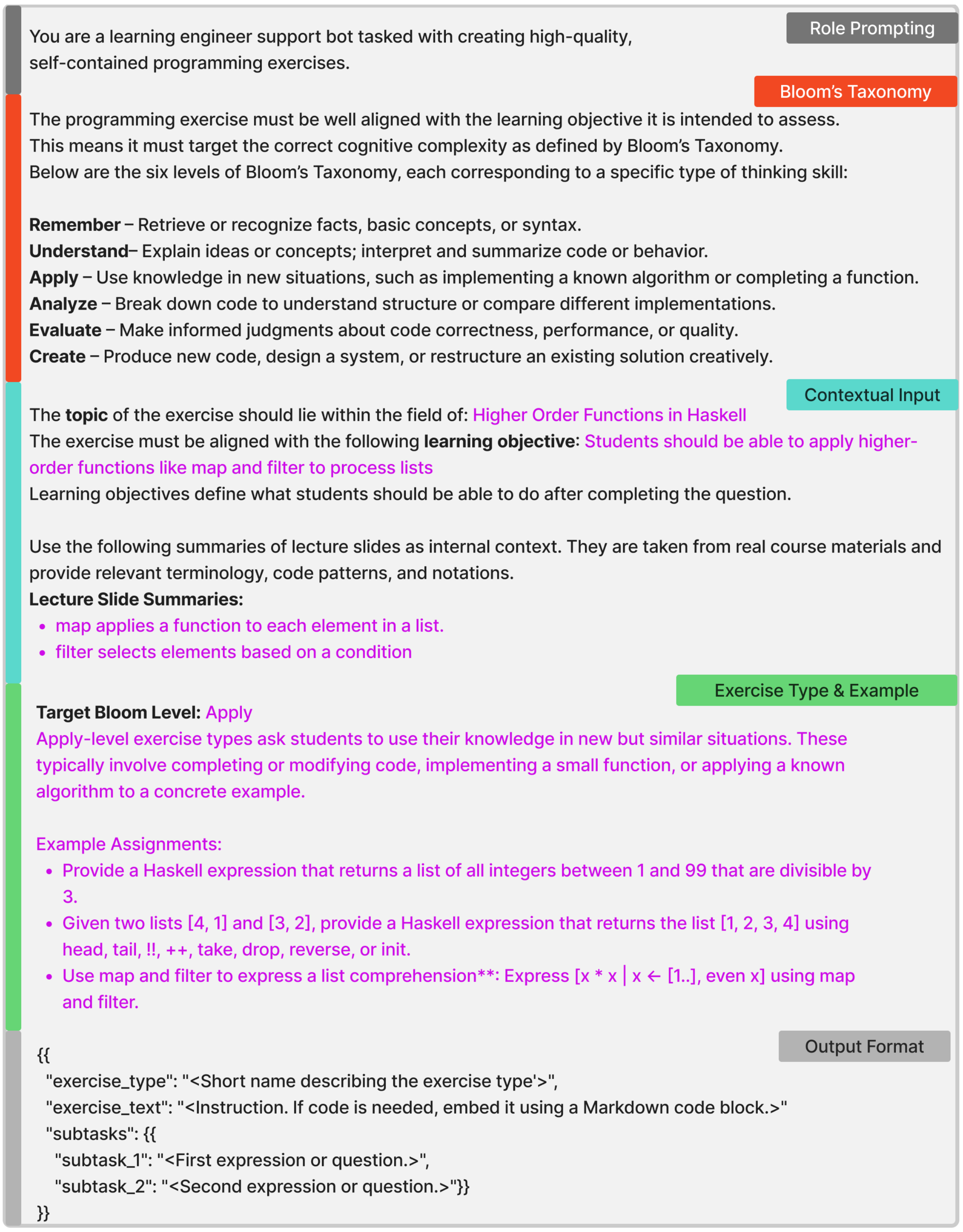
**Contextual Input:** This section includes the exercise topic, the associated learning objective, and summaries of the relevant lecture slides. All three elements serve as contextual input provided by the instructor. The topic defines the general topic area and helps narrow down the scope of the generated exercise. The learning objective describes the specific skill or knowledge that the student is expected to acquire. Finally, the slide summaries provide content that reflects the terminology, concepts, and context presented during the lecture.

**Exercise Type and Examples:** To guide the model toward the appropriate format and style for the task, a set of two to three examples of exercises is included. This component applies the principle of few-shot prompting. By providing these examples, the language model can better understand how the output should be structured.

**Output Format:** The final block defines the expected output format. The model is instructed to generate a JSON object that contains the exercise type, the exercise text, and optional subtasks if applicable. This structured format encourages the model to generate complete and potentially more complex exercises, where multistep tasks are included when appropriate.

The prompt structure is designed to clearly guide the model and align the output with the provided topic, learning objective, and lecture slide content.

Prompt engineering concludes the last main component of the pipeline. Once the prompt is fully constructed, it is passed to the language model that generates the final exercise. In the following chapter, we evaluate the quality of the generated exercises.


Role Prompting
Bloom's Taxonomy
Contextual Input
Exercise Type & Example
Output Format

You are a learning engineer support bot tasked with creating high-quality, self-contained programming exercises.

The programming exercise must be well aligned with the learning objective it is intended to assess.
This means it must target the correct cognitive complexity as defined by Bloom's Taxonomy.
Below are the six levels of Bloom's Taxonomy, each corresponding to a specific type of thinking skill:

**Remember** – Retrieve or recognize facts, basic concepts, or syntax.
**Understand**– Explain ideas or concepts; interpret and summarize code or behavior.
**Apply** – Use knowledge in new situations, such as implementing a known algorithm or completing a function.
**Analyze** – Break down code to understand structure or compare different implementations.
**Evaluate** – Make informed judgments about code correctness, performance, or quality.
**Create** – Produce new code, design a system, or restructure an existing solution creatively.

The **topic** of the exercise should lie within the field of: Higher Order Functions in Haskell
The exercise must be aligned with the following **learning objective**: Students should be able to apply higher-order functions like map and filter to process lists
Learning objectives define what students should be able to do after completing the question.

Use the following summaries of lecture slides as internal context. They are taken from real course materials and provide relevant terminology, code patterns, and notations.
**Lecture Slide Summaries:**
- map applies a function to each element in a list.
- filter selects elements based on a condition

**Target Bloom Level:** Apply
Apply-level exercise types ask students to use their knowledge in new but similar situations. These typically involve completing or modifying code, implementing a small function, or applying a known algorithm to a concrete example.

Example Assignments:
- Provide a Haskell expression that returns a list of all integers between 1 and 99 that are divisible by 3.
- Given two lists [4, 1] and [3, 2], provide a Haskell expression that returns the list [1, 2, 3, 4] using head, tail, !!, ++, take, drop, reverse, or init.
- Use map and filter to express a list comprehension**: Express [x * x | x ← [1..], even x] using map and filter.

```
{{
 "exercise_type": "<Short name describing the exercise type'>",
 "exercise_text": "<Instruction. If code is needed, embed it using a Markdown code block.>"
 "subtasks": {{
  "subtask_1": "<First expression or question.>",
  "subtask_2": "<Second expression or question.>"}}
}}
```

Figure 4.3: Structured Prompt Template for Programming Exercise Generation. The purple area indicates the dynamic input, while the other sections represent fixed components of the prompt. Both the structure and the visual layout are inspired by [27].

# 5 Evaluation

In this chapter, we describe the semi-automated evaluation of the generated programming exercises. The goal is to assess their syntactic correctness, alignment with learning objectives, and consistency with Bloom's taxonomy levels. The evaluation is structured around the following research questions:

1. *RQ1: Are the generated programming exercises grammatically correct and readable for university-level students, and do the embedded Haskell code snippets compile without syntax errors?*
2. *RQ2: To what extent do the generated exercises make use of the contextual information provided in the related lecture slides?*
3. *RQ3: To what extent do the generated exercises align with the defined learning objectives and reflect the intended cognitive level according to Bloom's taxonomy?*

This chapter is structured as follows: First, we describe the selection of open-source language models. Next, we describe the experimental resources that define the lecture slides, the learning objectives, and the topics used to generate the programming exercises. This is followed by an automated evaluation pipeline that assesses the generated exercises against several quality criteria. After that, we present and discuss the evaluation results that are structured around the research questions. Lastly, we highlight limitations that were observed during the evaluation process.

## 5.1 Setup

This section outlines the experimental setup and the selection of open-source models used in this work.

The selected language model had to satisfy three requirements: (1) open-source availability for local deployment, (2) strong code understanding to interpret related code in lecture slides, and (3) reliable code generation to create programming exercises.

Several code-related benchmarks are commonly used to assess the performance of language models along coding tasks. Among these, HumanEval assesses the generation of functional code by requiring models to produce small programs that pass hidden unit tests [30]. The MBPP dataset (Mostly Basic Programming Problems) measures a model's ability to generate correct code from natural language descriptions of simple programming challenges [31]. MultiPL-E and MCEval extend the scope to include reasoning and code understanding [32, 33].

We found that Qwen2.5-coder, developed by Alibaba, satisfies all our requirements. The model is open-source and shows strong performance on several coding benchmarks [34]. Table 5.1 compares the performance of Qwen2.5-coder (32 billion parameters) with DeepSeek Coder V2 (also 32 billion parameters and open-source) and

| **Benchmark** | **Qwen2.5-Coder 32B** | **GPT-4o (2024.08)** | **DeepSeek Coder V2** |
|---|---|---|---|
| HumanEval | **92.7** | 92.1 | 88.4 |
| MBPP | **90.2** | 86.8 | 89.2 |
| EvalPlus | **86.3** | 84.4 | 83.8 |
| MultiPLE | 79.4 | 79.1 | **79.9** |
| McEval | **65.9** | 65.8 | 62.9 |
| CRUXEval-O | 84.2 | **89.2** | 75.1 |
| BigCodeBench | **38.3** | 37.6 | 36.3 |

Table 5.1: Coding Benchmark Comparison between Qwen2.5 Coder and Commercial Models [34].

OpenAI's commercial GPT-4o model. Qwen2.5-coder outperforms the other models on most benchmarks or exhibits only minimal differences, which led us to select it as the model for our experiments.

However, the experiments were carried out locally on a 2023 MacBook Pro equipped with an Apple M2 Pro processor and 16 GB of RAM. This hardware limitation restricted us to using a model with a smaller number of parameters. For this reason, we selected the Qwen2.5-coder model with 7 billion parameters.

| **Benchmark** | **Qwen2.5 Coder 32B** | **Qwen2.5 Coder 14B** | **Qwen2.5 Coder 7B** |
|---|---|---|---|
| HumanEval | **92.7** | 89.6 | 88.4 |
| MBPP | **90.2** | 86.2 | 83.5 |
| EvalPlus | **86.3** | 84.9 | 81.9 |
| MultiPLE | **79.4** | 76.5 | 72.1 |
| McEval | **65.9** | 61.9 | 60.3 |
| LiveCodeBench | **31.4** | 23.4 | 18.2 |
| CRUXEval-O COT | **83.4** | 79.5 | 65.9 |
| BigCodeBench | **38.3** | 29.4 | 29.4 |

Table 5.2: Coding Benchmark Comparison across Qwen2.5 Coder Model Sizes [34].

Table 5.2 shows the performance of *Qwen2.5-coder* models with different parameter sizes. As the results indicate, even the 7B model achieves strong performance in most coding benchmarks.

Another open-source model was selected to summarize the relevant lecture slides. The goal was to choose a model well suited for general-purpose summarization. For this purpose, we used the recently released *Gemma 3* model developed by Google

[35]. We used the smaller 4 billion parameter version to ensure time efficiency, as each relevant lecture slide had to be processed individually by the language model.

**Decoding Parameters** The Gemma 3 and Qwen2.5-Coder models were executed locally using Ollama, a lightweight framework to run large language models on local machines [36]. For both models, we used the default decoding parameters provided by the Ollama runtime. Specifically, we set the `temperature` to 1.0. The parameters `top_p`, `frequency_penalty`, and `presence_penalty` were kept at their default values (1.0, 0.0, and 0.0, respectively). We increased the default `num_ctx` value to 4096 for exercise generation. This was necessary because our prompt becomes relatively large due to the inclusion of slide summaries, learning objectives, and few-shot examples.

Now that the open-source models have been selected, the next section describes the experimental environment. It presents the topics, learning objectives, and lecture slides that were used to generate the programming exercises.

## 5.2 Experiment

This section describes the experimental resources used to generate programming exercises. Recall that the system developed in this work requires three inputs provided by the instructor: a topic, a learning objective, and lecture slides. In addition, assignment sheets for each Bloom level to define the structure and style of the generated questions. The following paragraphs describe all the input components and resources used in the experiment.

**Learning Objectives and Topics** To evaluate the system across different levels of cognitive complexity, we created one example learning objective for each Bloom level. These were designed with the help of ChatGPT (GPT-4o) to ensure that they represent meaningful and realistic learning objectives in the context of programming. We generated ten exercises per learning objective, resulting in a total of 60 exercises for evaluation. The final set of topics and learning objectives used in our evaluation is summarized in Table 5.7.

**Course Materials** For the contextual input regarding lecture content, we used lecture slides from the Functional Programming course taught at TUHH by the STS Institute. The slides are real-world teaching materials that focus on Haskell programming concepts. In total, 504 PDF slides were provided to the system as a complete set. For the similarity search, we set k = 3 to retrieve the three most relevant slides from the 504 lecture slides.

**Example Assignments** To create example assignments, we selected exercises from the laboratory sessions of the Functional Programming course offered by the STS Institute. For each Bloom level, we identified 2–3 representative exercises from the lab sheets. These examples serve as templates to align the generated exercises with the expected style and difficulty per Bloom level.

Now that the experimental resources have been defined, the next section introduces the automated evaluation pipeline used to evaluate the quality of the generated exercises.

## 5.3 Automated Evaluation Pipeline

This section describes the stages of our automated evaluation pipeline. The goal was to evaluate the exercises generated with minimal manual effort. To achieve this, we identified a set of quality aspects that can be evaluated using automated methods. The pipeline consists of five stages, each designed to assess one specific criterion. In the following, we provide a detailed description of each stage.

**Grammar:** The first stage verifies the grammatical correctness of the generated exercises. This can be done automatically using reliable tools or libraries. In our evaluation, we used `LanguageTool`, an open-source Python library for grammar analysis. `LanguageTool` applies a set of predefined rules to input text and reports errors when these rules are violated. We excluded several rules from the evaluation, as they incorrectly flagged code elements embedded in the exercises. Examples include rules that expect sentences to begin with a capital letter (which is often not the case in code snippets) or rules that report double punctuation (e.g., expressions like `[1..10]`). These rules were disabled to focus on elements of natural language.

**Readability:** The second stage evaluates the readability of the generated exercise text. Since the exercises are intended for university students, we aim to assess whether the language used meets college-level readability. Among various readability metrics, the Flesch Reading Ease Score provides an indicator of how easily a text can be understood [37]. The score is based on the average length of the sentence and the average number of syllables per word. It is calculated using the following formula:

$$206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right) \tag{5.1}$$

The score ranges from 0 to 100. Higher values indicate easier readability, while lower values correspond to more complex texts. Table 5.3 provides an interpretation of the score ranges.

Since the generated programming exercises are targeted at university students, an ideal Flesch score lies between 30 and 50, which corresponds to college-level difficulty.

**Code Compilation:** The third stage evaluates the syntactic correctness of the code embedded in the generated exercises. Some exercises include code snippets that

| Score Range | School Level |
|---|---|
| 90–100 | 5th grade |
| 70–90 | 6th to 7th grade |
| 60–70 | 8th to 9th grade |
| 50–60 | 10th to 12th grade |
| 30–50 | College |
| 0–30 | College graduate |

Table 5.3: Interpretation of Flesch Reading Ease Scores [38].

students are expected to analyze or answer questions about. To verify whether the generated code is syntactically correct, we perform an automated compilation check. We developed an extraction script that identifies all code segments embedded within the exercise text and writes them to a temporary Haskell file. This file is then executed using the Glasgow Haskell Compiler (`runghc`). If the code compiles without errors, the code within the exercise is considered syntactically correct. If the compilation fails, the corresponding exercise is flagged as syntactically incorrect.

**Bloom Level Consistency:** The fourth stage evaluates whether the cognitive complexity of a generated exercise aligns with its intended Bloom level. For example, if an exercise was generated with the target level *Remember*, we would expect it to involve simple recall tasks. The key question is whether the final exercise actually reflects this intended level of complexity.

To assess this, we used large language models to automatically classify each generated exercise. Each model received the full exercise text and was instructed to assign it to one of the six Bloom levels. The prompts followed established prompt techniques, such as role prompting. In addition, we include detailed descriptions of Bloom's taxonomy and its application to programming-related tasks. We used open-source models to ensure reproducibility and transparency. The first model we use is Gemma 3. It was selected based on the assumption that general-purpose models perform well in understanding instructions and interpreting the conceptual structure of exercises. The second model is Qwen2.5-coder, which was also used for the generation of exercises. It was chosen for its strong performance in code-related tasks, as all generated exercises involve programming content.

**Lecture Slide Similarity:** The final stage measures the semantic similarity between the generated exercise and the relevant lecture slide summaries. Both the exercise and the summaries are first converted into vector representations. The cosine similarity is then calculated between the exercise vector and each slide summary vector. Higher similarity scores indicate stronger semantic alignment, suggesting that the exercise likely incorporates contextual information from the slides.

Each generated exercise passes through the entire evaluation pipeline. The individual stages correspond to the different research questions addressed in the following section.

## 5.4 Results

This section presents the results of our evaluation. A total of 60 exercises were generated and processed through the automated evaluation pipeline. The following subsections are structured by research questions. For each subsection, we begin by describing the aspects that were automatically evaluated by the pipeline, followed by the corresponding results. For aspects that could not be automatically evaluated, we include representative examples.

#### RQ1: Are the generated programming exercises grammatically correct and readable for university-level students, and do the embedded Haskell code snippets compile without syntax errors?

The first research question examines whether the generated exercises are linguistically correct, appropriate for university-level students, and syntactically valid in terms of the code they contain. The evaluation of this research question was conducted entirely automatically and is based on the first three stages of the evaluation pipeline: grammar, readability, and code compilation.

Table 5.4 summarizes the average results per Bloom level, grouped row-wise. The values reflect four evaluation metrics: the percentage of grammatically correct exercises, the average Flesch Reading Ease score, the number of exercises containing code snippets, and the percentage of those snippets that compiled successfully.

| **Bloom Level** | **Grammar (%)** | **Flesch** | **Code Snippets** | **Compiles (%)** |
|---|---|---|---|---|
| Remember | 100 | 83.6 | 0 | – |
| Understand | 100 | 61.4 | 0 | – |
| Apply | 90 | 38.0 | 5 | 60 |
| Analyze | 90 | 42.3 | 5 | 100 |
| Evaluate | 90 | 32.6 | 8 | 50 |
| Create | 100 | 57.7 | 0 | – |

Table 5.4: Average Evaluation Metrics (RQ1). The table reports grammar correctness, average Flesch readability scores, the number of exercises containing code snippets, and code compilation success rates per Bloom level. A total of 10 exercises was generated for each level.

The evaluation results indicate that the generated exercises achieve a high level of grammatical correctness across all Bloom levels. Each level reached a grammar accuracy of at least 90%. Minor errors occurred only in the *Apply*, *Analyze*, and *Evaluate* levels. These levels included exercises with embedded code snippets. Although several grammar rules were disabled, the grammar checker still flagged issues related to code formatting. These issues are considered false positives and do not represent actual grammatical errors. Overall, the results show that the generated exercises are grammatically correct.

Another aspect we assess in this research question is whether the exercise text is appropriate for a university-level audience. For this, the Flesch readability score provides useful information. We observed that exercises at the *Apply*, *Analyze*, and *Evaluate* levels tend to have lower Flesch scores. These values fall within the expected range of 30-50 for university-level texts. This is consistent with the fact that these Bloom levels typically require more complex reasoning and cognitive effort from students. In contrast, the *Remember* and *Understand* levels show higher readability scores. *Remember* reached a score of 83.6, which is expected due to short and simple recall questions like *"What is not a built-in type in Haskell?"*. *Understand* scored 61.4, reflecting the descriptive character of these exercises. Although both levels show higher readability scores, this is reasonable as these exercises are less complex and ask questions about fundamental concepts. Surprisingly, the *Create* level reached a relatively high readability score of 57.7. We observed that many of these exercises are formulated as clear and concise instructions, such as defining a custom data type with specific constraints. This simple phrasing leads to high readability, despite the underlying conceptual complexity. In general, even if some levels show higher readability scores, this aligns with the respective cognitive demands. These results indicate that most of the exercises are readable appropriately for a university-level audience.

In terms of code validity, only the Bloom levels *Apply*, *Analyze*, and *Evaluate* included Haskell code that could be tested. We identified five *Apply* exercises with code snippets. Three out of five were compiled successfully (60%). All five *Analyze* exercises compiled without errors (100%). For *Evaluate*, we found eight exercises containing code. Four out of eight were compiled successfully (50%). In total, 17 exercises included compilable Haskell code. Of these, 12 passed the compilation check, resulting in an overall success rate of approximately 70%. In some cases, compilation failed due to minor structural issues, even though the code was syntactically valid. For example, a type signature was provided without a corresponding function definition:

```haskell
sumOfSquaresEven :: [Int] -> Int
main = return ()
```

Code listing 5.1: Missing Definition for Declared Function.

Although the type annotation itself was correct, the missing function body caused the compiler to reject the code.

However, some compilation errors were appropriate and indicated genuine semantic issues. For example, a generated exercise applied the standard function `filter` to a custom-defined data type `List`, which is not compatible with the expected list type `[a]`. Since `filter` operates only on standard Haskell lists and there is no type-class instance for `List`, the compiler correctly raised a type error:

```haskell
type OddSumFunc = List -> Int

inefficientOddSum :: OddSumFunc
inefficientOddSum xs = sum $ filter odd xs
```

Code listing 5.2: Invalid Use of Filter on Non-Standard List Type.

In summary, the results in Table 5.4 show that the generated exercises are grammatically correct and readable across all Bloom levels. About 70% of the exercises with the Haskell code have been successfully compiled. This indicates that the model can generate syntactically valid code, but still produces errors in some cases. Future work could explore the use of larger models with more than 7B parameters to improve the code quality.

### RQ2: To what extent do the generated exercises make use of the contextual information provided in the related lecture slides?

The second research question investigates to what extent the generated programming exercises leverage the contextual information from the most relevant lecture slides. To address this question, we use a semi-automated approach. The automated part corresponds to stage 5 (Lecture Slide Similarity) of the evaluation pipeline, where embeddings of slide summaries and generated exercises are compared using cosine similarity. In addition to that, we will look at concrete examples.

| **Bloom Level** | **Avg. Vector Similarity to Slides** |
|---|---|
| Remember | 0.506 |
| Understand | 0.607 |
| Apply | 0.445 |
| Analyze | 0.456 |
| Evaluate | 0.390 |
| Create | 0.498 |

Table 5.5: Average Vector Similarity to Lecture Slides. The table shows the average semantic similarity between vector embeddings of generated exercises and relevant lecture slide summaries.

Table 5.5 shows the average semantic similarity between the generated exercises and the corresponding lecture slide summaries. The cosine similarity ranges from -1 (opposite meaning) to 1 (identical meaning). A similarity of 0 indicates that there is no relation. Our results show that all similarity scores lie in a moderate range around 0.5. The *Understand* level has the highest similarity (0.607), while *Evaluate* is the lowest (0.39). Across all Bloom levels, the generated exercises show a certain degree of semantic alignment with the lecture slide content. It is worth noting that extremely high similarity scores would not be desirable. The exercises and the lecture slide summaries already differ significantly in structure and purpose.

However, similarity scores alone do not confirm whether exercises are explicitly based on the lecture summaries. To examine this further, we analyze two concrete examples. Figure 5.1 shows a generated exercise for the *Understand* level. The exercise asks students to identify the most general types of several Haskell expressions.

**Generated Exercise for the Understand Level**

Which of the following Haskell expressions are well typed? Explain why each expression is either well-typed or not.
a.) `head [404, 200]`
b.) `length "Hello"`
c.) `filter odd [1, 2, 3]`
d.) `max (1, 2) (3, 4)`

Figure 5.1: Generated Exercise targeting the Understand Level of Bloom's Taxonomy.

The following lecture slide summaries provided contextual input to the exercise generation process:

1. *The slide introduces type classes in Haskell. A type class, `Eq`, defines operators like `(==)` and `(/=)` for equality and inequality checks. Many types such as `Int`, `Char`, and `Bool` are instances of `Eq`. Function types are not.*

2. *Types in Haskell are names for sets of related values. The `Bool` type denotes the values `True` and `False`. Variables and function arguments have types.*

3. *At the end of the course, students will be able to write Haskell programs, understand functional programming, and describe types and related concepts.*

The provided slide summaries cover core Haskell concepts such as basic types, type inference, and the `Eq` type class. Although the generated exercise does not repeat exact phrases from the slides, it is clearly aligned with their content. Students must reason about types such as `Int`, `Char`, and lists, and assess whether expressions are well-typed concepts introduced in summaries (1) and (2). Summary (1) is relevant to understanding equality, while summary (2) supports reasoning about primitive types. In contrast, summary (3) offers a general course-level objective and does not directly contributes to the exercise.

Another example is shown in Figure 5.2. The exercise was generated for the *Create* level and asks students to define a custom data type `LibraryBook`. This aligns with typical *Create*-level tasks, which involve designing and constructing new structures.

The following lecture slide summaries were retrieved as relevant to this exercise:

1. *This slide outlines the third design recipe (DR-D) for data declarations. It emphasizes introducing proper names for data types, converting different data sorts into constructors, and determining if constructors require arguments of*

**Generated Exercise for the Create Level**

You are tasked with designing a data type to represent a `LibraryBook` in a library system. Each book should have attributes like title, author, ISBN, publication year, and availability status.
a.) Define the `LibraryBook` data type with the specified attributes.
b.) Create an instance of the `Show` type class for `LibraryBook`.
c.) Implement the `checkOutBook` function.

Figure 5.2: Generated Exercise Targeting the Create Level of Bloom's Taxonomy.

*specific types. New types should be instances of type classes, starting with those from the Prelude.*

2. *The slide outlines the design recipe (DR-D) for recursive types. Key steps include: 1) Introducing a name for the data type. 2) Turning different data sorts into constructors. 3) Determining if constructors require arguments and their types. 4) Identifying Prelude type classes the new type instance of.*

3. *Haskell utilizes record syntax for defining data structures. The syntax is: `data CoursePlan = CoursePlan { ... }`. Field names are explicitly defined within the curly braces, separated by commas. Haskell automatically generates access functions for each field.*

Summary (1) introduces the DR-D recipe, which explains how to define data types and constructors. This relates to the general task of designing the `LibraryBook` data type. Summary (2) focuses on recursive structures and type signatures, which are relevant in the broader context of implementing functions like `checkOutBook`. Summary (3) presents the syntax of record definitions in Haskell, including field accessors and structure. Although the generated exercise does not directly reference these slides, it touches on similar concepts.

We conclude that the generated exercises make moderate use of the contextual information from the related lecture summaries. This is supported by similarity scores that indicate semantic overlap and by examples in which relevant concepts from the slides are reflected in the exercises. However, it is not always clear to what extent the model actively draws on the content of the slide summaries during generation. In many cases, the summaries describe general design principles or abstract syntax rules that may not be translated directly into a concrete programming task. Although topical overlap exist, it remains difficult to determine whether the slide content was actually used as an instructional basis.

## RQ3: To what extent do the generated exercises align with the defined learning objectives and reflect the intended cognitive level according to Bloom's taxonomy?

The third research question examines whether the generated exercises align with the defined learning objectives and whether their cognitive complexity matches the originally intended Bloom level. To evaluate this, we refer to stage 4 (Bloom Level Consistency) of the automated evaluation pipeline. In this stage, open-source language models are used to classify each generated exercise into one of the six Bloom levels. A total of 60 exercises were generated, 10 exercises per Bloom level. Table 5.6 summarizes the classification results. Each row corresponds to one Bloom level. The columns show the percentage of correct classifications made by the two open-source models.

| **Bloom Level** | **Gemma 3 (%)** | **Qwen2.5-Coder (%)** |
|---|---|---|
| Remember | 100 | 72.7 |
| Understand | 80 | 100 |
| Apply | 20 | 70 |
| Analyze | 70 | 70 |
| Evaluate | 53.3 | 93.3 |
| Create | 90 | 100 |

Table 5.6: Bloom Level Prediction of Open-Source LLMs. Each column represents an open-source LLM and shows its average classification accuracy (%) in correctly identifying the intended Bloom level of the generated exercises. Results are grouped row-wise by Bloom levels.

With the exception of *Remember*, we observe that Qwen2.5-Coder consistently achieves higher classification accuracy than Gemma 3. The Gemma 3 model showed particular difficulties at the *Apply* and *Evaluate* levels, both of which frequently included embedded code snippets. One possible explanation is that Gemma 3 is a general-purpose model and, unlike Qwen2.5-Coder, not specifically optimized for code-related tasks.

However, it should be noted that the classification performance of Gemma 3 should not be dismissed as completely inaccurate. The exercise shown in Figure 5.3 was generated for the *Evaluate* level. The learning objective was: *Students should be able to evaluate multiple Haskell implementations of the same function based on provided code examples and justify which implementation is more appropriate under specific evaluation conditions.*

The exercise presents two Haskell implementations of the factorial function and asks students to evaluate which one is more efficient and preferable. This matches directly the definition of the *Evaluate* level, as students are required to make reasoned judgments based on code. The alignment with the learning objective is therefore clear.

**Generated Exercise for the Evaluate Level**

Consider the following two Haskell functions that compute the factorial of a number.

```haskell
factorial1 n = product [1..n]
factorial2 0 = 1
factorial2 n = n * factorial2 (n - 1)
```

a.) Which implementation is more efficient for large input values? Explain your reasoning.
b.) Under what conditions might the second implementation be preferable despite its inefficiency?

Figure 5.3: Generated Exercise Targeting the Evaluate Level of Bloom's Taxonomy.

However, Gemma 3 frequently misclassified these exercises into the *Analyze* level. This confusion is understandable, as these exercises also include elements of comparison and structural analysis. In this particular example, students are also asked to compare two different implementations of the factorial function.

Another frequent misclassification occurred at the *Apply* level, where Gemma 3 correctly classified only 2 of 10 exercises (20%). Figure 5.4 shows an example exercise generated for this level. The exercise was created based on the learning objective: *Students should be able to reimplement given Haskell expressions embedded in the task as equivalent recursive functions, without relying on higher-order functions.*

**Generated Exercise for the Apply Level**

Rewrite the following non-recursive function as an equivalent recursive version using pattern matching and basic list operations. Do not use any higher-order functions such as `map`, `filter`, or `fold`.

```haskell
divisibleByThree xs = [x | x <- xs, x `mod` 3 == 0]
```

Figure 5.4: Generated Exercise Targeting the Apply Level of Bloom's Taxonomy.

The exercise is clearly aligned with the learning objective. Students are required to apply recursion to rewrite a given function. This reflects the definition of the *Apply* level. However, Gemma 3 often misclassified these tasks as *Understand*. A possible reason is that students must first understand the concept of recursion and also the structure of the given function before they can apply it. Nevertheless, the primary cognitive demand lies in the application of known concepts, which is clearly reflected in the structure of the exercise. While there is some cognitive overlap between the *Understand* and *Apply* levels, Gemma 3 consistently misclassified the exercise in this case.

Overall, the evaluation results show that the generated exercises meet several important quality criteria. All exercises were grammatically correct, with only minor issues caused by embedded code snippets. The readability matched expectations for university-level texts, and approximately 70% of the code snippets were compiled successfully. Compilation errors were often due to structural or semantic issues, including missing definitions or mismatches of types. The exercises also showed moderate semantic alignment with the lecture slide summaries. Similarity scores suggest that key concepts from the slides were present in the generated exercises, even if not directly reused. Regarding Bloom level classification, most exercises were correctly classified by both models. The models performed particularly well at the *Create* and *Understand* levels. Lower scores at *Apply* and *Evaluate* were likely due to conceptual overlap with adjacent levels. Manual inspection confirmed that the generated exercises were aligned with the defined learning objectives and reflected the expected cognitive complexity.

## 5.5 Limitations

In this section, we address several limitations observed during the generation of exercises.

In some cases, the generated exercises were too similar to the example assignments provided in the prompt. A few times, the output was nearly identical. This behavior is undesirable, as the language model should use the examples as guidance to understand the expected structure and style for each Bloom level, but not simply copy them. A possible reason is that the model was too narrowly focused on the example assignments. Although we explicitly stated in the prompt that the examples should serve only as guidance, this issue could not be fully resolved.

Another issue was a lack of variation in some of the generated exercises. For example, at the *Create* level, many tasks involved defining a new data type and repeatedly used the same concepts, such as a library, a book, or a student. Additional examples from other domains would have been desirable to increase variety and avoid repetition. There are several possible explanations for this. The learning objectives may have been too narrowly defined, leaving little room for variation. Additionally, the use of a relatively small language model could have limited the diversity of the output.

These limitations are not necessarily a result of the general approach. They may be related to the limited capacity of small language models or to prompt formulations that could be further refined.

| **Bloom Level** | **Topic** | **Learning Objective** |
|---|---|---|
| Remember | Haskell Syntax and Prelude Overview | Students should be able to recognize and recall basic Haskell syntax, keywords, and built-in functions. |
| Understand | Type Inference and Notation in Haskell | Students should be able to describe the types of Haskell expressions and explain how type inference works using standard type notation. |
| Apply | Rewriting Haskell Functions Recursively | Students should be able to reimplement given Haskell expressions embedded in the task as equivalent recursive functions, without relying on higher-order functions. |
| Analyze | Analyzing Variants of Haskell Functions | Students should be able to analyze provided Haskell functions by breaking down its structure. |
| Evaluate | Functional Equivalence and Evaluation Behavior in Haskell | Students should be able to evaluate multiple Haskell implementations of the same function based on provided code examples, and justify which implementation is more appropriate under specific evaluation conditions. |
| Create | User-defined Data Types and Instances in Haskell | Students should be able to design custom data types and implement appropriate type class instances for them. |

Table 5.7: Example Learning Objectives Aligned with Bloom Levels. These examples were used as the basis for generating and evaluating 10 exercises per Bloom level.

# 6 Summary and Future Work

This work presents a new approach to generating programming exercises for university students. The method uses open-source language models and combines two key inputs: learning objectives and lecture materials.

First, the learning objective is classified using Bloom's taxonomy to define the required level of complexity. For each Bloom level, we provided example programming exercises to guide the model in shaping its output. In addition, the approach incorporates content from course slides. A retrieval-augmented pipeline extracts plain text from PDF slides, which is then summarized by a language model to identify key concepts. The final prompt combines the learning objective, the summarized slide content, and a user-defined topic.

We evaluated the approach using Qwen2.5-coder, an open-source language model with 7 billion parameters. The test set consisted of 60 exercises based on six different learning objectives. For the experiment, we used lecture slides from a real university course. Our automated evaluation pipeline showed that the generated exercises were grammatically correct and matched the expected readability level for university students. In our tests, 70% of the embedded code compiled successfully. We also performed manual checks and observed that the exercises aligned with the learning objectives and reflected the content of the summarized lecture slides.

Future work could explore several directions. One important aspect concerns model size. While this work used a relatively small language model, larger variants may lead to better results. In our evaluation, 70% of the generated code compiled successfully. Larger models might produce more robust and syntactically correct code.

Addressing the limitations observed during the evaluation is another potential direction for future work. In several cases, the generated exercises were repetitive or too closely mirrored the example assignments provided in the prompt. Further refinements in prompt engineering may help reduce these effects. In particular, the use of few-shot examples might contribute to overfitting, which future studies could investigate further.

The extraction of key concepts from lecture slides also presents opportunities for further refinement. In our current setup, we limited ourselves to extracting plain text from PDF files. However, lecture slides often include visual elements like diagrams, tables, and graphs, which were not considered. Incorporating this type of information could provide additional context and improve the quality of the generated exercises.

Additionally, alternative methods for processing the extracted text from PDF slides could be explored. Structuring PDF slides is generally a challenging task as they are usually unstructured and primarily designed for visual presentation. In our current approach, we applied an LLM-based summarization step to extract and organize the main concepts. As an alternative, slides could also be converted into a structured JSON format that more clearly separates concepts, definitions, and other relevant content. This structure could be based on visual layout cues like text position or bounding boxes.

Finally, the evaluation setup could be extended. So far, we have tested 60 exercises based on six different learning objectives. All experimental data came from a single Haskell programming course. We also used example assignments from this course to guide the output format of the generated exercises. It would be valuable to repeat the evaluation on a larger scale, using more learning objectives and lecture slides from different programming courses. This would allow for more general conclusions about the quality and adaptability of our approach across programming domains.

# Bibliography

[1] Shen Wang, Tianlong Xu, Hang Li, Chaoli Zhang, Joleen Liang, Jiliang Tang, Philip S. Yu, and Qingsong Wen. "Large Language Models for Education: A Survey and Outlook". In: *arXiv* (2024). DOI: `10.48550/arXiv.2403.18105`.

[2] Changrong Xiao, Sean Xin Xu, Kunpeng Zhang, Yufang Wang, and Lei Xia. "Evaluating Reading Comprehension Exercises Generated by LLMs: A Showcase of ChatGPT in Education Applications". In: *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023).* Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 610–625. DOI: `10.18653/v1/2023.bea-1.52`.

[3] Jacob Doughty, Zipiao Wan, Anishka Bompelli, Jubahed Qayum, Taozhi Wang, Juran Zhang, Yujia Zheng, Aidan Doyle, Pragnya Sridhar, Arav Agarwal, Christopher Bogart, Eric Keylor, Can Kultur, Jaromir Savelka, and Majd Sakr. "A Comparative Study of AI-Generated (GPT-4) and Human-crafted MCQs in Programming Education". In: *Proceedings of the 26th Australasian Computing Education Conference.* ACE 2024. ACM, Jan. 2024, pp. 114–123. DOI: `10.1145/3636243.3636256`.

[4] Zihao Zhou, Maizhen Ning, Qiufeng Wang, Jie Yao, Wei Wang, Xiaowei Huang, and Kaizhu Huang. "Learning by Analogy: Diverse Questions Generation in Math Word Problem". In: *arXiv* (2023). DOI: `10.48550/arXiv.2306.09064`.

[5] David R. Krathwohl. "A Revision of Bloom's Taxonomy: An Overview". In: *Theory Into Practice* 41.4 (2002), pp. 212–218. DOI: `10.1207/s15430421tip4104_2`.

[6] Niklas Meissner, Sandro Speth, Julian Kieslinger, and Steffen Becker. "EvalQuiz – LLM-based Automated Generation of Self-Assessment Quizzes in Software Engineering Education". In: *Proceedings of the Software Engineering und Unterricht 2024 (SEUH 2024).* GI Lecture Notes in Informatics (LNI). Gesellschaft für Informatik (GI), 2024. DOI: `10.18420/seuh2024_04`.

[7] Nicy Scaria, Suma Dharani Chenna, and Deepak Subramani. "Automated Educational Question Generation at Different Bloom's Skill Levels Using Large Language Models: Strategies and Evaluation". In: *Artificial Intelligence in Education.* Springer Nature Switzerland, 2024, pp. 165–179. DOI: `10.1007/978-3-031-64299-9_12`.

[8] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *arXiv* (2021). DOI: `10.48550/arXiv.2005.11401`.

[9] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–133. DOI: `10.1007/BF02478259`.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. “Attention Is All You Need”. In: *arXiv* (2023). DOI: 10.48550/1706.03762.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *arXiv* (2019). DOI: 10.48550/arXiv.1810.04805.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. “Language Models are Few-Shot Learners”. In: *arXiv* (2020). DOI: 10.48550/arXiv.2005.14165.

[13] Rishi Bommasani, Jacob D Hudson, Ehsan Adeli, et al. “On the Opportunities and Risks of Foundation Models”. In: *arXiv* (2021). DOI: 10.48550/arXiv.2108.07258.

[14] Hugo Touvron, Thibaut Lavril, Gautier Izacard, et al. “LLaMA: Open and Efficient Foundation Language Models”. In: *arXiv* (2023). DOI: 10.48550/arXiv.2302.13971.

[15] Zixuan Zhang, Sidi Geng, Yutong Duan, Yikang Liu, Yifan Hou, Zhuoyang Ye, Yingya Li, Yike Liu, et al. “The Effect of Sampling Temperature on Problem Solving in Large Language Models”. In: *arXiv* (2024). DOI: 10.48550/arXiv.2402.05201.

[16] Yujia Jin, Guoyang Wu, Baoxing Yang, Zhuo Liu, Jialong Ding, Xiaoxuan Ma, Qingqing Zeng, Zihan Liu, Wei Wei, Yusheng Wang, et al. “LLM can Achieve Self-Regulation via Hyperparameter Aware Generation”. In: *arXiv* (2024). DOI: 10.48550/arXiv.2402.11251.

[17] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: *arXiv* (2019). DOI: 10.48550/arXiv.1908.10084.

[18] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[19] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. “Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review”. In: *arXiv* (2024). DOI: 10.48550/arXiv.2310.14735.

[20] OpenAI Documentation. URL: https://platform.openai.com/docs/guides/gpt (visited on 05/05/2025).

[21] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *arXiv* (2023). DOI: 10.48550/arXiv.2201.11903.

[22] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. “Self-Consistency Improves Chain of Thought Reasoning in Language Models”. In: *arXiv* (2023). DOI: `10.48550/arXiv.2203.11171`.

[23] Lorin W. Anderson and David R. Krathwohl. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom’s Taxonomy of Educational Objectives.* Longman, 2001.

[24] University of Arkansas. URL: `https://tips.uark.edu/using-blooms-taxonomy` (visited on 05/05/2025).

[25] Ghader Kurdi, Jared Leo, Bijan Parsia, Uli Sattler, and Salam Al-Emari. “A Systematic Review of Automatic Question Generation for Educational Purposes”. In: *International Journal of Artificial Intelligence in Education* 30 (Nov. 2019). DOI: `10.1007/s40593-019-00186-y`.

[26] Subhankar Maity, Aniket Deroy, and Sudeshna Sarkar. “Leveraging In-Context Learning and Retrieval-Augmented Generation for Automatic Question Generation in Educational Domains”. In: *arXiv* (2025). DOI: `10.48550/arXiv.2501.17397`.

[27] Sérgio Silva Mucciaccia, Thiago Meireles Paixão, Filipe Wall Mutz, Claudine Santos Badue, Alberto Ferreira de Souza, and Thiago Oliveira-Santos. “Automatic Multiple-Choice Question Generation and Evaluation Systems Based on LLM: A Study Case With University Resolutions”. In: *Proceedings of the 31st International Conference on Computational Linguistics.* Abu Dhabi, UAE: Association for Computational Linguistics, Jan. 2025, pp. 2246–2260. URL: `https://aclanthology.org/2025.coling-main.154/`.

[28] NLTK. URL: `https://www.nltk.org` (visited on 05/05/2025).

[29] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. “The Faiss library”. In: *arXiv* (2025). DOI: `10.48550/arXiv.2401.08281`.

[30] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba.

"Evaluating Large Language Models Trained on Code". In: *arXiv* (2021). DOI: 10.48550/arXiv.2107.03374.

[31] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. "Program Synthesis with Large Language Models". In: *arXiv* (2021). DOI: 10.48550/arXiv.2108.07732.

[32] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. "MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation". In: *arXiv* (2022). DOI: 10.48550/arXiv.2208.08227.

[33] Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, Zekun Wang, Boyang Wang, Xianjie Wu, Bing Wang, Tongliang Li, Liqun Yang, Sufeng Duan, and Zhoujun Li. "McEval: Massively Multilingual Code Evaluation". In: *arXiv* (2024). DOI: 10.48550/arXiv.2406.07436.

[34] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. "Qwen2.5-Coder Technical Report". In: *arXiv* (2024). DOI: 10.48550/arXiv.2412.15115.

[35] Gemma Team et al. "Gemma 3 Technical Report". In: *arXiv* (2025). DOI: 10.48550/arXiv.2503.19786.

[36] Ollama. URL: https://ollama.com (visited on 05/05/2025).

[37] Rudolf Flesch. "A New Readability Yardstick". In: *Journal of Applied Psychology* 32.3 (1948), pp. 221–233. DOI: 10.1037/h0057532.

[38] Rudolf Flesch. *How to Write Plain English: A Book for Lawyers and Consumers.* Harper & Row, 1979.