



Code Safari: An Ai-Powered Learning Tutor for Children

SUBMITTED BY:

Ahmed Mostafa AbdEl-Rahman

Mazen Gaber Mahmoud

AbdelRahman Mohamed Abdel-Hadi

Aya Abdelmonem Mohammed

Doaa Samir El-Sayed

Under the Supervision of:

Dr. Christine Basta

Faculty of computer and Data Science,
Alexandria university
2024-2025

APPROVAL PAGE

Graduation Project Documentation

Code Safari: An Ai-Powered Learning Tutor for Children

Presented by:

Ahmed Mostafa AbdEl-Rahman

Mazen Gaber Mahmoud

AbdelRahman Mohamed Abdel-Hadi

Aya Abdelmonem Mohammed

Doaa Samir El-Sayed

Advisor:

Dr. Christine Basta

Alexandria university

2024-2025

Acknowledgements

This project was successfully completed thanks to the engagement and contributions of many people, especially those who took the time to offer their well-considered advice and suggestions.

We would like to express our deepest gratitude to our project advisor, Dr. Christine Basta, for her invaluable guidance, unwavering support, and insightful feedback throughout the development of this project. This project would not have been completed if it wasn't for her guidance and care for us and for the project.

We also would like to thank ourselves for not giving up and for overcoming a lot of challenges.

Moreover, we would like to thank all our professors for the quality of teaching they gave us during our studies. A Special thanks to our family and parents for supporting and believing in our work throughout the learning journey.

DECLARATION

We hereby certify that this material, which we now submit for assessment on the program of study leading to the award of Bachelor of Science from Alexandria University is entirely our own work, that we have exercised reasonable care to ensure that the work is original and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of our work.

Date: 2024-2025

Abstract

In today's digital era, equipping young minds with programming skills is no longer a luxury but a necessity. Early exposure to coding fosters critical thinking, problem-solving abilities, and creativity among children, preparing them to thrive in a technology-driven society. Recognizing the importance of engaging educational tools, this project introduces an AI-powered Python tutoring platform specifically designed for children aged 10 to 13. The platform aims to revolutionize the way young learners interact with programming concepts, making the learning process both enjoyable and effective.

The project presents a multifaceted learning environment that combines gamification with advanced artificial intelligence to enhance user engagement and comprehension. Central to the platform is a code multiple-choice questions (MCQ) section structured as an interactive game. Here, learners answer progressively challenging questions generated using a custom Python dataset and a fine-tuned Llama-3.2-3B-Instruct model.

Complementing the MCQ game is a code snippets section, where students are presented with incomplete code from real-world projects. Utilizing datasets and the Salesforce/codegen model, the platform generates these snippets, prompting learners to fill in the missing parts.

The Debugging Detective game offers an engaging and interactive experience where children identify and correct errors in AI-generated code snippets, reinforcing programming concepts through hands-on problem-solving in a fun and accessible way. The courses section hosts curated video content from YouTube playlists, integrated with Gemma3 : 4b model that summarizes videos to bolster understanding. To personalize the learning journey

An "Ask a Question" feature is designed to make learning more interactive and supportive by allowing young learners to seek immediate, easy-to-understand explanations tailored to their level. Powered by the Gemma-2b instruct model, it provides clear, child-friendly answers that help clarify concepts and resolve doubts, ensuring students remain engaged and confident as they progress.

Additionally, we have introduced a novel multimodal capability using the Gemini-1.5-flash model. This feature enables the platform to analyze a user's code alongside its visual output, converting both into detailed textual feedback. It describes what the code does, creating a richer, more intuitive learning experience that bridges coding and visual understanding.

Table of Contents

Chapter 1: Introduction	10
1.1 Problem Statement.....	11
1.2 Goal	12
1.3 Objectives.....	12
1.4 Aim of Work	13
Chapter 2: Background	16
2.1 Introduction	16
2.2 Artificial Intelligence	16
2.3 Deep Learning	16
2.4 Natural Language Processing	17
2.5 Language Models and Transformers	18
2.6 Large Language Models	19
2.6.1 Salesforce/codegen-350M-mono	19
2.6.2 Gemma3: 4B.....	20
2.6.3 Gemma-2b Instruct.....	20
2.6.4 unsloth/Llama-3.2-3B-Instruct.....	21
2.6.5 gemini-1.5-flash	21
2.7 Quantization.....	22
Chapter 3: Methodology and Results	23
3.1 General Description of the Project	23
3.2 Tasks and models	24
3.2.1 MCQ Exercises Section	25
3.2.2 Code Visualization Section	31
3.2.3 Debugging Game.....	33
3.2.4 Code Snippet.....	38
3.2.5 Video Content	43
3.2.6 Chatbot Text-to-Text QA for Kids.....	50
3.3 Environment.....	58
3.3 Tutor.....	60
3.3.1 MCQ generation feature	60
3.3.2 Code Visualizer feature	63
3.3.3 Debugging Detective game feature	66

3.3.4 Code Snippet feature	68
3.3.5 Video Content	72
3.3.6 Chatbot Text-to-Text QA for Kids.....	79
Chapter 4: Conclusion.....	84
4.1 Future work.....	84
4.2 Challenges	85
4.3 Conclusion.....	86
Appendix A.....	87
A demo of the tutor	87
References	91

LIST OF FIGURES

Figure 1: Our project Logo	11
Figure 2: Transformer Architecture	18
<i>Figure 3: Project Components</i>	24
Figure 4: MCQ Dataset.....	26
Figure 5: Llama 3.2-3B-Instruct Prompt Engineering.....	28
Figure 6: MCQ Training Results.....	30
Figure 7: Debugging Game Dataset	33
Figure 8: Code Snippet Dataset.....	39
Figure 9: Code Snippet training results.....	42
Figure 10: Chatbot QA Dataset	52
Figure 11: Chatbot QA Training Results	56
Figure 12: Dashboard.....	87
Figure 13: Video Lessons.....	87
Figure 14: Python Assistant	88
Figure 15: AI Artist	88
Figure 16: Code Snippets	89
Figure 17: Debugging Detective.....	89
Figure 18: Exercises.....	90

LIST OF Tables

Table 1: Environment Software	58
Table 2: Environment hardware	59

Chapter 1: Introduction

In today's digital era, programming has become a fundamental skill, especially for young learners. With over 54% of parents believing that coding is an essential skill for children, there is a growing demand for engaging and accessible tools to teach programming. Traditional methods often fail to capture children's interest, emphasizing the need for innovative approaches.

Reports show that 80% of students aged 10-13 are more likely to engage in gamified learning experiences, making interactive platforms crucial for education. Additionally, with 73% of educators highlighting the benefits of AI in personalized learning, integrating AI into programming education is pivotal.

The AI-Powered Python Programming Tutor addresses these challenges by providing a comprehensive learning platform tailored for children aged 10-13 years. The system features gamified elements such as a Code MCQ Game, a Code Snippets Section, and AI-powered tools for interactive learning. This approach aligns with the increasing adoption of digital learning tools, where 65% of students report higher retention rates when learning through interactive platforms.

This project not only makes programming education accessible but also fosters interest and skill development, preparing young learners for a technology-driven future.

1.1 Problem Statement

In the current educational landscape, traditional methods of teaching programming to children often fail to capture their interest and sustain engagement. Despite the growing importance of programming as a fundamental skill, **75% of students aged 10-13** report losing motivation when faced with static, lecture-based learning methods.

Challenges in programming education include:

- Lack of interactive and gamified learning experiences, which **80% of young learners** find more engaging.
- Limited access to personalized feedback and guidance tailored to the needs of children.
- Difficulty in introducing real-world programming concepts in an age-appropriate and approachable manner.
- Absence of a structured, goal-oriented system to track progress and reward achievements, which discourages continued learning.



Figure 1: Our project Logo

1.2 Goal

The primary goal of this project is to create an AI-powered programming tutor for children aged 10-13 years, designed to make learning Python programming engaging, interactive, and rewarding.

The system aims to achieve the following:

1. Introduce a gamified learning approach with Code MCQ Games and Code Snippets Challenges.
2. Incorporate a structured progress tracking system with badges and rewards to motivate learners.
3. Offer a comprehensive platform featuring courses, games, and interactive chatbots to summarize and explain Python concepts.
4. Leverage NLP and machine learning to generate age-appropriate content and validate user responses effectively.

1.3 Objectives

1. Interactive Code MCQs: Implement an AI-powered system to generate multiple-choice coding questions in a gamified format, allowing students to test their understanding of Python concepts while enjoying the learning process. How can we optimize these MCQs to enhance engagement and knowledge retention among young learners?
2. Real-World Code Snippets Practice: The objective of the code snippet feature using the CodeGen model is to create an automated system for generating coding exercises that present users with both a programming task description and a partially implemented code snippet. The user role is then to analyze the provided instructions and complete the missing parts of the code, thus actively engaging

with the problem. This approach is designed to simulate real-world coding scenarios where developers often work with incomplete or legacy code, requiring them to fill in gaps or add necessary functionality. By leveraging the capabilities of the CodeGen model, the system aims to produce diverse, scalable, and personalized coding challenges without manual intervention, thereby enhancing learning experiences through active problem solving and practical application.

3. Debugging Detective Game: Develop an interactive game powered by the Llama-3.2-3B-Instruct model where children are presented with coding questions containing errors and tasked with identifying and correcting these mistakes. How can this hands-on debugging experience improve error detection skills and deepen understanding of programming concepts in a fun, challenge-driven format?
4. Multimodal Code and Visual Feedback Using Gemini-1.5-Flash: Introduce an innovative feature that analyzes both user-submitted code and its corresponding visual output, converting them into detailed, easy-to-understand textual feedback. How can this multimodal integration deepen learners' understanding by connecting coding logic with its visual manifestations and guiding iterative improvement?
5. Educational Chatbot Integration: Develop a conversational interface using AI to answer student questions, summarize course videos, and provide explanations suitable for children. How will integrating a chatbot enhance accessibility and comprehension of programming concepts for young learners?

1.4 Aim of Work

The proposed AI-powered Python tutoring platform aims to address the challenges faced by children aged 10 to 13 in learning programming concepts. Traditional educational methods often struggle to engage young learners effectively, leading to a gap in essential programming skills and a lack of interest in the subject. To overcome these challenges,

the platform will implement an interactive and engaging learning environment powered by artificial intelligence and gamification techniques. This approach seeks to make learning Python both enjoyable and accessible, fostering a genuine interest in programming among young students.

The platform will utilize advanced AI models, such as fine-tuned versions of Llama-3.2-3B-Instruct, to generate personalized and age-appropriate learning content. By leveraging these models, the system will provide features like a code multiple-choice questions (MCQ) game, where learners can test their knowledge through progressively challenging questions. Correct answers allow them to continue their streaks, promoting a sense of achievement and encouraging continuous engagement with the material.

In addition to the MCQ game, the platform will offer a code snippets section that presents incomplete code from real-world projects. This feature challenges students to apply their knowledge practically by completing the code, enhancing their problem-solving skills.

The implementation of the platform will utilize Flask for the backend, handling server-side operations, user data management, and integration of AI functionalities. For natural language processing tasks, such as generating hints and answering user queries, models from Hugging Face will be incorporated. The frontend will be designed with an intuitive and child-friendly interface, featuring elements like avatars, progress bars, and badges to personalize the learning experience and motivate users.

An interactive courses section will host curated video content from YouTube playlists, with an integrated Gemma 3: 4b model that summarizes videos to enhance comprehension. The model will provide explanations suited to the learners' age group.

Moreover, the platform will include an "Ask a Question" feature, empowering students to seek help whenever needed. By leveraging Gemma-2b instruct model capable of providing clear and concise answers, the system will support learners in overcoming obstacles and deepening their understanding of programming concepts.

The platform further enhances engagement through the Debugging Detective game, an interactive feature powered by the Llama-3.2-3B-Instruct model. In this game, children are presented with AI-generated coding questions that intentionally contain errors.

Learners take on the role of a detective, identifying and correcting these mistakes, which not only reinforces their understanding of programming concepts but also sharpens their debugging and critical thinking skills in a playful and motivating environment.

Complementing this is an innovative multimodal feedback system that utilizes the Gemini-1.5-flash model to analyze both the user's code and its visual output. This feature converts the combined information into clear, descriptive textual feedback that helps learners make connections between their written code and its graphical results. By bridging the gap between code logic and visual behavior, this multimodal approach deepens comprehension and guides learners through iterative improvements, making coding both more intuitive and accessible.

This comprehensive approach not only lowers the barriers to learning programming but also aligns with modern educational preferences of young learners who are accustomed to interactive and technology-driven experiences. By integrating advanced AI technologies with gamified learning methods, the proposed platform aims to provide a personalized and efficient programming education. Ultimately, the goal is to inspire and equip the next generation with essential coding skills, fostering creativity, critical thinking, and problem-solving abilities that are crucial in today's digital world.

Chapter 2: Background

2.1 Introduction

After deciding on our project to develop an AI-powered Python tutoring platform for children aged 10 to 13, we began researching Artificial Intelligence techniques that could enhance educational engagement and provide personalized learning experiences. We identified various algorithms and methodologies in Deep Learning and Natural Language Processing (NLP) that are essential for building an interactive and supportive learning environment tailored to young learners.

2.2 Artificial Intelligence

Artificial Intelligence (AI) is the simulation of human intelligence processes by machines, especially computer systems. AI involves the development of algorithms that enable computers to perform tasks that typically require human intelligence, such as learning, reasoning, and problem-solving. In the context of our project, AI is crucial for creating intelligent systems that can adapt to individual student needs, generate educational content dynamically, and provide interactive assistance throughout the learning process.

2.3 Deep Learning

Deep Learning is a subset of machine learning that utilizes neural networks with multiple layers to model complex patterns in data. These neural networks, inspired by the human brain's structure and function, can learn representations of data with multiple levels of abstraction. For our tutoring platform, deep learning techniques are vital for tasks such as:

- Generating Personalized Content: Creating customized coding challenges and exercises that match the learner's skill level.

- **Predicting Student Performance:** Analyzing student interactions to identify areas where they may need additional support.

By implementing deep learning algorithms, we aim to optimize accuracy in content generation and deliver a more effective educational tool for young programmers.

2.4 Natural Language Processing

Natural Language Processing (NLP) is a field of AI focused on the interaction between computers and human language. It enables machines to understand, interpret, and generate human language in a meaningful way. NLP is essential to our project for several reasons:

- **Educational Chatbot:** Developing an interactive chatbot that can communicate with students, answering questions and providing explanations in language appropriate for their age.
- **Hint Generation:** Offering context-sensitive hints and guidance during coding exercises without giving away the solutions outright.
- **Content Summarization:** Summarizing instructional videos and textual content to highlight key learning points and reinforce understanding.

By incorporating NLP techniques, we enhance the platform's ability to engage with learners naturally and support them effectively through their coding journey.

2.5 Language Models and Transformers

Language Models are algorithms that can predict the likelihood of a sequence of words, enabling them to generate coherent text.

Transformers are a type of neural network architecture that has revolutionized NLP by allowing models to understand context and relationships in language more effectively through a mechanism called self-attention.

In our project, we utilize language models and transformers to:

- Generate Educational Content: Using models like Llama-3.2-3B-Instruct to create multiple-choice questions that are age-appropriate.
- Provide Intelligent Assistance: Employing fine-tuned models to generate helpful hints tailored to the learner's current context.
- Enhance Natural Interaction: Enabling the chatbot to engage in more natural and meaningful conversations with students.

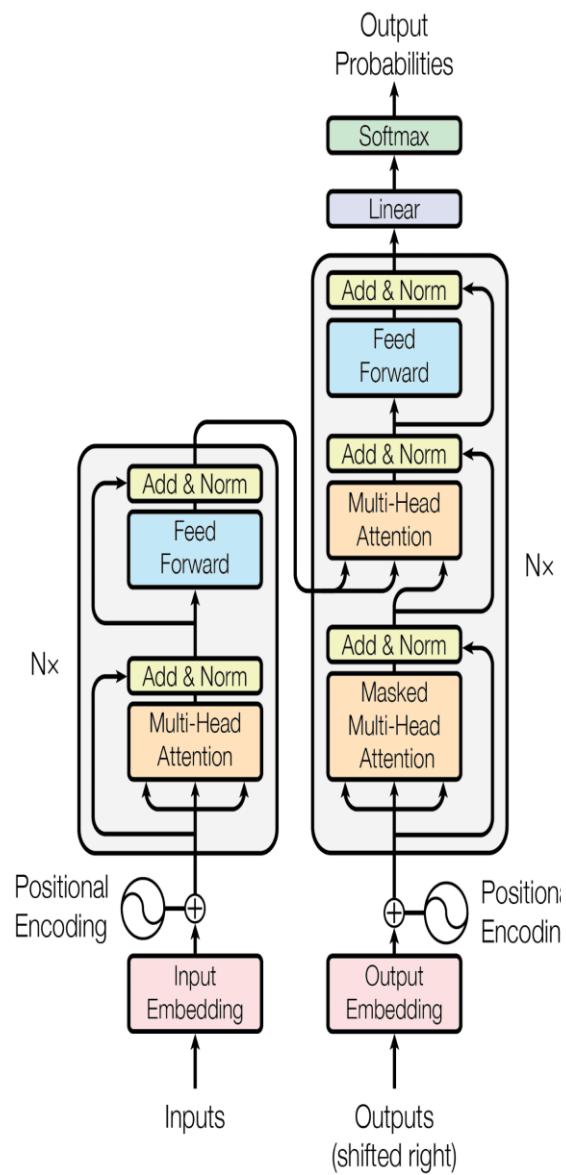


Figure 2: Transformer Architecture

By leveraging transformers, we improve platform's ability to process and generate language, making interactions more intuitive and beneficial for young learners.

2.6 Large Language Models

Large Language Models (LLMs) are advanced language models trained on extensive datasets, allowing them to generate human-like text and perform a variety of language tasks with high proficiency. Models like LLaMA are instrumental in our platform for:

- Advanced Content Generation: Creating diverse and rich educational materials that keep learners engaged.
- Adaptive Learning Pathways: Analyzing learner responses to adapt the difficulty and type of content presented.
- Comprehensive Support: Providing detailed explanations and clarifications that cater to individual learner queries.

By integrating LLMs, we enhance the intelligence and responsiveness of the platform, ensuring that each student receives a personalized and effective learning experience.

2.6.1 Salesforce/codegen-350M-mono

Salesforce/codegen-350M-mono is a member of the CodeGen family developed by Salesforce Research, designed specifically for code generation tasks. Based on a transformer decoder architecture, it leverages causal self-attention to generate syntactically coherent and contextually relevant code. This model has been trained on diverse open-source code repositories, making it proficient in understanding and producing well-structured code snippets, function definitions, and logic constructs.

The “mono” suffix indicates that it has been fine-tuned for a single programming language (commonly Python), enhancing its ability to specialize in syntax and semantics specific to that language. The model variants available include:

- **Base**: A general-purpose variant for code generation.
- **INT8**: An 8-bit quantized version optimized for lower memory consumption.

- **FP16:** A half-precision floating-point variant that accelerates inference on compatible GPUs.

This model was selected for code-focused generation tasks in this project due to its domain-specific training and integration availability through Hugging Face Transformers.

2.6.2 Gemma3: 4B

Gemma3: 4B is a large language model geared toward content generation, especially from audio-visual sources such as video transcripts. Built on a transformer architecture, it follows a dual-phase training strategy: first on general-purpose text corpora, then fine-tuned on multimedia-derived transcripts. This enables the model to excel at context-aware content creation, including summarization, scripting, and instructional writing.

The model variants include:

- **Base:** Balanced performance for general content generation.
- **Large:** Enhanced capacity for multi-topic or long-form content.
- **Lite:** A smaller version ideal for fast prototyping and constrained environments.

Gemma3: 4B was considered for multimedia-based generation due to its ability to synthesize coherent long-form content from domain-specific inputs, especially useful in educational and documentation contexts.

2.6.3 Gemma-2b Instruct

Gemma-2b Instruct is a smaller-scale instruction-tuned model within the Gemma2 family, specifically optimized for user guidance and conversation tasks. Fine-tuned on diverse instruction-following datasets—including question–answer pairs, dialogs, and task-specific prompts—this model excels in understanding user intent and generating relevant, context-aware replies.

Model variants include:

- **Instruct-chat**: Fine-tuned for maintaining coherent multi-turn conversations.
- **Instruct-lite**: Lightweight version for low-latency environments.
- **Instruct-qa**: Specialized for accurate and concise responses to direct queries.

Due to its focus on interactive instruction tasks, Gemma-2b Instruct was integrated into components requiring virtual assistant-style interactions, where prompt adherence and clarity are critical.

2.6.4 unsloth/Llama-3.2-3B-Instruct

unsloth/Llama-3.2-3B-Instruct is a 3-billion parameter instruction-tuned model derived from a LLaMA-like architecture. It is optimized for tasks that require step-by-step reasoning, including MCQ generation, debugging, and other educational or diagnostic scenarios. This model strikes a balance between computational efficiency and functional capacity, making it ideal for deployment in moderately constrained environments.

Key variants include:

- **Base**: General-purpose instruction-following model.
- **MCQ**: Fine-tuned for structured educational content generation.
- **Debug**: Targeted at code error identification and reasoning.

Its quantized versions (e.g., INT4, INT8) were used in this project to reduce resource consumption while maintaining high performance in problem-solving and task-specific reasoning.

2.6.5 gemini-1.5-flash

gemini-1.5-flash is a multimodal model that integrates both visual and textual input processing. Designed for high-speed inference, it employs transformer-based architecture

with optimized attention mechanisms. By combining image and text embeddings, it supports scenarios where visual context (e.g., screenshots, diagrams) accompanies code or textual information.

Available variants include:

- **Code-visual:** Tailored for visual-code pair interpretation (e.g., UI bug reviews).
- **Design:** Geared toward tasks involving UI mockups and design feedback.
- **Lite:** A resource-efficient version retaining key multimodal capabilities.

In this project, gemini-1.5-flash serves as a bridge between visual inputs and textual outputs, offering richer insight where traditional text-only models may fall short—particularly in debugging and interface-based feedback.

2.7 Quantization

Quantization is a technique used to reduce the size and computational requirements of machine learning models by representing their weights and activations with lower precision. Instead of using 32-bit floating-point numbers (FP32), quantized models use smaller data types like 8-bit integers (INT8) or 4-bit formats. This results in faster inference and lower memory usage, with minimal impact on model performance when applied carefully.

In this project, quantization was applied to the **Llama-3.2-3B-Instruct** and **Gemma-2b Instruct** models to make them more efficient for deployment on resource-constrained environments, such as consumer GPUs or CPUs.

Chapter 3: Methodology and Results

3.1 General Description of the Project

Interactive learning tools have become essential for engaging young learners in complex subjects such as programming. This project presents an AI-powered Python tutoring platform designed specifically for children aged 10 to 13, aiming to make coding education accessible, fun, and effective.

The platform offers a multifaceted learning experience by combining gamified exercises, real-world coding challenges, and AI-driven personalized support. Learners interact with features such as multiple-choice coding questions, code completion tasks, and debugging games that adapt to their skill level, fostering active problem-solving and critical thinking.

Advanced AI models generate age-appropriate content and provide contextual hints, explanations, and instant feedback to support the learning process. Additionally, a conversational chatbot enables students to ask questions and receive clear, child-friendly answers, making the platform more interactive and responsive.

Our implementation consists of two main parts: the user interface and chatbot integration, built with intuitive frontend tools to engage young learners; and the backend AI models, which include code generation, content summarization, and multimodal feedback systems. Together, these components create a personalized, engaging environment that motivates children to develop foundational programming skills effectively.

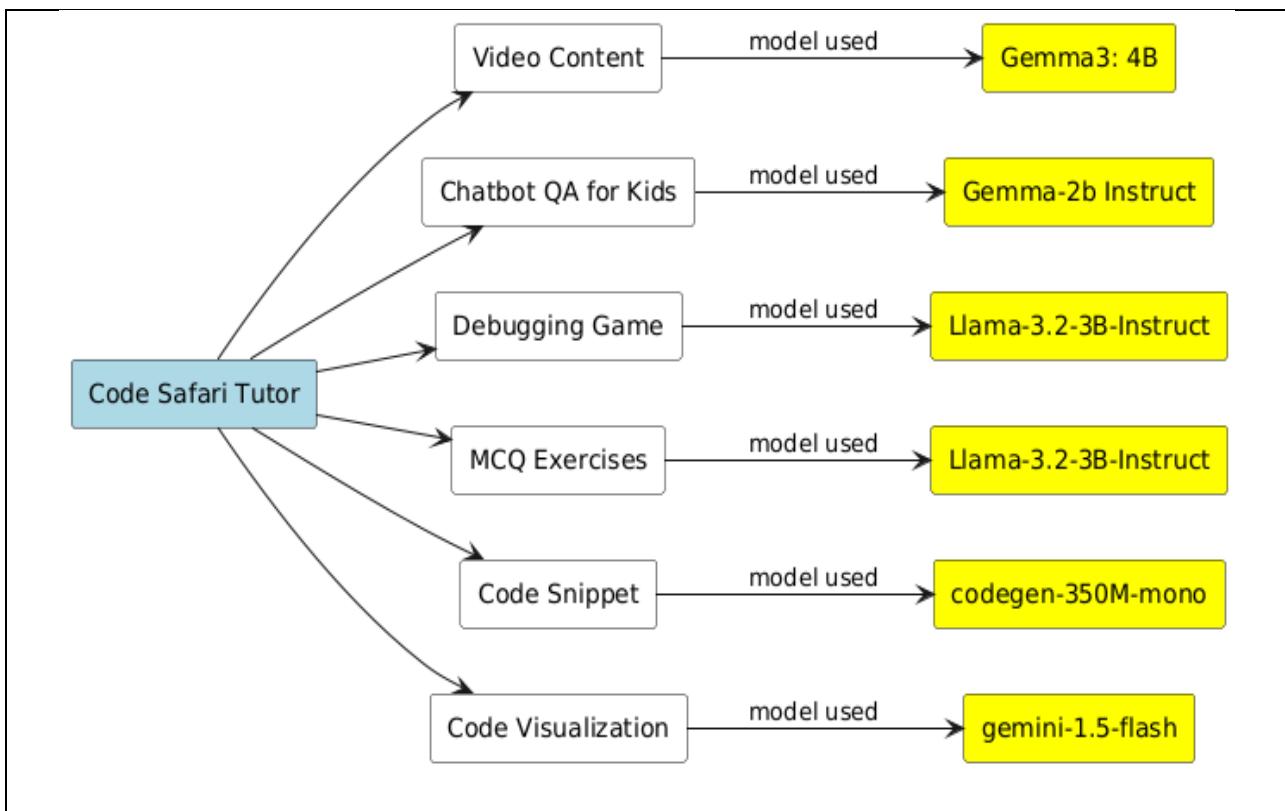


Figure 3: Project Components

3.2 Tasks and models

The project aims to enhance programming education by integrating advanced AI models that generate and personalize learning content for young learners. It employs gamified question generation to create interactive coding challenges and uses real-world code snippets to develop practical skills. Additionally, the platform includes an intelligent hint system that offers context-sensitive support and a chatbot feature to answer student inquiries with clear, age-appropriate explanations. Multimodal feedback combining code and visual outputs further aids comprehension, enabling learners to understand the relationship between their code and its results efficiently.

3.2.1 MCQ Exercises Section

To create a robust multiple-choice question (MCQ) platform, we focused on generating a diverse dataset of questions covering fundamental Python programming concepts suitable for our target age group.

3.2.1.1 Dataset

We began by importing necessary libraries for data handling, randomization, and modeling. To ensure reproducibility, we set random seeds for all random operations.

We defined a function to generate multiple-choice options for each question, ensuring that the options are plausible and challenging. Each question focuses on one of five key Python topics: mathematical operations, string manipulation, loops, conditionals, and functions.

For each topic, we created functions that generate questions by randomly selecting parameters such as numbers, strings, and operators. For instance:

- **Mathematical Operations:** Questions ask for the result of basic arithmetic operations.
- **String Manipulation:** Questions involve string methods like uppercasing or finding the length.
- **Loops:** Questions pertain to the number of iterations in `for` or `while` loops.
- **Conditionals:** Questions assess understanding of `if-else` statements and comparison operators.
- **Functions:** Questions require determining the output of simple custom functions.

We combined the questions from all topics into a single dataset and shuffled it to prevent ordering biases. The dataset was then converted into a DataFrame for ease of manipulation.

The dataset was split into training and validation sets, maintaining a 90% to 10% ratio, respectively. To handle any potential imbalance in topics or difficulty levels, we ensured that the split datasets were representative of the overall data distribution.

```
{  
    "content": "You are an AI tutor that creates multiple-choice questions (MCQs) to teach Python programming concepts to children aged 10 to 13. Your questions should be clear, engaging, and age-appropriate.",  
    "role": "system"  
},  
{  
    "content": "Generate a Python MCQ question for kids about functions.",  
    "role": "user"  
},  
{  
    "content": "<QUESTION>\ndef add(a, b):\n    return a + b\n\nwhat does add(4, 1) return?\n<OPTIONS>\n1. 7\n2. 10\n3. 5\n4. 4\n<ANSWER>\nAnswer: 3",  
    "role": "assistant"  
}
```

Figure 4: MCQ Dataset

3.2.1.2 Model selection

For our MCQ exercises section, we utilized the unsloth/Llama-3.2-3B-Instruct model, a fine-tuned large language model designed specifically for instructional tasks. With 3.2 billion parameters, this model effectively generates age-appropriate, progressively challenging Python coding questions tailored to young learners. Its transformer-based architecture enables it to maintain contextual understanding while producing high-quality, relevant educational content. The model's balance of size and performance makes it well-suited for interactive learning applications where responsiveness and adaptability are crucial. Integration through platforms like Hugging Face facilitates seamless deployment and scalability within the tutoring system.

Quantization

Although the chat variant of the Llama 3.2-3B-Instruct model is relatively smaller compared to other family members, it remains too large to run efficiently on our hardware, which lacks high-end GPUs capable of handling substantial computational

loads. To address this, we applied quantization techniques to reduce the model's memory footprint and enable smoother operation within limited VRAM constraints.

Specifically, we utilized 4-bit quantization provided by the bitsandbytes library. This method compresses the original 64-bit floating-point representation down to just 4 bits, significantly decreasing GPU memory usage without substantial loss in model performance. Compared to the standard 32-bit precision, 4-bit quantization offers faster inference speeds and more efficient memory consumption, making it ideal for resource-constrained environments.

Model Loading

To deploy the model, we first loaded the pre-trained Llama 3.2-3B-Instruct model via the Hugging Face Transformers library, followed by initializing its tokenizer. Tokenizers play a vital role in converting raw text input into numerical token representations that the model can process, and subsequently translating model-generated token outputs back into readable text. The Llama 3.2-3B-Instruct tokenizer employs a Byte Pair Encoding (BPE) scheme implemented through SentencePiece, enabling efficient and accurate tokenization aligned with the model's training.

Llama 3.2-3B-Instruct Prompt Engineering

Once the AI tutor model and tokenizer were loaded, we leveraged their capabilities through carefully crafted prompts to generate Python programming questions appropriate for children aged 10 to 13. Because our model is designed to function as an instructional question generator, effective prompting is essential to produce clear, engaging, and age-appropriate coding questions with precise answers.

```
"""
<s>[INST] <<SYS>>

{{ system Prompt }}

<</SYS>>

{{ User Prompt }}
[/INST]

{{ Model Answer }}
"""


```

Figure 5: Llama 3.2-3B-Instruct Prompt Engineering

Our prompting strategy follows a structured template consisting of two main parts: the system prompt and the user prompt.

- The **system prompt** establishes the model’s role and behavior throughout the task. For our MCQ generator, it instructs the model to act as an AI tutor who creates Python questions that are fully verifiable through code execution, strictly avoiding story or real-world scenario problems, and ensuring questions use valid Python syntax.
- The **user prompt** provides the specific task instructions and context. It includes the topic for the question generation and a list of previously generated questions, instructing the model to produce a completely new question that targets a different aspect of the topic. The prompt also enforces strict formatting rules, requiring the model to return only a JSON object with the question text and correct answer.

Within the user prompt, detailed guidelines specify the accepted question types—focusing on outputs of print statements, string operations, variable assignments, boolean expressions, and basic math—while enforcing syntactic correctness and answer formatting rules consistent with Python’s behavior.

To automate question generation, this prompt template is fed into the model repeatedly, each time requesting a unique, non-redundant question. The model outputs are parsed to extract the JSON-formatted question and answer pairs, ensuring that the questions are valid, unambiguous, and suitable for young learners.

This prompt engineering approach ensures that the MCQ generator produces high-quality, diverse coding questions that effectively support Python learning among children.

Results

The fine-tuning of our MCQ generation model was performed using the Unsloth framework, which enabled a 2x faster and more resource-efficient training process. The training ran on a single GPU with the following configuration:

```
==((=====))= Unsloth - 2x faster free finetuning | Num GPUs used = 1
    \\  /| Num examples = 20,000 | Num Epochs = 2 | Total steps = 2,500
  0^0/ \_/\ Batch size per device = 2 | Gradient accumulation steps = 8
  \       / Data Parallel GPUs = 1 | Total batch size (2 x 8 x 1) = 16
  "-____-" Trainable parameters = 48,627,712/3,000,000,000 (1.62% trained)
[2500/2500 3:30:41, Epoch 2/2]
```

- **Number of training examples:** 20,000
- **Epochs:** 2
- **Total training steps:** 2,500
- **Batch size per device:** 2
- **Gradient accumulation steps:** 8
- **Effective total batch size:** 16 ($2 \times 8 \times 1$)
- **Trainable parameters:** Approximately 48.6 million out of 3 billion total (around 1.62% of the model)

The training progressed steadily over the 2,500 steps and approximately 3.5 hours of runtime. Initial training loss started around 2.36 and decreased consistently, reaching

approximately 0.29 by the final step. This smooth decline in the loss value indicates effective learning and convergence of the fine-tuning process.

2400	0.316100
2410	0.301300
2420	0.288900
2430	0.308000
2440	0.307800
2450	0.306700
2460	0.338000
2470	0.301900
2480	0.324300
2490	0.297300
2500	0.287800

Figure 6: MCQ Training Results

Loss values at selected steps illustrate this progression:

- Step 10: 2.365
- Step 100: 1.772
- Step 500: 0.359
- Step 1000: 0.313
- Step 1500: 0.306
- Step 2000: 0.303
- Step 2500: 0.288

The significant reduction in training loss reflects the model's ability to adapt to generating coherent, syntactically correct, and age-appropriate MCQ programming questions for children.

This efficient fine-tuning process, leveraging a small subset of trainable parameters via parameter-efficient methods, allowed us to optimize the model effectively within limited hardware resources while maintaining high quality in output generation.

3.2.2 Code Visualization Section

This section focuses on analyzing users' code alongside its visual output, converting both into descriptive textual feedback.

3.2.2.1 Model selection

For the code explanation and visual feedback functionality, we employed the Gemini-1.5-Flash multimodal model. This state-of-the-art model excels at integrating both textual and visual inputs to generate coherent, context-aware natural language feedback. With its enhanced transformer architecture and 1.5 billion parameters trained on richly annotated multimodal data, Gemini-1.5-Flash is capable of interpreting user-submitted code alongside corresponding visual output (e.g., program execution snapshots, graphical renderings) to produce descriptive textual feedback that explains the code's behavior, highlights key visual cues, and offers meaningful commentary. Its strong multimodal alignment makes it ideal for educational tools that require synthesis of code and visuals into accessible, instructive language.

Model Loading

Model initialization involved loading Gemini-1.5-Flash through the Hugging Face Transformers API, alongside its multimodal tokenizer and vision processor modules. The tokenizer employs a hybrid Byte Pair Encoding (BPE) approach tailored for programming language tokens, while the vision processor converts input images or graphical visualizations into dense embeddings compatible with the model encoder. Together, these components allow seamless fusion of textual and visual inputs into unified token streams for multimodal reasoning.

Gemini-1.5-Flash Prompt Engineering

To generate insightful textual feedback that cohesively addresses both code and visuals, we designed a prompt template blending system and user instructions:

- The system prompt directs the model to act as a knowledgeable programming tutor capable of interpreting code snippets and their visual output diagrams, aiming to provide clear, constructive textual feedback that describes program functionality and visual behavior. The prompt emphasizes clarity, accuracy, and learner-friendly explanations without excessive technical jargon.
- The user prompt presents the exact code segment alongside its visual output reference, requesting the model to generate a detailed, coherent explanation that connects code logic steps to visual outcomes, highlights any notable elements, and offers suggestions or clarifications if applicable.

By iteratively feeding carefully structured multimodal prompts, the model outputs rich descriptive text that contextualizes both code and visuals effectively. This feedback aids users in understanding program effects, debugging results, and conceptual insights, enhancing their learning experience with multimodal comprehension.

Integration

We implemented an API-driven pipeline that accepts user code and associated visual artifacts, then preprocesses and tokenizes these inputs before passing them to Gemini-1.5-Flash. The model's text-based feedback outputs are parsed and rendered in the tutor interface, forming an interactive explanation layer that bridges programming logic with its visual manifestations.

3.2.3 Debugging Game

This game is designed to enhance students' debugging skills and deepen their code comprehension.

3.2.3.1 Dataset

The fine-tuning process utilized a specialized educational dataset focused on Python debugging challenges for children aged 12-15. The dataset exhibits the following characteristics:

```
{
    "content": "You are an AI tutor that creates debugging challenges to teach Python programming concepts to children aged 12-15. Your questions should be clear, engaging, and age-appropriate.",
    "role": "system"
},
{
    "content": "Generate a Python debugging challenge for kids about lists at hard level.",
    "role": "user"
},
{
    "content": "<ORIGINAL_CODE>\nmatrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]\nfor row in matrix:\n    for num in row:\n        print(num, end=' ')\n    print()\n</ORIGINAL_CODE>\n\n<BUGGY_CODE>\nmatrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]\nfor i in range(len(matrix)):\n    for j in range(len(matrix[i])):\n        print(matrix[i][j], end=' ')\n    print()\n</BUGGY_CODE>\n\n<QUESTION>\nThis code tries to print a 3x3 matrix but throws an index error. What's wrong?\n</QUESTION>\n<OPTIONS>\n1. Using same range for the inner loop, should be range(len(matrix[i]))\n2. The matrix is not properly initialized\n3. print() function is used incorrectly\n4. The loops should be reversed\n</OPTIONS>\n<ANSWER>\nAnswer: 1\n</ANSWER>\n\n<EXPLANATION>\nThe inner loop uses range(len(matrix)), assuming all rows have the same length as the entire matrix. It should use range(len(matrix[i])) to get the correct length of each row.\n</EXPLANATION>\n\n<DIFFICULTY>\nHard\n</DIFFICULTY>\n\n<CATEGORY>\nLists\n</CATEGORY>",
    "role": "assistant"
}
```

Figure 7: Debugging Game Dataset

- **Structure:** Implemented as a series of conversational exchanges in a standardized format consisting of system messages, user queries, and assistant responses.
- **Content Type:** Each example contains comprehensive debugging scenarios with original code, intentionally buggy implementations, graduated hints, solutions, and explanatory content.
- **Domain Coverage:** The dataset encompasses fundamental Python programming concepts including conditional logic, iteration structures, data structures (lists, dictionaries), and function implementations.

- **Format:** Data is structured in JSON format with conversations organized as arrays of role-content pairs, following standard conversational AI training practices.
- **Size:** The dataset comprises approximately 20,000 examples, providing substantial training material to establish domain expertise while remaining manageable for efficient fine-tuning iterations.

3.2.3.2 Model selection

The implementation employs the Llama-3.2-3B-Instruct model from Unislot. This selection was determined by several technical considerations:

Technical Specifications

- **Architecture:** 3 billion parameter transformer-based language model from the Llama family
- **Quantization:** Implemented with 4-bit quantization to optimize memory utilization
- **Fine-tuning Approach:** Low-Rank Adaptation (LoRA) with rank=32
- **Target Parameters:** Modified key attention components (q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj)

Selection Criteria

- **Computational Efficiency:** The 3B parameter model represents an optimal compromise between performance capabilities and resource requirements, making it suitable for deployment in constrained computational environments.
- **Instruction Optimization:** Pre-existing instruction tuning provides foundational capabilities for following structured directives, reducing the adaptation required for task-specific fine-tuning.

- **Memory Utilization:** 4-bit quantization significantly reduces VRAM requirements, enabling training on consumer-grade hardware or cloud-based environments with limited resources.
- **Training Efficiency:** LoRA implementation allows for efficient parameter updating (1-10% of total parameters), substantially reducing computational demands compared to full-model fine-tuning.
- **Performance Optimization:** Unsloth library integration provides approximately 2x improvement in both training and inference speeds, with additional memory optimizations reducing VRAM consumption by approximately 30%.
- **Conversation Format Compatibility:** Native support for the Llama-3.1 chat template enables seamless handling of multi-turn educational interactions.

Alternative Models Considered

Several alternative models were available but not selected due to specific limitations:

- **Larger Llama Variants (8B, 70B, 405B):** While offering potentially superior performance, these models introduce prohibitive computational requirements that would limit practical deployment.
- **Smaller Models (Llama-3.2-1B):** May provide insufficient reasoning capacity for generating comprehensive educational content with explanatory depth.
- **Base Models Without Instruction Tuning:** Would necessitate more extensive fine-tuning to achieve comparable instructional capabilities.
- **Alternative Model Families:** Models from Mistral, Phi-3, and Gemma architectures were available but presumably offered less favorable performance-efficiency trade-offs for this specific application context.

The selected model configuration demonstrates an approach that balances performance requirements with practical deployment considerations for educational AI applications.

Model Deployment Procedure

The implementation of the Debugging Detective educational game required a systematic approach to model initialization, fine-tuning, and deployment. The following methodology was employed to ensure optimal performance while maintaining computational efficiency.

Base Model Initialization

The deployment process commenced with the initialization of the Llama-3.2-3B-Instruct foundation model utilizing the Unsloth optimization framework. The following configuration parameters were implemented:

- **Maximum Sequence Length:** 2048 tokens to accommodate detailed programming challenges
- **Precision Handling:** Automatic detection of optimal precision based on available hardware
- **Memory Optimization:** 4-bit quantization to reduce memory requirements by approximately 75% compared to standard 16-bit implementations

The initialization process leveraged FastLanguageModel's `from_pretrained` method with optimized loading procedures that mitigate out-of-memory issues while maintaining model integrity.

Parameter-Efficient Fine-tuning Configuration

Following initialization, the model was configured for Parameter-Efficient Fine-Tuning (PEFT) using Low-Rank Adaptation with the following specifications:

```
Unsloth - 2x faster free finetuning | Num GPUs used = 1
Num examples = 63 | Num Epochs = 13 | Total steps = 100
Batch size per device = 2 | Gradient accumulation steps = 4
Data Parallel GPUs = 1 | Total batch size (2 x 4 x 1) = 8
Trainable parameters = 48,627,712/3,000,000,000 (1.62% trained)
```

- **Adaptation Rank:** 32, selected to balance parameter update capacity with computational efficiency
- **Target Module Selection:** Seven critical transformer components were targeted for adaptation: query projection, key projection, value projection, output projection, gate projection, up-projection, and down-projection matrices
- **Alpha Parameter:** Set to 16 to regulate adaptation magnitude
- **Memory Optimization:** Implementation of "unsloth" gradient checkpointing to reduce VRAM requirements by approximately 30%
- **Dropout Configuration:** Set to 0 for maximum computational efficiency
- **Bias Treatment:** Configured as "none" to optimize computational requirements

This configuration ensured that only 1-10% of the model's parameters required updating, substantially reducing the computational demands of the fine-tuning process.

Model Persistence Protocol

Upon completion of the fine-tuning process, both the adapted model parameters and tokenizer configuration were preserved through a standardized persistence protocol. This ensured that the specialized knowledge acquired during fine-tuning was retained for subsequent deployment without requiring retraining.

Inference Optimization

For deployment in the game environment, the persisted model was loaded with additional inference optimizations:

- **Inference Mode Activation:** The model was explicitly configured for inference using FastLanguageModel's inference optimization routines
- **Quantization Maintenance:** 4-bit quantization was preserved to maintain minimal memory footprint

- **Generation Parameter Optimization:** Text generation parameters were calibrated for the educational context:
 - Temperature: 0.7 to balance creativity with accuracy
 - Top-p (nucleus sampling): 0.9 to ensure appropriate diversity in generated content
 - Maximum new tokens: 512 to accommodate comprehensive explanations

3.2.4 Code Snippet

Traditional programming education frequently relies on fixed sets of problems that lack variation and fail to adapt to individual learner progress, which can result in disengagement and limited skill development. Many platforms either present fully written code for analysis or require solutions from scratch, which does not fully reflect common development workflows where partial code completion is essential. Additionally, manual creation and curation of programming exercises demand significant effort from educators and often limit the availability of fresh, relevant tasks. Furthermore, conventional evaluation methods that check user solutions primarily through exact output matching can overlook deeper logical correctness and understanding, leading to inadequate assessment of a learner's true capability. These challenges underscore the need for an automated, adaptive, and interactive coding exercise generation and evaluation system.

The primary goals of integrating the CodeGen model in this code snippet is to promote active learning by engaging users in completing partial code, to introduce variability and adaptability in coding challenges, and to automate the generation process for scalability. By presenting exercises with incomplete code, learners are encouraged to think critically, understand underlying logic, and apply programming concepts more effectively. This approach also helps replicate realistic coding scenarios that developers face in practice. Moreover, the system aims to dynamically generate a wide range of problems targeting different skill levels and programming topics, reducing repetition and boredom while

addressing individual learning gaps. Finally, by enabling automated, dynamic evaluation of user submissions, the system provides immediate and meaningful feedback, fostering a more efficient and personalized learning experience that better prepares users for real-world programming tasks.

3.2.4.1 Dataset

Instruction	Input	Output
Programming question	Input for the question	Output code for the question
13077 unique values	[null] [1, 2, 3, 4, 5] Other (7477)	42% 1% 57%
Write a function to find squares of individual elements in a list using lambda function.		<pre>def square_nums(nums): square_nums = list(map(lambda x: x ** 2, nums)) return square_nums</pre>
Create a function in Python which takes a list of numbers and returns the sum of squares of those nu...	[3, 4, 5]	<pre>def sum_squares(nums): result = 0 for num in nums: result += num*num return...</pre>
Design a function in Python that sorts a	['red', 'green', 'orange']	<pre>def sort_list_alphabetic ...</pre>

Figure 8: Code Snippet Dataset

We used this dataset called “Python Programming Language” from Kaggle to fine-tune the CodeGen language model for the purpose of generating Python code snippets. This fine-tuning process aimed to enhance the model's ability to act as a code generation, particularly suited for educational and practical Python programming tasks.

The dataset is structured to support instruction-based code generation, where each data sample contains:

- **Instruction:** A natural language description of a coding task (e.g., "Write a function to calculate the factorial of a number").
- **Input:** Optional context or partial code that can assist the model in understanding the task.
- **Output:** The corresponding Python code snippet that fulfills the instruction.

3.2.4.2 Model selection

The model that we choose is the CodeGen 350M as it is the best choice. Among the available model sizes in the CodeGen like 350M, 2B, 6B, and 16B, the 350M version offers a highly favorable balance between performance, ease of use, resource requirements, and practicality for focused tasks. This section outlines the rationale behind selecting this model over larger variants.

CodeGen 350M offers fast training and inference speeds, which makes it exceptionally well-suited for interactive applications for our education website. For tasks where fast response time is essential—like generating coding problems on-the-fly or responding to user requests in a learning interface—a smaller model like 350M ensures smooth and efficient performance. The lightweight nature of this model also means that training iterations can be completed more quickly, allowing for faster experimentation and refinement of results. One of the advantages is the low resource requirement. CodeGen 350M can be easily trained or deployed on environments like Google Colab Free or Pro tiers, standard desktop GPUs, or even CPU-based systems for limited use. This makes it highly accessible for individual developers, researchers, and educational platforms that do not have access to high-performance computing resources. Unlike its larger counterparts, it does not require a multi-GPU setup or expensive cloud infrastructure.

When it comes to ease of fine-tuning, the 350M model is best as well. Due to its small size, it supports a variety of fine-tuning techniques—including full fine-tuning or parameter-efficient methods like LoRA (Low-Rank Adaptation) and PEFT (Parameter-Efficient Fine-Tuning)—without exhausting memory or compute resources. This allows

for quicker iterations, easier debugging, and more agile model development, especially when the training dataset is evolving or being cleaned over time. Also, using smaller models helps to avoid unwanted behaviors, such as generating too much code or solving the problem entirely, when the goal is to stop at a partial implementation. Larger models like CodeGen 6B or 16B can be harder to control and may overgenerate, producing completed solutions rather than the intended instructional scaffolding.

On the other hand, the larger models in the CodeGen 2B, 6B, and 16B introduce challenges that are not justified by the potential gain in quality for this specific use case. For example in CodeGen 2B while it is more capable than 350M, it requires more VRAM for at least 16 GB, and is slower to train and serve, and has minimal advantage in generating templated instructional content. CodeGen 6B and 16B, though significantly more powerful in terms of generalization and code quality, require expensive infrastructure such as A100 GPUs or distributed training systems. They are best reserved for tasks that demand high-level reasoning, full-code solutions, or multi-step program generation—none of which are central to the current goal.

Model Loading & Deployment

A pretrained language model called **Salesforce/codegen-350M-mono** is first loaded using Hugging Face’s AutoTokenizer and AutoModelForCausalLM. This model is designed to generate code and serves as the base model for fine-tuning. After preparing and tokenizing a dataset of Python programming instructions and solutions, the model is trained using this custom data. Once training is complete, the fine-tuned model and tokenizer are saved locally to a directory named `./codegen-finetuned`. Later, for practical use for our generating new coding questions and evaluating user-submitted solutions, the fine-tuned model is loaded again—this time from the Hugging Face Hub under the name `doaa18/codegen`. This step allows the application to use the trained model without retraining it, making it efficient and reusable for deployment or further testing.

Model Result

```
TrainOutput(global_step=3277, training_loss=0.7528896339062623, metrics={'train_runtime': 2289.1849, 'train_samples_per_second': 5.726, 'train_steps_per_second': 1.432, 'total_flos': 1.2252825778126848e+16, 'train_loss': 0.7528896339062623, 'epoch': 0.9998474446987032})
```

Figure 9: Code Snippet training results

The fine-tuning process was conducted using the Salesforce/codegen-350M-mono model on a dataset of Python programming problems, with prompts consisting of an instruction, optional input, and expected solution. The training was performed for one epoch using the Hugging Face Trainer API. The final output showed a training loss of approximately 0.75, which indicates that the model was able to learn meaningful patterns from the data and generate plausible code completions. The training ran for approximately 2,289 seconds, processing about 5.7 samples per second and completing a total of 3,277 steps.

Model Prompt



Coding Exercise

Challenge yourself with coding problems

Problem

 **Model-Based Code Exercise System**

 **How It Works:**

- **Model Comparison:** Your solution is compared directly with an AI-generated model solution
- **Sample Testing:** Tests your code with various inputs against the model
- **Smart Feedback:** Shows exactly where your output differs from expected

 Your solution just needs to produce the same results as the AI model - multiple approaches accepted!

 Generating new problem...

 **Your Solution**

Write your solution here...

[Generate New Problem](#) [Check My Solution](#)

At the top, it says “Coding Exercise” with a little brain emoji, which gives the whole thing a fun and encouraging feel. This is a feature in a sidebar in our website to test and improve their coding skills by solving problems directly in their browser.

In the light blue box, there’s a description of how the system works. It basically tells the kid that when they write the code to solve a task, the system will compare their solution to one created by an AI model. The goal isn’t necessarily to write the same code, but to make sure the output matches what the model produces. It also says that the user’s code will be tested with multiple examples, and if their output differs from what’s expected, it’ll show you exactly where things went wrong. That makes it easier to learn from your mistakes and fix your code.

Below that, the system is in the middle of creating a new problem to solve, which is why it says “Generating new problem...”. Underneath that, there’s a big text box labeled “Your Solution” where the kid will eventually write their code once the problem is ready.

At the very bottom, there are two buttons. One is for getting a new problem if they want to try multiple Python tasks, and the other is for checking their solution to see if it’s correct. When the kid click the “Check My Solution” button, the system will run their code and let you know how it did.

3.2.5 Video Content

Primary Video Source: "Hello Python 3 Programming for Kids & Beginners"

After extensive evaluation of available Python programming courses suitable for children, we selected the "Hello Python 3 Programming for Kids & Beginners" course as our primary video content source. This decision was based on several critical factors that align with our educational objectives and target audience requirements.

Course Characteristics and Advantages:

- Age-Appropriate Content Design: The course specifically targets young learners and beginners, employing simplified explanations and visual aids that facilitate comprehension for children aged 10-13
- Progressive Learning Structure: Content is organized in a logical sequence, building from fundamental programming concepts to more complex applications
- Interactive Elements: The course incorporates hands-on coding exercises and practical examples that engage young learners effectively
- Clear Communication Style: Instructors use accessible language and avoid overly technical jargon while maintaining accuracy
- Practical Application Focus: Emphasis on real-world programming applications that children can relate to and find motivating

Content Alignment with Educational Standards

The selected video content aligns with established computer science education standards for K-12 learners, including concepts from the Computer Science Teachers Association (CSTA) guidelines. The curriculum covers essential programming fundamentals such as:

- Variables and data types
- Control structures (loops and conditionals)
- Functions and modular programming
- Basic algorithm design
- Problem-solving methodologies
- Debugging techniques

3.2.5.1 Dataset

The foundation of our automated study materials generation system relies on the carefully selected "open-phi/programming_books_llama" dataset, which represents a comprehensive collection of programming literature specifically optimized for educational applications.

Dataset Composition and Quality Metrics:

- **Total Token Count:** 400 million tokens of programming-focused educational content
- **Dual-Source Architecture:** The dataset combines two distinct generation methodologies to ensure comprehensive coverage and quality

Primary Source - GPT-3.5 Generated Content (70M tokens):

- Represents the highest quality segment of our dataset
- Generated using advanced prompting techniques to ensure educational relevance
- Maintains consistent formatting and pedagogical structure
- Emphasizes clarity and accessibility for learners at various skill levels
- Incorporates best practices in programming education methodology

Secondary Source - Fine-tuned CodeLlama 34B Content:

- Provides additional breadth to the dataset with programming-specific expertise
- While shorter in length compared to GPT-3.5 content, maintains good quality standards
- Contributes specialized programming knowledge and code examples
- Enhances the diversity of explanatory approaches and learning styles addressed

Quality Assurance and Validation:

The dataset underwent rigorous quality assessment using the "textbook quality repo" methodology, ensuring that all included content meets educational standards appropriate for young learners. This validation process included:

- Content accuracy verification
- Age-appropriateness assessment
- Pedagogical effectiveness evaluation
- Consistency in terminology and explanations

3.2.5.2 Model selection

Base Model Selection: Google Gemma-3-4B-IT We selected Google's Gemma-3-4B-IT as our base model due to its optimal balance of performance and computational efficiency. This model offers several advantages for our educational application:

- Instruction-following capabilities essential for generating structured educational content
- Manageable computational requirements suitable for real-time generation
- Strong performance on text generation tasks relevant to educational content creation
- Established reliability and safety features appropriate for children's education platforms

Fine-Tuning Methodology: The fine-tuning process involved training the base Gemma model on our curated programming_books_llama dataset, resulting in a specialized model optimized for generating Python programming educational content. The fine-tuning process incorporated:

- **Domain Adaptation:** Specializing the model's knowledge base for Python programming education

- **Style Consistency:** Ensuring generated content maintains appropriate tone and complexity for young learners
- **Structural Optimization:** Training the model to produce well-organized study materials with clear hierarchical structure
- **Example Integration:** Enhancing the model's ability to provide relevant code examples and practical applications

Automated Content Generation Pipeline

Video Transcription System

Our content generation pipeline begins with the automated transcription of selected video content using the YouTube Transcript API. This system provides several key advantages:

Technical Implementation:

- Automated extraction of video transcripts from YouTube content
- Support for multiple languages and subtitle formats
- Real-time processing capabilities for immediate content generation
- Error handling and quality assurance for transcript accuracy

Content Processing Features:

- Timestamp preservation for content synchronization
- Speaker identification where available
- Automatic formatting and punctuation correction
- Noise reduction and clarity enhancement for improved processing

AI-Generated Study Materials

The fine-tuned Gemma model processes video transcriptions to generate comprehensive study materials tailored to each video's content. This automated system produces several types of educational resources:

Study Material Components:

- **Concept Summaries:** Concise explanations of key programming concepts covered in each video
- **Learning Objectives:** Clear statements of what students should understand after engaging with the content
- **Key Terminology:** Definitions of important programming terms and concepts
- **Code Examples:** Practical programming examples that reinforce video content
- **Practice Exercises:** Hands-on activities designed to test understanding and application
- **Common Mistakes:** Identification of typical errors and how to avoid them

Content Adaptation Features:

- **Reading Level Optimization:** Content automatically adjusted for age-appropriate reading levels
- **Visual Learning Support:** Integration of code snippets and formatting that enhances comprehension
- **Progressive Complexity:** Materials structured to build upon previous knowledge systematically
- **Cultural Sensitivity:** Content reviewed to ensure inclusivity and accessibility for diverse learners

Quick Reference Guide Generation

In addition to comprehensive study materials, our system generates quick reference guides that serve as valuable learning aids for students during their programming practice sessions.

Reference Guide Features:

- **Syntax Quick Cards:** Essential Python syntax presented in easy-to-scan format

- **Function Libraries:** Commonly used functions with parameters and examples
- **Error Solutions:** Quick fixes for frequently encountered programming errors
- **Best Practices:** Concise guidelines for writing clean, efficient code
- **Troubleshooting Tips:** Step-by-step problem-solving approaches

Integration and User Experience

Seamless Content Delivery

The integration of video content with AI-generated study materials creates a cohesive learning experience where students can:

- Access synchronized study materials while watching videos
- Review key concepts through generated summaries
- Practice skills using automatically created exercises
- Reference quick guides during hands-on coding sessions

Personalization and Adaptability

Our system's AI-driven approach enables personalized learning experiences by:

- Generating content that adapts to different learning styles
- Providing multiple explanation approaches for complex concepts
- Adjusting content complexity based on student progress
- Offering diverse practice exercises to reinforce learning

Technical Implementation Considerations

System Architecture

The technical infrastructure supporting our content generation system includes:

- **Robust API Integration:** Reliable connection to YouTube Transcript API for video processing

- **Scalable Computing Resources:** Cloud-based infrastructure to handle varying content generation demands
- **Data Storage Solutions:** Efficient storage and retrieval systems for generated content and user data
- **Security Measures:** Comprehensive data protection and privacy safeguards appropriate for educational platforms serving children

Performance Optimization

To ensure optimal user experience, our system incorporates:

- **Caching Mechanisms:** Efficient storage of frequently accessed content to reduce load times
- **Load Balancing:** Distribution of computational tasks to maintain consistent performance
- **Quality Monitoring:** Continuous assessment of generated content quality and system performance
- **Error Handling:** Robust error management to ensure system reliability and user satisfaction

3.2.6 Chatbot Text-to-Text QA for Kids

This section describes the implementation of a chatbot designed to provide clear, age-appropriate answers to Python programming questions for children aged 10 to 13. The chatbot leverages a fine-tuned large language model (LLM) with Retrieval-Augmented Generation (RAG) to deliver accurate, child-friendly explanations, making it an effective tool for fostering programming education.

3.2.6.1 Dataset

The chatbot's training and evaluation utilized a custom dataset tailored for Python programming education, specifically designed to align with the cognitive and learning

needs of young learners. The dataset exhibits the following characteristics: it is structured as a CSV file with two columns (Question and Child friendly Answer), where each row represents a programming question, its standard technical answer, and a simplified, child-friendly explanation using metaphors and emojis to enhance engagement. The content type covers foundational Python concepts, including variables, functions, strings, loops, conditionals, and object-oriented programming, with answers crafted to be both technically accurate and accessible, using child-friendly versions with relatable analogies such as variables as "magical treasure chests 🏆" and functions as "superhero moves 🦸". The dataset spans a wide range of beginner-to-intermediate Python topics, ensuring comprehensive coverage of concepts relevant to young learners, including examples of differentiating between tuples and lists, explaining recursion, and describing context managers. The CSV format facilitates easy parsing and integration into the training pipeline, with each child-friendly answer adhering to a consistent tone, incorporating playful language and visual cues like emojis to maintain engagement. The dataset contains approximately 1,000 question-answer pairs, providing sufficient diversity for fine-tuning while remaining manageable for efficient training on limited hardware. The dataset was preprocessed to ensure consistency, removing any incomplete or invalid entries, and to support RAG, questions and answers were embedded into a vector store using a sentence transformer model, enabling efficient retrieval of relevant context during inference.

Index	Question	Answer
0	What is the difference between a variable and an object?	, here's the difference between a magical treasure chest 🎁 that holds things and an object: "a magical treasure chest 🎁 that holds things is a named memory location that stores a single value.", "it is a placeholder for a specific amount of data.", "a magical treasure chest 🎁 that holds things can hold different values throughout the program.", "they are declared using the '=' operator and assigned a value.", "object: "an object is a complex data structure that contains data and methods.", "it is an instance of a class.", "objects can have multiple magical treasure chest 🎁 that holds things and methods associated with them.", "they are created using the 'new' keyword and contain a copy of all the magical treasure chest 🎁 that holds things and methods of the class.", "in summary, a magical treasure chest 🎁 that holds things is a single piece of memory that stores a single value, while an object is a complex data structure that contains data and methods.]
1	What is the difference between a built-in function and an imported function?	. here's the difference between built-in superhero move 🚶 that does something special and imported a superhero move 🚶 that does something special: "built-in superhero move 🚶 that does something special: "built-in superhero move 🚶 that does something special are superhero move 🚶 that does something special that come built with the operating system or programming language itself.", "they have a specific syntax and are not available in other parts of the program.", "examples of built-in superhero move 🚶 that does something special include the 'megaphone 🔈 that announces things on the screen()', 'len()', and 'sum()' superhero move 🚶 that does something special in python.", "imported a superhero move 🚶 that does something special: "imported a superhero move 🚶 that does something special are superhero move 🚶 that does something special that are defined in other files or modules.", "they can be called from anywhere in the program, regardless of its location.", "imported a superhero move 🚶 that does something special can be defined using the 'import' command 🛡 that tells Python what to do in python, for example, "key difference: "imported a superhero move 🚶 that does something special are available throughout a program and cannot be changed.", "imported a superhero move 🚶 that does something special can be customized or overridden, depending on the module they're defined in.", "benefits of using built-in superhero move 🚶 that does something special: "built-in superhero move 🚶 that does something special are generally well-tested and efficient.", "they are tested and guaranteed to work correctly.", "they can be used directly without any need for external modules.", "benefits of using imported a superhero move 🚶 that does something special: "imported a superhero move 🚶 that does something special can be customized to meet specific program requirements.", "they can be used from any part of the program, regardless of location.", "they allow for modularity and easier code organization.]
2	What is the difference between the 'print' function	. here's the difference between the two superhero move 🚶 that does something special: "a megaphone 🔈 that announces things on the screen" a superhero move 🚶 that does something special a megaphone 🔈 that announces things on the screens its arguments to the terminal window or other specified destination.", "the default value of the 'a megaphone 🔈 that announces things on the screen' a superhero move 🚶 that does something special is the terminal window.", "the 'a megaphone 🔈 that announces things on the screen' multiple a magical treasure chest 🎁 that holds things or objects, separated by commas.", "a megaphone 🔈 that announces things on the screen" a superhero move 🚶 that does something special with keyword arguments: "the 'a megaphone 🔈 that announces things on the screen' a superhero move 🚶 that does something special with keyword arguments a megaphone 🔈 that announces things on the screens its arguments to the terminal window or other specified destination with a colon separating each argument.", "the 'a megaphone 🔈 that announces things on the screen' a superhero move 🚶 that does something special with keyword arguments a megaphone 🔈 that announces things on the screen only one a magical treasure chest 🎁 that holds things or object".

Figure 10: Chatbot QA Dataset

3.2.6.2 Knowledge Base Integration

To enhance the chatbot's ability to provide accurate and educational responses, a knowledge base was constructed by integrating authoritative programming textbooks. Specifically, Lutz, M. (2010). Programming Python (4th ed.). O'Reilly Media and Learning Python by Mark Lutz were incorporated. These resources were selected for their clear, practical examples and beginner-friendly explanations, which align with the chatbot's goal of facilitating Python education for children. Relevant sections from these texts were extracted, processed, and embedded into the vector store alongside the dataset, ensuring that the chatbot could draw upon both structured Q&A data and broader conceptual knowledge during response generation.

3.2.6.3 Model Selection

The chatbot implementation employs Google's gemma-2b-bit model, a 2-billion-parameter transformer-based LLM optimized for instructional tasks. This model was chosen based on several technical considerations, including its transformer-based architecture with 2 billion parameters pre-trained for instruction-following tasks, utilization of 4-bit

quantization via the BitsAndBytes library to optimize memory usage, Low-Rank Adaptation (LoRA) with a rank of 8 for efficient parameter updates, and adaptation of key attention components (q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj) to optimize performance for educational Q&A. The selection criteria focused on computational efficiency, as the 2B parameter size balances performance and resource requirements, making it suitable for deployment on hardware with limited GPU capabilities. The pre-trained instruction-following capabilities reduce the need for extensive fine-tuning, enabling quick adaptation to child-friendly Q&A tasks, while 4-bit quantization reduces VRAM usage by approximately 75% compared to 16-bit precision, allowing smooth operation on consumer-grade hardware. LoRA fine-tuning updates only 1–10% of parameters, minimizing computational demands while maintaining high performance, and the model integrates seamlessly with RAG pipelines, leveraging retrieved context to enhance response accuracy and relevance. Alternative models were considered but rejected: larger models such as Gemma-7B and Llama-3.2-8B were rejected due to high computational and memory requirements unsuitable for hardware constraints, smaller models like DistilBERT and Llama-3.2-1B lacked sufficient reasoning capacity for generating nuanced, child-friendly explanations, and non-instruction-tuned models required more extensive fine-tuning, increasing development time and resource usage. The Gemma-2b-it model, combined with RAG, provides an optimal balance of performance, efficiency, and adaptability for the chatbot's educational objectives.

3.2.6.4 Quantization

To ensure efficient deployment on resource-constrained hardware, 4-bit quantization was applied using the BitsAndBytes library. This approach compresses the model's 64-bit floating-point weights to 4-bit representations, reducing memory usage by approximately 75% while preserving performance. The quantization configuration included nf4 (4-bit NormalFloat) quantization type optimized for transformer models, and torch.bfloat16 compute data type, enabling faster inference on compatible hardware with minimal

accuracy loss. This quantization enabled the model to run efficiently within limited VRAM constraints, ensuring responsiveness for real-time interactions in the educational platform.

3.2.6.5 Model Loading

The Gemma-2B-IT model was loaded using the Hugging Face Transformers library through several key steps. The tokenizer was initialized using AutoTokenizer.from_pretrained, configured with the model's chat template to handle conversational inputs and outputs, employing a hybrid Byte Pair Encoding (BPE) scheme that ensures efficient tokenization for both questions and responses. The model was initialized using AutoModelForCausalLM.from_pretrained with the 4-bit quantization configuration (bnbConfig) and device_map="auto" for automatic device placement (CPU or GPU), ensuring compatibility with the hardware environment and optimized memory usage.

3.2.6.5.1 Prompt Engineering

To ensure the chatbot delivers clear, engaging, and age-appropriate responses, a structured prompt template was designed. The system prompt defines the model's role as a friendly Python tutor, emphasizing simple language, metaphors such as "variables are like treasure chests 🏆", and emojis like 😊 and 🎉 to enhance engagement, while instructing the model to avoid complex jargon and focus on verifiable, code-based explanations. The user prompt combines the child's question, retrieved context from the RAG pipeline, and instructions to generate a response that is concise, accurate, and tailored to the 10–13 age group, enforcing a consistent tone and format to ensure responses are both educational and fun. The prompt template ensures that the chatbot produces responses that are educational, engaging, and tailored to the cognitive level of young learners, such as explaining variables as "named boxes that store information" when asked "What is a variable in Python?", using analogies suitable for children. The

model was fine-tuned using the Supervised Fine-Tuning (SFT) framework, utilizing the SFTTrainer from the Transformers library optimized for supervised fine-tuning tasks.

3.2.6.5.1 Fine-tuning Configuration

Fine-tuning was performed using LoRA with an adaptation rank of 8 to balance expressivity and efficiency, targeting modules including q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, and down_proj for task-specific adaptation, with an alpha parameter set to 32 to control adaptation magnitude. The training setup included a batch size per device of 1 suitable for limited GPU memory, gradient accumulation steps of 4 yielding an effective batch size of 4, warmup steps of 2 for gradual learning rate increase, max steps of 50 sufficient for fine-tuning on the dataset, learning rate of 2e-4 as a standard rate for fine-tuning transformer models, FP16 mixed precision for faster training with minimal accuracy loss, paged_adamw_8bit optimizer optimized for memory-efficient training, and output directory saved to outputs for model checkpoints and logs. The fine-tuning process was performed on a single GPU, utilizing a dataset of 80 training steps with total training time of approximately 769 seconds (~12.8 minutes), and the following loss progression was observed during training:

Step 1: 1.0472

Step 20: 0.4137

Step 40: 0.3417

Step 60: 0.3367

Step 80: 0.2203

The average training loss was approximately 0.4126, showing a strong and consistent downward trend from the initial value, indicating successful learning and convergence.

3.2.6.6 Results

The fine-tuning process was conducted using the transformers library with LoRA, achieving efficient training on a single GPU. Key training details included approximately 900 question-answer pairs for training examples, 2 epochs, 80 total training steps, batch size per device of 1, and gradient accumulation steps of 4 resulting in an effective batch size of 16. The training loss decreased steadily, indicating effective learning, progressing from Step 1 at 1.0472, Step 20 at 0.4137, Step 40 at 0.3417, Step 60 at 0.3367, to Step 80 at 0.2203. Training metrics demonstrated approximately 769 seconds (~12.82 minutes) of training time, around 1–10% of total model parameters as trainable parameters adapted using LoRA for parameter-efficient fine-tuning, final training loss of 0.2203 with a steady decline from 1.0472 (Step 1) to 0.2203 (Step 80) reflecting successful convergence and task adaptation, and total FLOPs of 4.48 quadrillion ($\approx 4,482$ trillion FLOPs), indicating efficient yet substantial computation during training.



```
TrainOutput(global_step=80, training_loss=0.41259115319699047, metrics={'train_runtime': 768.9042, 'train_samples_per_second': 0.832, 'train_steps_per_second': 0.104, 'total_flos': 4482151500042240.0, 'train_loss': 0.41259115319699047})
```

Figure 11: Chatbot QA Training Results

3.2.6.7 Retrieval-Augmented Generation (RAG)

To enhance the chatbot's ability to provide accurate and contextually relevant responses, Retrieval-Augmented Generation (RAG) was integrated. RAG combines retrieval-based mechanisms with generative response capabilities, allowing the chatbot to access external knowledge sources for improved response quality. The RAG implementation included document loading using PyPDFDirectoryLoader from the LangChain library to load relevant Python programming documentation such as tutorials and guides suitable for beginners, and text splitting using RecursiveCharacterTextSplitter to segment documents into manageable chunks for efficient retrieval, ensuring each chunk contains coherent information with `chunk_size=500` as the maximum number of characters in each split chunk and `chunk_overlap=100` as the number of overlapping characters between consecutive chunks to preserve context continuity. Embedding generation utilized HuggingFaceEmbeddings to convert document chunks into dense vector representations using a pre-trained sentence embedding model from Sentence Transformers "sentence-transformers/all-mpnet-base-v2" enabling semantic similarity searches, while the vector store leveraged the Chroma vector database from LangChain.vectorstores to store the generated embeddings, with the `Chroma.from_documents` method used to initialize the vector store and `persist()` called to enable persistent storage of the database for future use. The retrieval chain implemented a RetrievalQA chain using LangChain, integrating the fine-tuned Gemma-2B-IT model via HuggingFacePipeline, where this chain retrieves the top-k relevant document chunks and augments the model's context before generating a response. The RAG pipeline ensures that the chatbot's responses are grounded in external knowledge, improving factual accuracy and relevance, especially for complex or less familiar queries. Response quality demonstrated that the chatbot consistently produced accurate, concise, and engaging explanations tailored to young learners, such as when asked "What is a loop in Python?", the chatbot responded with analogies like "a loop is like a merry-go-round that repeats actions until told to stop," followed by simple code examples. RAG performance showed that the integration of RAG improved response

accuracy for queries requiring external knowledge, as evaluated on a subset of test questions, with the retrieval mechanism ensuring responses were grounded in verified Python documentation.

3.3 Environment

We have used Google Colab, also known as “Colaboratory”, which is a cloud-based Jupyter notebook environment provided by Google Research and also we have used anaconda’s jupyterlab environment

Software Environment

The pre-processing and training of the models were done using two different tools, in Google Colaboratory graphic processing environment (GPU) which is a cloud based Jupyter notebook environment where we used the NVIDIA Tesla T4 with 15 GB of Ram in our Colab’s environment. Five of our 6 models were pre-processed and trained in Google Colaboratory graphic processing environment.

Technology/tools	Description
Windows 11	Used operating system
Visual studio code	Open-source code editor strong support for a wide range of programming
Google Colab	Cloud-based jupyter notebook used for data analysis , machine learning and other computational tasks

Table 1: Environment Software

Hardware Environment

For specific parts of the project, we have used the following hardware environment to develop and train some of the models.

Environment	Specifications
RAM	16.0 GB
Processor	AMD Ryzen 7 5800H with Radeon Graphics
Graphical card	Nvidia GeForce RTX 3070
Hard Disk	954 GB SSD
Screen	15.6

Table 2: Environment hardware

Programming Language and Libraries

Python serves as the primary programming language for implementing and developing our project. It is an interpreted, high-level, and versatile programming language that supports multiple programming paradigms. Python was instrumental in training and testing machine learning and deep learning algorithms. We utilized several key libraries and frameworks throughout our work, including Pandas for data manipulation and analysis, NumPy for numerical computations, TensorFlow for implementing neural networks and deep learning models, and Keras for simplifying model construction and experimentation. These tools collectively supported various stages of our project, including preprocessing, training, testing, evaluation, and analysis of results, ensuring efficient development and comprehensive exploration of machine learning methodologies.

3.3 Tutor

This section discusses the implementation of the Code Safari in detail. Code Safari is to provide an engaging and interactive Python learning experience for children aged 10 to 13. To deliver personalized coding questions, hints, and explanations, it interacts dynamically with our AI models, adapting content to each learner's level and progress.

The Code Safari interface offers a friendly and responsive environment where young learners can practice coding, ask programming questions, and receive immediate feedback.

Here is detailed implementation of our tutor :

3.3.1 MCQ generation feature

Conceptual Foundation and Pedagogical Approach

The MCQ generation system is built on a profound understanding of how children aged 10-13 learn programming. Unlike traditional educational tools, this system recognizes that learning to code is not about memorizing syntax, but about developing computational thinking and problem-solving skills. The entire architecture is designed to transform programming education from a potentially intimidating technical exercise into an engaging, interactive learning experience.

Question Generation: The Intellectual Core

The heart of the system is its question generation mechanism, which is far more complex than a simple random selection process. When initiated, the system begins a multi-stage process that combines artificial intelligence, educational theory, and sophisticated filtering mechanisms. The AI model is not just generating questions; it's creating carefully crafted learning experiences.

The generation process starts with a comprehensive set of guidelines that act as both a creative framework and a strict filter. These guidelines are meticulously designed to ensure that every generated question meets multiple criteria:

1. Syntactic Validity: Every question must represent 100% valid Python syntax that would run without errors.
2. Conceptual Clarity: Questions must test specific, tangible programming concepts.
3. Age Appropriateness: The language and complexity must be suitable for 10-13-year-olds.
4. Programmatic Verifiability: Each question must have a definitive, executable answer.

The system employs an innovative approach to preventing question repetition. It maintains a detailed history of previously asked questions stored in the browser's localStorage. This history is not just a simple list, but a sophisticated tracking mechanism that considers factors like topic distribution, question complexity, and temporal proximity.

Verification and Validation: The Quality Assurance Layer

Once a question is generated, it undergoes an incredibly rigorous verification process. This is not a simple syntax check, but a multi-layered validation that examines the question from multiple perspectives:

- Syntax Validation: Checks for complete code blocks, proper indentation, and correct use of Python-specific constructs.
- Conceptual Integrity: Ensures the question tests a clear, specific programming concept.
- Distractor Quality: Analyzes the generated options to ensure they represent plausible misunderstandings.

The verification process includes sophisticated pattern matching and AI-assisted evaluation. It looks for potential issues like incomplete code blocks, references to undefined variables, or questions that might introduce misconceptions.

Adaptive Learning Mechanism

The quiz is not a static test, but an adaptive learning environment. The system dynamically adjusts its question selection based on the user's performance. If a user struggles with a particular topic, the system will subtly increase the representation of questions from that area, creating a personalized learning path.

The streak tracking mechanism is particularly ingenious. It's not just a simple counter, but a complex scoring system that reflects the user's growing understanding. As users maintain their streak, the system gradually introduces more nuanced and challenging questions, creating a sense of progression and achievement.

Psychological and Educational Design

The entire user interface is designed with careful consideration of child psychology. The 30-second timer isn't just a technical constraint - it's a carefully calibrated mechanism to challenge users without causing undue stress. The immediate feedback system is crafted to provide constructive learning, transforming potential failures into educational opportunities.

Technical Complexity and Innovation

From a technical perspective, the system is a marvel of client-side processing. It leverages modern web technologies to create a responsive, dynamic learning experience. The use of localStorage for persistent tracking, the AI-driven question generation, and the sophisticated verification mechanisms represent cutting-edge approaches to educational technology.

The fallback question system is particularly noteworthy. It's not simply a predetermined set of questions, but a carefully curated collection of verified, high-quality programming challenges that can be deployed if the dynamic generation encounters any issues.

Broader Educational Impact

Beyond its immediate function as a quiz, the system serves a broader educational mission. It aims to:

- Demystify programming
- Build computational thinking skills
- Create a positive, engaging relationship with coding
- Provide a safe, supportive learning environment

The system goes beyond traditional educational approaches by treating learning as an interactive, enjoyable process. It transforms coding from an abstract, intimidating concept into an exciting challenge that children can approach with confidence and curiosity.

Continuous Evolution

The design allows for continuous improvement. Each interaction provides valuable data about how children interact with programming concepts, potentially informing future iterations of the learning tool.

3.3.2 Code Visualizer feature

The experience begins when the web page loads the “AI Artist” game container. A light Tailwind-styled interface immediately asks for a Gemini API key. As soon as the key is supplied, a background routine probes the Gemini family—starting with `gemini-1.5-flash`, then falling back to `gemini-1.5-pro`, `gemini-pro-vision` and finally `gemini-pro`—by sending a harmless hello message. The first model that answers successfully is stored in a

global variable so every subsequent call goes straight to that endpoint. This probing does two jobs: it hides any latency spikes that would otherwise appear the first time a real prompt is sent, and it gives you graceful degradation if the newest model is not yet enabled on a learner’s quota.

When the learner presses “**Visualize this code now**”, the raw text from the textarea is copied into a single source of truth (CURRENT_CODE) and a quick set of safeguards checks that something non-empty has been provided. A confirmation panel then echoes the code back to the learner—important for trust when you are about to call an external API—and a small animated spinner appears to signal that the heavy lifting has started.

Stage one of the heavy lifting is syntactic and semantic inspection performed entirely in JavaScript. The file you supplied contains a homemade parser that does not try to execute the Python but instead walks line by line building an intermediate representation (IR). For every line the parser notes indentation, detects classes of statements (assignments, loops, conditionals, function definitions, calls, print statements, obvious string operations, simple arithmetic expressions) and stores rich metadata such as variable names, literal types, range parameters or conditional expressions. While creating that IR the script also estimates likely console output so the later commentary can mention “and this code prints five rows of stars” without ever running untrusted user code. At the end of that pass a scoring heuristic decides which broad “topic” the snippet most strongly teaches—Variables, Loops, Conditionals, String Manipulation, Functions or Operators. All of these details are kept in one object that reads almost like an AST lite.

Stage two is the drawing phase. A canvas with a white reset is targeted and the IR decides which bespoke renderer to invoke. If Loops win, a circular timeline of iteration boxes is painted; if Conditionals win, a diamond-shaped flow-chart with coloured branches appears; if nothing scores highly a generic three-box start–process–end diagram is displayed. Each renderer is tuned for children: pastel backgrounds, large typography, short explanatory captions and bright arrows whose colours match the logical paths (green for true, red for false and so on). Because the code may be visualised on phones or

desktops, every measurement is expressed as a proportion of the canvas and re-calculated when the slider resizes the surface so that nothing blurs. At the end of the draw call the canvas now contains a tidy PNG-compatible picture that summarises the core concept.

Stage three is the multimodal piece. The canvas is instantly converted to a Base64 data-URL in memory and, alongside the original source code and the IR, fed into two distinct prompt templates. The first prompt asks Gemini to craft an “AI Artist’s Commentary”. It explicitly enumerates how many loops, variables and so on were discovered, identifies the dominant concept and gives stylistic instructions: five sentences maximum, simple but correct terminology, upbeat tone appropriate for ten-year-olds. The second prompt asks for “What the AI Sees”, nudging the model to describe colours, shapes and spatial relationships in the picture and then tie those visuals back to the code semantics.

Although the image itself is not uploaded (that would require Gemini Pro Vision), the prompt provides enough structural hints—title text, central diagram themes—for the model to speak credibly about the canvas layout. Both calls use the working model found earlier; if a 401 or quota error comes back a friendly status bar explains what went wrong so the learner is never left staring at a silent page.

Once the two textual responses arrive, the interface hides the spinner, expands the commentary and perception panels and injects the plain strings returned by Gemini. No HTML is allowed in those strings, protecting the page from injection, and the field is toggled with Tailwind classes so the layout stays neat. From there the learner can drag the size slider to redraw a larger or smaller version of the graphic (the code is reparsed only if necessary, keeping the session responsive), or tap “Download” to save the current canvas as a PNG—perfect for printing or attaching to homework submissions.

The final stage of the loop is key management. The yellow banner at the top lets a parent or teacher paste their own key at any time. Pressing “Update key” instantly runs the model-probing handshake again so you can switch accounts without refreshing the page. If the key box is cleared or the connection test fails, a red alert band shows a short diagnostic message and any AI-dependent buttons are effectively paused; the rest of the

visualiser continues to work offline so the learner can still see diagrams generated purely from the client-side parser.

Altogether this workflow knits together static analysis, real-time graphics and LLM explanations in a self-contained browser application. Each layer adds context for the next one—code to IR, IR to picture, picture plus IR to language—so that the textual advice Gemini produces feels grounded in what the child actually typed and what the child can see. By shielding the model behind a friendly commentary veneer, ensuring every explanatory sentence directly references either a code element or a visible colour/shape, and by filtering everything through age-appropriate language, the feature succeeds in turning abstract Python syntax into concrete, memorable lessons for the 10-to-13 age bracket.

3.3.3 Debugging Detective game feature

The Debugging Detective game leverages the fine-tuned Llama-3.2-3B-Instruct model to deliver an interactive educational experience teaching programming concepts through debugging challenges. The complete gameplay workflow operates as follows:

Challenge Generation Phase

Content Preparation:

- The fine-tuned model dynamically generates debugging challenges based on specified parameters
- Each challenge request includes a programming topic (e.g., Conditional Logic, Loops) and difficulty level
- Generated content follows the structured format established during training: original code, buggy implementation, graduated hints, solution, and explanatory content

Game Presentation Loop

1. Case Introduction:

- Player is presented with a new "case" (debugging challenge)
- The buggy code is presented for inspection

2. Interactive Decision Points:

- Players are offered three action options at each step:
 - Request a hint (incremental assistance)
 - Submit their proposed fix (active problem-solving)

3. Hint System:

- The player can request a hint for the question if they find it difficult to solve

4. Solution Submission:

- Players can input their corrected code implementation
- Self-assessment mechanism allows players to indicate whether their solution works
- Successful solutions trigger case completion and score advancement

Scoring and Progression Mechanics

1. Point Allocation System:

- Base points awarded for successful completion: 1 point

2. Score Tracking:

- Current score: Cumulative points from the active gameplay session
- Session termination resets the score, creating replay incentive

- Best score: Persistent high score maintained across sessions

Session Termination

1. Continuation Decision Point:

- After each case resolution, players are prompted to continue or end session
- Game loop continues until player opts to exit

2. Session Summary:

- Final score displayed upon session termination
- Challenge completion count reported
- Implicit encouragement to improve through score reset mechanism

3.3.4 Code Snippet feature

After I finetuned the model with our dataset and upload it in hugging face called "doaa18/codegen" from Hugging Face's model repository. This model is specifically designed to generate Python coding questions. When the system initializes, it loads both the tokenizer which converts text to numbers the model can understand and the actual neural network model into memory. The model uses transformer architecture, similar to GPT models, but specifically fine-tuned for code generation tasks.

The code generation process begins when a user requests a new exercise. The system feeds a minimal seed prompt ("# Task:") to the AI model, which then generates a complete programming exercise. The model uses sophisticated sampling techniques with temperature and top-p parameters to ensure the generated code is both creative and syntactically correct. The temperature setting of 0.7 provides a balance between randomness and coherence, while top-p sampling of 0.95 ensures the model considers a diverse range of possible continuations while filtering out very unlikely options.

Once the AI generates the raw output, the system performs intelligent parsing to extract meaningful components. The generated text typically contains a task description followed by a function definition with a complete implementation. The system splits this content at the first occurrence of "def", separating the problem description from the actual code solution. This parsing is important because it allows the system to present with just the problem statement while keeping the complete solution for evaluation purposes.

The system then uses python's Abstract Syntax Tree (AST) module to analyze the generated function code. This parsing extracts critical information like the function name, parameter names, and the overall structure of the code. The AST parsing is much more reliable than simple string manipulation because it understands Python's syntax rules and can handle complex code structures, nested functions, and various formatting styles that might appear in the AI-generated code.

One of the most sophisticated aspects of the system is its automatic test case generation. The system doesn't just create random inputs; instead, it analyzes the function signature to understand how many parameters the function expects and then generates appropriate test cases based on common Python data types and edge cases. For single-parameter functions, it creates a comprehensive test suite including various integers (positive, negative, zero, large numbers), strings (empty strings, single characters, common words), lists (empty lists, single-element lists, multi-element lists), and boolean values.

For functions with multiple parameters, the system creates combinations of different input types, testing scenarios like mixed data types, boundary conditions, and typical use cases. The test generation is intelligent enough to actually execute each test case against the AI model's solution before including it in the final test suite. This means that only valid test cases that the model solution can actually handle are included, preventing issues where the test framework itself generates invalid scenarios.

The system implements a sophisticated code execution environment that balances functionality with security. When evaluating the user code, it doesn't simply use python's

built-in exec() function with full system access. Instead, it creates a restricted execution environment that includes only safe built-in functions like len(), str(), int(), mathematical operations, and common data structure manipulations. This prevents users from accidentally or intentionally running dangerous system commands while still allowing them to write comprehensive solutions.

The execution framework captures both standard output and error output, allowing the system to provide meaningful feedback when student code fails. It uses context managers and exception handling to ensure that even if student code contains infinite loops or other problematic constructs, the evaluation process won't crash the entire system. Each code execution is isolated, so one student's buggy solution won't affect other users or the system's stability.

Problem

```
# Task:  
Create a function in Python that takes in an array of integers and returns the maximum value.  
  
# Input:  
[7, 3, 6, 8, 4]  
  
# Solution:  
  
def find_max(arr):  
    max_val = arr[0]  
    for i in arr:  
        if i > max_val:  
            max_val = i  
    return max_val
```

Your Solution

```
def find_max(arr):  
    max_val = arr[0]  
    for i in arr:  
        if i > max_val:  
            max_val = i  
    return max_val
```

[Generate New Problem](#)

[Check My Solution](#)

 **Code is correct**

Great job! Your solution produces the same results as the model solution.

As we can see in the image above, the code is checked as correct answer since it follows the model solution and the code structure and logically is correct and actually returns the maximum value.

Problem

```
# Task:  
Create a function in Python that takes in an array of integers and returns the maximum value.  
  
# Input:  
[7, 3, 6, 8, 4]  
  
# Solution:  
  
def find_max(arr):  
    max_val = arr[0]  
    for i in arr:
```

Your Solution

```
def find_max(arr):  
    max_val = arr[0]  
    for i in arr:  
        if i < max_val:  
            max_val = i  
    return max_val
```

[Generate New Problem](#)

[Check My Solution](#)

 **Code is wrong - syntax error**

 [Show Expected Solution Structure](#)

In the image above, it's considered as wrong since the user solution does not follow the output of the model solution since it returns the minimum and not the maximum value as the question says as it is also structurally incorrect.

The result comparison logic is far more sophisticated than simple equality checking. The system implements a recursive comparison algorithm that can handle various Python data types appropriately. For numeric results, it allows for small floating-point precision differences, recognizing that different algorithms might produce results that are mathematically equivalent but not byte-identical due to floating-point arithmetic limitations.

For complex data structures like lists and tuples, the system performs element-wise comparison, recursively applying the same intelligent comparison logic to each element. This means that a student's solution that returns [1, 2, 3] will be considered equivalent to the model solution that returns (1, 2, 3) if the logical content is the same, even though the data types differ slightly. The system also handles string comparisons carefully, accounting for different string representations while maintaining accuracy requirements.

To ensure responsive performance, the system implements intelligent caching mechanisms. When a new problem is generated, the system immediately calculates the expected outputs for all test cases and stores them in memory. This pre-calculation means that when students submit solutions, the evaluation process doesn't need to re-run the AI model's solution multiple times, significantly speeding up the feedback process.

The caching system is sophisticated enough to handle different types of inputs and outputs, storing results in a way that allows for quick retrieval while maintaining data integrity. This approach also makes the system more reliable because it eliminates potential inconsistencies that could arise if the model solution were re-executed multiple times with slightly different system states.

3.3.5 Video Content

1. Workflow Overview and Objectives

The educational content generation workflow was designed with the primary objective of creating comprehensive, age-appropriate learning materials that complement video-based instruction for young Python programming students. The system addresses the critical need for synchronized study materials that reinforce video content while providing immediate reference resources for hands-on programming practice.

The workflow operates on the principle of automated content generation, leveraging advanced natural language processing technologies to transform video content into structured educational materials. This approach ensures consistency in educational quality while maintaining the scalability necessary to serve diverse learning needs across multiple programming topics.

2. Phase One: Video Content Selection and Organization

The workflow commences with the systematic organization of educational video content sourced from the carefully curated "Hello Python 3 Programming for Kids & Beginners" course. Each video in our educational sequence addresses specific programming concepts, ranging from fundamental Python introduction to advanced topics such as functions and operators.

During this initial phase, every video is assigned a unique identifier and categorized according to its primary educational focus. The system maintains detailed metadata for each video, including topic classification, target learning objectives, and relevant programming keywords. This organizational structure enables the subsequent artificial intelligence system to generate appropriately focused educational content that aligns precisely with each video's instructional goals.

The video organization process also involves the establishment of learning progressions, ensuring that concepts build logically from foundational principles to more complex programming applications. This structured approach facilitates the creation of coherent study materials that support the overall educational narrative established by the video curriculum.

3. Phase Two: Interactive User Interface and Content Request

The workflow incorporates an intuitive user interface that enables students and educators to interact with the educational content generation system. The interface presents video content in an organized, visually appealing format that encourages engagement while providing clear pathways for accessing additional learning resources.

Students navigate through the educational content using a structured lesson progression system that tracks their advancement through the curriculum. The interface includes interactive elements that allow learners to request customized study materials for any specific video lesson. This on-demand content generation approach ensures that learning

materials are created precisely when needed, optimizing system resources while maintaining immediate accessibility.

The user interface also incorporates progress tracking mechanisms that monitor student advancement through the curriculum. This tracking capability enables the system to provide personalized learning experiences while ensuring that generated content remains appropriate for each student's current skill level and learning progression.

4. Phase Three: Video Topic Analysis and Content Mapping

When a student requests learning materials for a specific video, the system initiates a comprehensive topic analysis process that forms the foundation for subsequent content generation. This analysis phase examines the video's educational focus, identifying key programming concepts, relevant terminology, and appropriate instructional approaches for the target age group.

The system maintains a comprehensive database of topic-specific information that includes detailed descriptions of programming concepts, relevant keywords, and age-appropriate explanations suitable for children aged ten to sixteen. This database serves as a reference framework that guides the artificial intelligence system in generating contextually appropriate educational content.

During the topic analysis phase, the system also determines the appropriate complexity level for the generated content, ensuring that explanations remain accessible to young learners while maintaining technical accuracy. This balance between simplicity and precision represents a critical aspect of the educational content generation process.

5. Phase Four: Artificial Intelligence Content Generation

The core of the workflow involves the deployment of our fine-tuned language model to generate comprehensive educational materials based on the analyzed video topics. The artificial intelligence system processes the topic information and creates two distinct types of learning resources: detailed study guides and concise quick reference materials.

The study guide generation process focuses on creating comprehensive explanations of programming concepts that reinforce and expand upon the video content. These materials include conceptual explanations, practical applications, common programming patterns, and potential challenges that students might encounter. The artificial intelligence system ensures that all explanations are presented in language appropriate for young learners while maintaining technical accuracy.

Simultaneously, the system generates quick reference guides that serve as immediate support resources during hands-on programming activities. These references include essential syntax information, common programming patterns, troubleshooting tips, and best practices relevant to the specific topic being studied. The quick reference format enables students to access critical information efficiently during their programming practice sessions.

6. Phase Five: Content Processing and Enhancement

Following the initial content generation, the workflow incorporates sophisticated content processing mechanisms that enhance the educational value and accessibility of the generated materials. This processing phase includes formatting optimization, code example verification, and content structure refinement to ensure maximum educational effectiveness.

The content processing system applies consistent formatting standards across all generated materials, creating a cohesive visual experience that supports effective learning. This includes the implementation of appropriate heading structures, bullet point organization, and visual emphasis techniques that highlight critical information and improve content readability.

During this phase, the system also performs quality assurance checks to verify the accuracy and appropriateness of all generated content. These checks include technical accuracy verification, age-appropriateness assessment, and educational coherence evaluation to ensure that all materials meet established quality standards.

7. Phase Six: Dynamic Content Rendering and Display

The processed educational content undergoes dynamic rendering that transforms the raw generated materials into visually appealing, interactive learning resources. This rendering process incorporates modern web technologies to create engaging presentations that capture student attention while facilitating effective learning.

The rendering system applies consistent visual styling that aligns with contemporary educational design principles. This includes the use of appropriate color schemes, typography choices, and layout structures that support different learning styles and preferences. The system also incorporates interactive elements that encourage student engagement with the educational content.

Code examples within the generated content receive special formatting treatment that includes syntax highlighting and clear visual differentiation from explanatory text. This formatting approach helps students distinguish between conceptual explanations and practical programming examples, supporting more effective learning outcomes.

8. Phase Seven: Download and Preservation Functionality

Recognizing the importance of offline access to educational materials, the workflow incorporates comprehensive download functionality that enables students to preserve generated content for future reference. This capability addresses the practical needs of students who may not always have reliable internet access or who prefer to work with printed materials.

The download system supports multiple format options, with particular emphasis on high-quality document generation that maintains the visual appeal and educational effectiveness of the original content. The system automatically optimizes content for print formatting while preserving all essential educational elements including code examples, explanations, and visual organization.

The preservation functionality also includes automatic filename generation that creates meaningful, organized file names based on the video topic and content type. This approach enables students to build comprehensive libraries of educational resources that remain easily accessible and well-organized over time.

9. Phase Eight: Quality Assurance and Continuous Improvement

The workflow incorporates ongoing quality assurance mechanisms that monitor the effectiveness of generated educational content and identify opportunities for system improvement. This includes automated content quality checks, user feedback integration, and performance monitoring to ensure consistent educational value.

The quality assurance process evaluates multiple aspects of the generated content, including technical accuracy, educational appropriateness, and alignment with established learning objectives. These evaluations inform continuous refinement of the artificial intelligence system and content generation parameters to enhance educational effectiveness over time.

The system also incorporates mechanisms for collecting and analyzing user engagement data, enabling the identification of particularly effective content patterns and areas where additional improvement may be beneficial. This data-driven approach supports the continuous evolution of the educational content generation system.

10. Workflow Integration and Educational Impact

The complete workflow represents a seamless integration of multiple technological components working in concert to support effective programming education for young learners. The system successfully addresses the challenge of providing immediate, relevant, and age-appropriate educational support that complements video-based instruction.

The workflow's design prioritizes educational effectiveness while maintaining the flexibility necessary to adapt to different learning styles and preferences. Students can

access comprehensive study materials for in-depth concept exploration while also obtaining quick reference resources for immediate problem-solving support during hands-on programming activities.

The integration of artificial intelligence with educational best practices creates a scalable solution that can accommodate growing numbers of students while maintaining consistent quality in educational content delivery. This scalability represents a significant advantage for educational institutions seeking to provide comprehensive programming education without proportional increases in instructional staff requirements.

11. Technical Architecture and System Reliability

The workflow operates on a robust technical architecture designed to ensure reliable performance under varying usage conditions. The system incorporates appropriate error handling mechanisms, alternative content generation pathways, and graceful degradation capabilities that maintain educational functionality even when individual system components experience challenges.

The architecture supports both real-time content generation and efficient resource utilization, ensuring that students experience minimal delays when requesting educational materials. The system also incorporates caching mechanisms that optimize performance for frequently requested content while maintaining the ability to generate fresh materials when needed.

System reliability measures include comprehensive logging, performance monitoring, and automated backup capabilities that protect against data loss and ensure continuous availability of educational resources. These measures support the consistent educational experience that students and educators require for effective learning outcomes.

3.3.6 Chatbot Text-to-Text QA for Kids

Code Safari represents a sophisticated educational chatbot designed specifically to teach Python programming to children aged 10-13 through an engaging, interactive conversation-based approach. The implementation combines modern web technologies with advanced AI techniques, including Retrieval-Augmented Generation (RAG) and fine-tuned language models, to create a personalized learning experience that adapts to each child's programming journey.

Dynamic Chat Interface Management

The chat interface employs sophisticated DOM manipulation to create a fluid conversational experience. The system initializes key interface elements including the chat messages container, message input field, and send button, establishing event listeners that respond to both click and keyboard interactions. The implementation supports multi-line input with automatic textarea resizing, accommodating longer questions or code snippets while maintaining visual appeal with a maximum height constraint of 150 pixels.

Intelligent Conversation History Management

One of the most sophisticated aspects of the JavaScript implementation is its conversation history management system. The application maintains a persistent conversationHistory array that tracks the complete dialogue between the student and AI tutor. Each interaction is stored with role identification ("user" or "assistant") and content, enabling context-aware responses that build upon previous exchanges.

The system implements intelligent history trimming to maintain optimal performance while preserving educational continuity. When the conversation exceeds 20 exchanges, the implementation employs a strategic pruning algorithm that retains the initial welcome message, adds a system notification about summarized content, and preserves the most

recent 10 interactions. This approach ensures that the AI maintains sufficient context for meaningful responses while preventing memory bloat that could impact performance.

Advanced Message Processing and Formatting

The addMessage function represents a cornerstone of the educational experience, implementing sophisticated content processing that transforms raw AI responses into engaging, readable educational content. For bot messages, the system employs marked.js for Markdown parsing, enabling rich formatting including headers, lists, and emphasized text that helps structure learning content effectively.

The implementation includes specialized code block processing that enhances the programming education experience. Code snippets are automatically wrapped in styled containers with language identification headers and integrated copy buttons. The system carefully handles HTML entity decoding within code blocks to ensure that programming syntax remains accurate and executable, addressing common issues where Markdown processors might escape important programming characters.

Real-Time User Experience Enhancements

Code Safari prioritizes immediate feedback and responsive interaction to maintain student engagement. The system implements visual processing indicators that disable input elements during API calls, preventing user confusion and duplicate submissions. A dynamic typing indicator with animated dots creates the illusion of the AI tutor actively formulating responses, maintaining engagement during processing delays.

The interface includes sophisticated error handling that gracefully manages network failures or API timeouts. When primary systems fail, the application seamlessly transitions to fallback response mechanisms while maintaining the conversational flow, ensuring that technical issues never completely interrupt the learning experience.

Educational Content Enhancement Features

Recognizing the unique needs of programming education, the JavaScript implementation includes specialized features for code presentation and interaction. The system integrates highlight.js for syntax highlighting, automatically processing code blocks to provide visual cues that help students understand programming structure and syntax patterns.

Each code block includes a sophisticated copy mechanism accessible through a dedicated button that provides visual feedback when code is successfully copied to the clipboard. The copy function includes error handling and temporary visual state changes, creating a polished user experience that encourages code experimentation and practice.

Fallback Response System with Educational Intelligence

The implementation includes a comprehensive fallback response system that maintains educational value even when the primary AI system is unavailable. The getFallbackResponse function analyzes user input against a curated set of programming topics including variables, loops, and functions, providing structured educational content with properly formatted code examples.

The fallback system demonstrates context awareness by analyzing conversation history to provide more nuanced responses for follow-up questions. It recognizes patterns in user queries and adjusts response strategies accordingly, encouraging more detailed questions when initial queries are too brief or vague.

Model Deployment and Optimization

Inference Optimization

For deployment, the fine-tuned model was loaded with the following optimizations:

- **Inference Mode:** Enabled via `torch.inference_mode()` for faster execution
- **Quantization:** Maintained 4-bit quantization for low memory usage
- **Generation Parameters:**

- Temperature: 0.6 for controlled, child-appropriate responses
- Top-p: 0.9 for diverse yet relevant outputs
- Max New Tokens: 256 to accommodate detailed explanations

The deployment process utilized a Modelfile configuration to upload the fine-tuned model to Ollama. This Modelfile specifies the quantized model path (`./gemma-2b-finetuned-q4_k_m.gguf`), and includes a system prompt that defines the model's behavior as a friendly tutor who explains programming concepts to 10-year-olds using emojis and simple, relatable examples. This configuration ensures the model maintains its educational focus and child-appropriate communication style when deployed through the Ollama framework.

Backend Architecture: RAG-Enhanced API Service

The backend implementation (`rag_app.py`) creates a robust API service that orchestrates the interaction between the web interface and the sophisticated AI tutoring system. Built on FastAPI, the service provides high-performance endpoints that handle both simple queries and complex conversation-aware interactions.

Advanced RAG Pipeline Integration

The system employs a sophisticated Retrieval-Augmented Generation pipeline that combines the fine-tuned "gemma-2b-finetuned" model with semantic search capabilities. The implementation uses SentenceTransformerEmbeddings with the "all-mpnet-base-v2" model to create high-quality vector representations of educational content, stored in a Chroma vector database for efficient similarity-based retrieval.

The RAG chain is implemented using LangChain's RetrievalQA framework, which orchestrates the complex process of query analysis, context retrieval, and response generation. This architecture ensures that responses are grounded in verified educational content while allowing the language model to adapt explanations to the specific context of each conversation.

Conversation-Aware Response Generation

The API service implements dual endpoints to support both legacy single-query interactions and advanced conversation-aware processing. The primary /ask endpoint accepts conversation history alongside current queries, enabling the AI to provide responses that build upon previous interactions and maintain educational continuity.

The system formats conversation history into a structured context that helps the AI understand the learning progression and adjust explanations accordingly. This approach enables more sophisticated educational interactions where the AI can reference previous examples, build upon established concepts, and provide increasingly advanced guidance as students demonstrate mastery.

Quality Assurance and Reliability Features

Both the frontend and backend implementations include comprehensive error handling and quality assurance mechanisms. The JavaScript client implements multiple fallback layers, while the backend service includes response validation that filters out poor-quality or inappropriate responses, ensuring consistent educational value.

The system's architecture creates a resilient learning environment where technical failures are handled gracefully, conversation context is preserved across interactions, and educational content is presented in an engaging, age-appropriate manner that encourages continued learning and exploration of programming concepts.

Through this comprehensive implementation, Code Safari's chatbot successfully bridges the gap between advanced AI capabilities and practical educational applications, creating an interactive learning environment that adapts to individual student needs while maintaining the reliability and safety standards essential for educational technology platforms.

Chapter 4: Conclusion

4.1 Future work

While our AI-powered Python programming tutor provides a strong foundation for children aged 10 to 13 to learn programming, we recognize there is room for enhancement. We plan to add several features to further enrich the learning experience.

Firstly, we plan to enhance interactivity by adding more educational games and coding challenges. These additions will make learning more fun and help reinforce programming concepts through practice.

Moreover, for educators and parents, we intend to develop comprehensive analytics dashboards that offer insights into the student's progress, strengths, and areas needing improvement. These tools will enable educators to tailor their teaching strategies and provide targeted support, while parents can stay informed about their child's learning journey.

Additionally, we aim to support children with disabilities by introducing accessibility features such as screen readers, alternative input methods, and visual or auditory aids. This will ensure that children with varying abilities can benefit from the platform without barriers.

Finally, we plan to introduce animated characters integrated into the platform. These characters will interact with and assist the child, acting as personal instructors. They will guide learners through lessons, provide encouragement, and make the learning process more engaging and interactive.

4.2 Challenges

The integration of multiple advanced AI models within the platform posed several significant challenges. High-performance GPUs and extended training times demanded substantial computational resources, which required careful management due to limited hardware availability and financial constraints. Securing the infrastructure necessary to fine-tune and deploy models such as unsloth/Llama-3.2-3B-Instruct and Gemini-1.5-flash was particularly resource-intensive.

Developing the code snippet generation and debugging features using Salesforce/codegen-350M-mono involved overcoming data scarcity issues. Real-world coding examples had to be carefully curated and adapted, as there was no readily available dataset perfectly suited for young learners. This necessitated painstaking data collection, annotation, and validation to ensure both pedagogical relevance and technical correctness.

Evaluating the performance of generative models like Gemma3 : 4b and Gemma-2b instruct presented additional difficulties. Traditional automatic evaluation metrics, such as BLEU or ROUGE, proved insufficient to capture the nuanced quality of tutoring content or conversational clarity. Consequently, the project relied heavily on qualitative human assessments and iterative feedback cycles to validate output coherence and educational effectiveness.

Finally, implementing the multimodal feedback system leveraging Gemini-1.5-flash required novel methodologies for merging visual and textual data streams. Designing robust evaluation strategies and ensuring smooth interpretation of combined code and visual outputs demanded cross-disciplinary expertise and extensive experimentation.

These challenges highlighted the complexity of delivering a truly adaptive, interactive AI-powered learning platform tailored for young programmers, underscoring the need for ongoing refinement and resource investment.

4.3 Conclusion

Implementing the AI-powered Python programming tutor involved careful planning, data preparation, model selection, and training. Each component was designed with the target audience in mind, ensuring that the content is engaging, educational, and age-appropriate.

By integrating interactive exercises, real-world code examples, educational games, and supplementary materials like videos and cheat sheets, we have created a comprehensive learning platform. The use of advanced AI models, such as Llama-3.2-3B-Instruct and Gemma-2b instruct, has enabled us to deliver personalized and effective learning experiences.

Throughout the development process, we remained committed to addressing the challenges of data imbalance, computational constraints, and the need for content suitability. The result is a robust system that not only teaches Python programming but does so in a way that is accessible and enjoyable for young learners.

Appendix A

A demo of the tutor

This appendix consists of multiple screenshots showcasing **Code Safari Tutor**

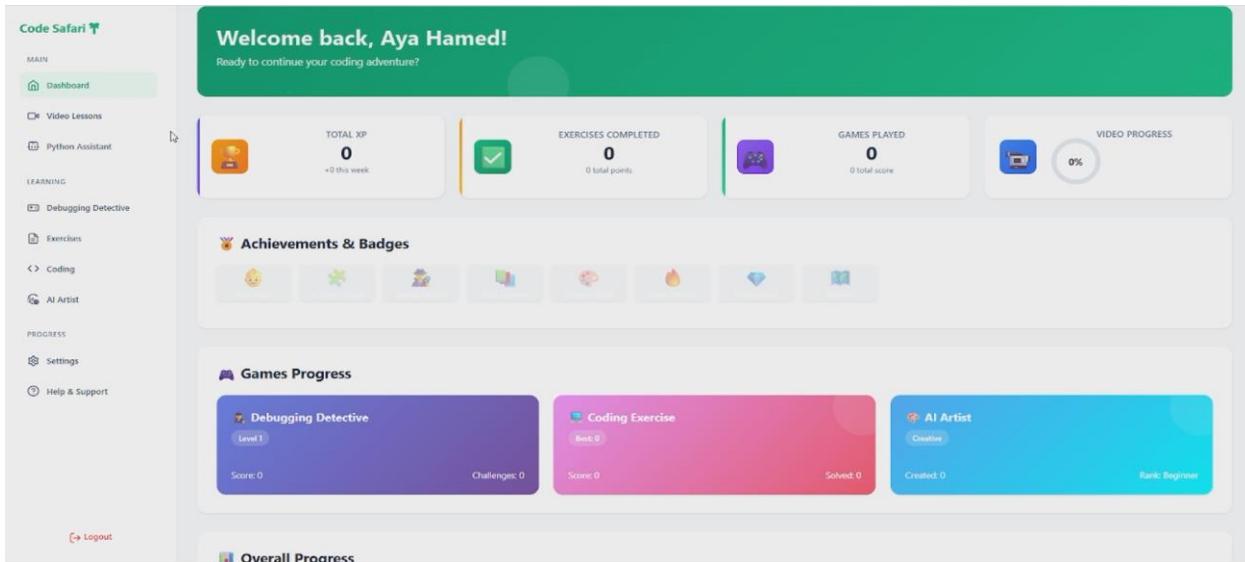


Figure 12: Dashboard



Figure 13: Video Lessons

Figure 14: Python Assistant

Figure 15: AI Artist

The screenshot shows a web-based coding exercise interface. At the top left is a pink speech bubble icon followed by the text "Coding Exercise". Below it is a sub-header "Challenge yourself with coding problems". A section titled "Problem" contains a heading "Model-Based Code Exercise System" with a small icon. Underneath is a "How It Works:" section with three bullet points: "Model Comparison", "Sample Testing", and "Smart Feedback". A note below states, "Your solution just needs to produce the same results as the AI model - multiple approaches accepted!". A button labeled "Generating new problem..." is visible. The main workspace is titled "Your Solution" with a yellow speech bubble icon. It includes a placeholder text "Write your solution here...". At the bottom are two blue buttons: "Generate New Problem" and "Check My Solution".

Figure 16: Code Snippets

The screenshot shows the "Debugging Detective" feature within the Code Safari platform. On the left is a sidebar with navigation links: MAIN (Dashboard, Video Lessons, Python Assistant), LEARNING (Debugging Detective, Exercises, Code Snippets, AI Artist), PROGRESS (Settings), and HELP & SUPPORT. The "Debugging Detective" link is highlighted. The main area has a purple header bar. Below it is a "Case Description" box containing the text "This code should print the string value stored in a variable." A "Code with Bugs" box shows the following Python code:

```
1 name = 'Hello World'  
2 print(name)
```

. A "Detective Hint" box says "Check if the variable assignment is correct." At the bottom are buttons for "Need a Hint?", "Submit Solution", and "Next Case". A green feedback box at the bottom right says "Case Solved! Great job! You fixed the bug in the code. You earned 15 points!"

Figure 17: Debugging Detective

Python Exercises

15 seconds left

What will `print(7 - 2 + 10)` ?

23

30

14

15

 **Correct! Well done!***In Python, this print statement outputs exactly what you see in the correct answer.*

References

- [1] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., & Xiong, C. (2022). *A Conversational Paradigm for Program Synthesis*. (presented along with the release of the CodeGen family)
- [2] Gemma Team & Kamath, A., Ferret, J., Pathak, S., Vieillard, N., Merhej, R., Perrin, S., Matejovicova, T., Rivière, M., & Rouillard, L. (2025, March 12). *Gemma 3 Technical Report*. (introducing Gemma 3 across scales up to 27B parameters)
- [3] Gemma Team, Rivière, M., ... (2024, August). *Gemma 2: Improving Open Language Models at a Practical Size*. arXiv preprint arXiv:2408.00118. (introduces 2B and 9B Gemma models with architecture improvements)
- [4] Meta Llama Team. (2024). *Llama 3.2-3B-Instruct Model Card*. (documentation for instruction-tuned 3B Llama 3.2)
- [5] Gemini Team, Georgiev, P., Lei, V. I., Burnell, R., Bai, L., Gulati, A., ... & Lepikhin, D. (2024, March 8). *Gemini 1.5: Unlocking Multimodal Understanding Across Millions of Tokens of Context*. arXiv preprint arXiv:2403.05530. (introduces Gemini 1.5 Pro and the lightweight 1.5 Flash variant)
- [6] Van Brummelen, J., Wu, T., & Lin, P. (2021). Teaching AI Concepts to K-12 Students with AI Dodgers: A Game-Based Approach. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (pp. 1324-1330).
- [7] Lorenzo-Lledó, A., Pérez Vázquez, E., Andreu Cabrera, E., & Lorenzo Lledó, G. (2023). Application of gamification in early childhood education and primary education: Thematic analysis.
- [8] Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., ... & Werner, L. (2011). Computational thinking for youth in practice. *ACM Inroads*, 2(1), 32-37.

- [9] Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. Proceedings of the 2012 Annual Meeting of the American Educational Research Association.
- [10] Kordaki, M. (2016). A framework for designing computer programming learning environments for children based on computational thinking perspectives. International Journal of Child-Computer Interaction, 9, 1-11.
- [11] Deterding, S., Dixon, D., Khaled, R., & Nacke, L. (2011). From game design elements to gamefulness: defining gamification. Proceedings of the 15th International Academic MindTrek Conference, 9-15.
- [12] Papadakis, S. (2020). Educational robotic applications that promote collaborative learning and computational thinking through gamification for young children. Journal of Educational Computing Research, 57(3), 682-701.
- [13] Holmes, W., Bialik, M., & Fadel, C. (2019). Artificial intelligence in education: Promises and implications for teaching and learning. Center for Curriculum Redesign.
- [14] Lee, J., & Ko, A. J. (2015). Personifying programming environments to help novices learn computational thinking. Proceedings of the 2015 International Conference on Software Engineering, 281-290.
- [15] Smith, S. & Bers, M. U. (2020). How Robot Programming Motivates Young Children to Learn Coding and Computational Thinking. International Journal of Social Robotics, 12, 1293–1306.