Project Report: Edu-Tutor AI

1. INTRODUCTION

1.1 Project Overview

Edu-Tutor AI is a personal learning companion developed with Gradio and based on an IBM Granite language model. It provides multiple features for learning support such as a chatbot for conversation, summarization and explanation of text, text improvement, word meaning and usage search, sentence translation, and an interactive quiz generation and performance monitoring system. The program is made user-friendly to allow an interactive learning process.

1.2 Purpose

The primary purpose of Edu-Tutor AI is to provide students and learners with a versatile tool that enhances their understanding of concepts, improves writing skills, expands vocabulary, and assesses their knowledge through customizable quizzes. By offering these integrated features, it aims to make learning more efficient and engaging.

2. IDEATION PHASE

2.1 Problem Statement

Students tend to struggle with reading difficult texts, improving writing, learning new words, and checking their own learning properly. Conventional learning can be inefficient in providing customized feedback as well as practice tools to make things interactive.

2.2 Empathy Map Canvas

• Say: "I wish I had someone to explain this simply." "My writing needs to be clearer." "How do I know if I really understand this topic?" "I need more practice questions."

• Think: "This passage is too thick." "Am I writing coherently in my essays?" "What does this word imply in context?" "I hope I'm ready for my exam."

• Feel: Struggling with thick material, bogged down with writing assignments, interested in learning new words, worried about grasping concepts.

• Do: Read texts again and again, look up online for definitions, attempt to paraphrase sentences, look for practice quizzes or exercises.

2.3 Brainstorming

• AI chatbot for straightforward questions and definitions.

• Text summarizer for instant understanding.

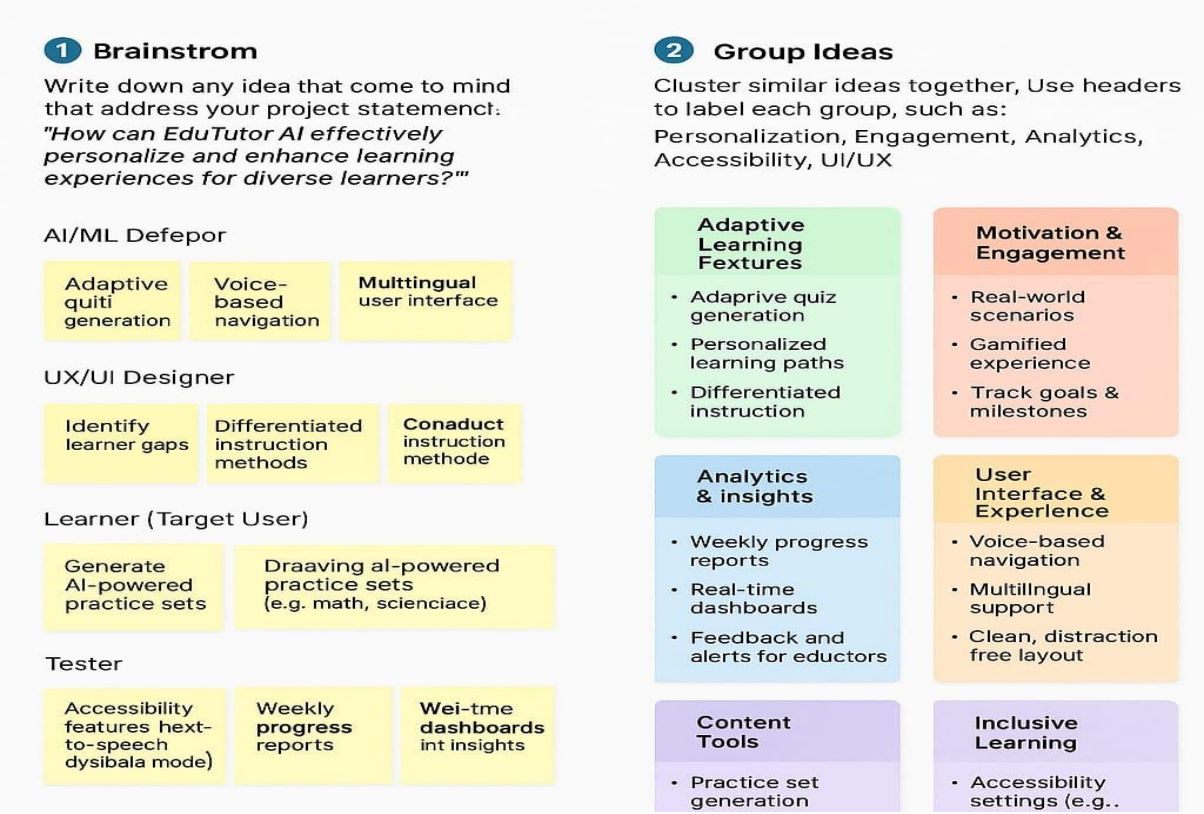
• Grammar and style refiner for enhancing writing.

• Vocabulary builder with examples and meanings.

• Quiz generator from user-defined topics or text.

• Performance monitoring for quizzes for weak area identification.

• Support for PDF input for content analysis.



3. REQUIREMENT ANALYSIS

3.1 Customer Journey Map

1. **Need to learn a topic:** User visits "Chat with Edu-Tutor" or "Text Summarizer."

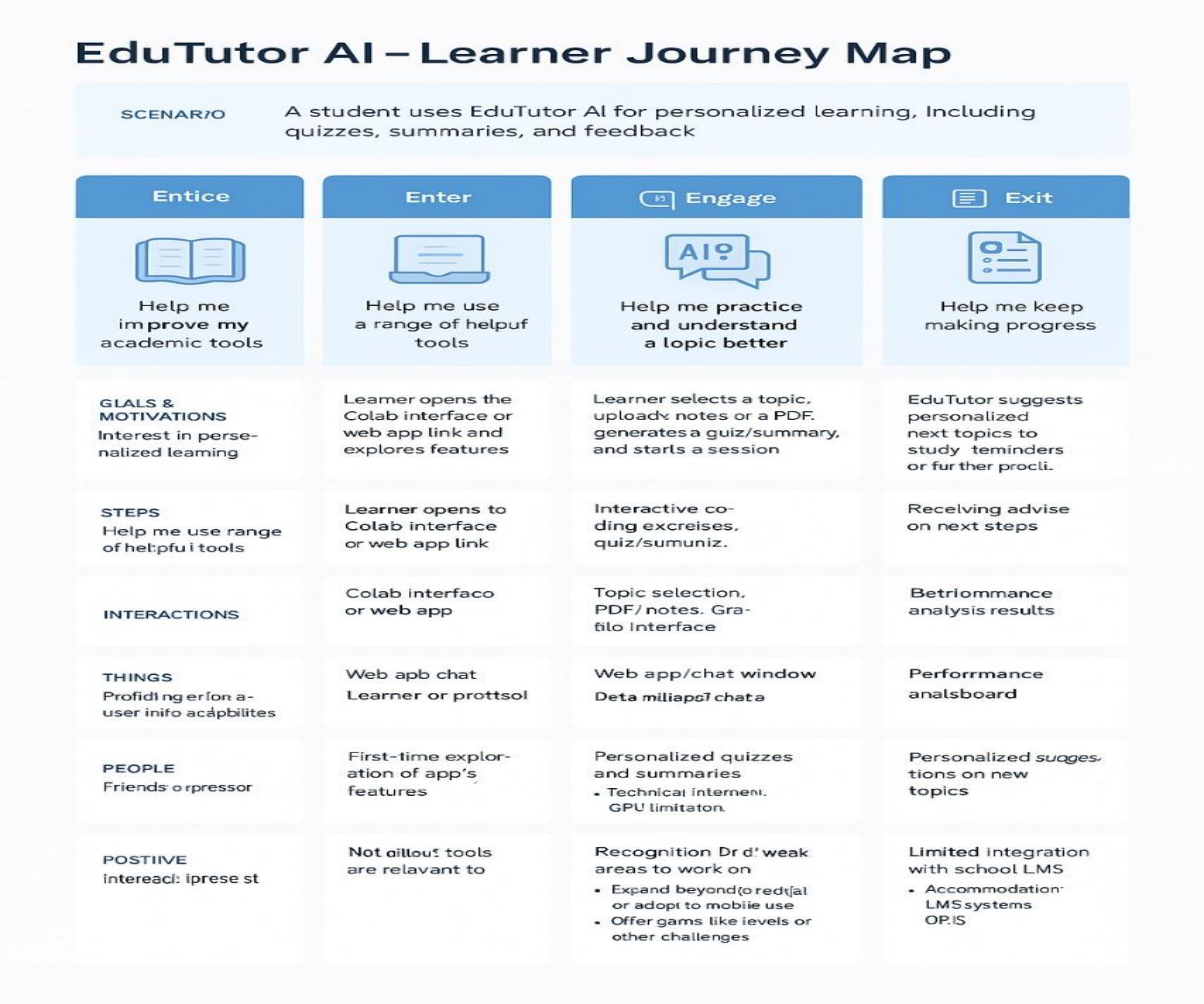
2. **Need to enhance writing:** User visits "Text Refiner."

3. **Encounter unknown word:** User navigates to "Word Meaning & Usage."

4. **Have to translate a sentence:** User navigates to "Sentence Translator."

5. **Need to practice knowledge:** User navigates to "Generate Quiz", enters topic/PDF, chooses difficulty, and takes the quiz.

6. **Check progress:** User navigates to "Performance Dashboard" to view previous scores and weak spots.



3.2 Solution Requirement

•Functional Requirements:

- o F.1: General-purpose AI chatbot.
- o F.2: Summarize text to brief, detailed, or easy explanations.
- o F.3: Enrich text for grammar, spelling, and clarity.
- o F.4: Offer meaning and usage examples for words.
- o F.5: Translate sentences into several target languages.
- o F.6: Create multiple-choice quizzes from user-supplied text or topics.
- o F.7: Accommodate PDF file upload for quiz creation.
- o F.8: Enable users to submit quiz answers and view immediate scoring and detailed feedback.
- o F.9: Retain quiz results (topic, difficulty, score) in a database.
- o F.10: Show historical quiz performance data.
- o F.11: Mark "weak spots" based on quiz results.
- o F.12: Recommend next steps for improvement based on performance.

•Non-Functional Requirements:

o NFR.1: Performance: The AI answers should be generated and streamed in a timely manner.

o NFR.2: Usability: Easy-to-use and intuitive Gradio interface.

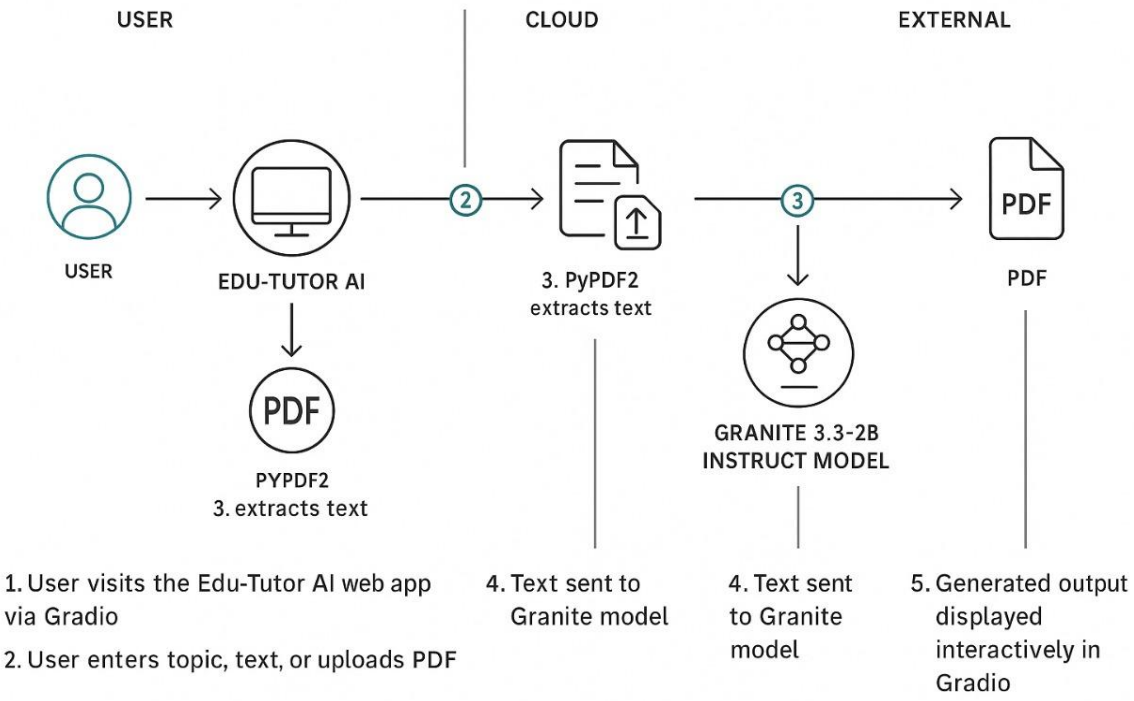
o NFR.3: Reliability: Stable model performance and database operation.

o NFR.4: Scalability: Can support multiple users (though not specifically designed for high concurrency in current implementation).

o NFR.5: Security: Data stored locally (SQLite).

3.3 Data Flow Diagram

- User Input: Text (chat, summarize, refine, word, translate, quiz topic), PDF file (quiz), Quiz answers (radio buttons)
- Application Logic:
- Gradio UI receives input.
- Inputs are sent to Python functions.
- Python functions build prompts for AI model.
- AI model produces responses.
- Appropriate for quizzes: raw text is processed; user responses are matched against correct responses.
- Quiz results are stored in SQLite database.
- Performance data read from SQLite.
- AI output structure: Chat, Summary, Refined Text structure, Meaning, Translation
- Interactive quiz questions
- Quiz results (score, feedback)
- Performance dashboard (history table, weak points, recommendations)



3.4 Technology Stack

•Frontend/UI: Gradio

•Backend/Logic: Python

•AI Model: IBM Granite 3.3-2b-instruct (or GPT-2 for mock)

•Natural Language Processing (NLP): Hugging Face Transformers library

•PDF Processing: PyPDF2

•Database: SQLite3

•GPU Acceleration: Accelerate, BitsAndBytes (model loading)

4. PROJECT DESIGN

4.1 Problem Solution Fit

Edu-Tutor AI confronts the identified issues head-on through offering a one-stop-shop learning assistance. The natural language interface breaks down difficult concepts, the refiner and the summarizer facilitate reading and writing comprehension, and the customizable quiz feature with performance monitoring provides an adaptive test tool.

4.2 Suggested Solution

The answer is a browser-based AI helper with separate functional modules merged into one Gradio program. Every module (Chat, Summarizer, Refiner, Word Meaning, Translator, Quiz Generator, Performance Dashboard) is offered as a different tab for easy navigation. The main AI functionality is implemented using a large language model, and data processing, UI handling, and database interactions are performed using Python.

4.3 Solution Architecture

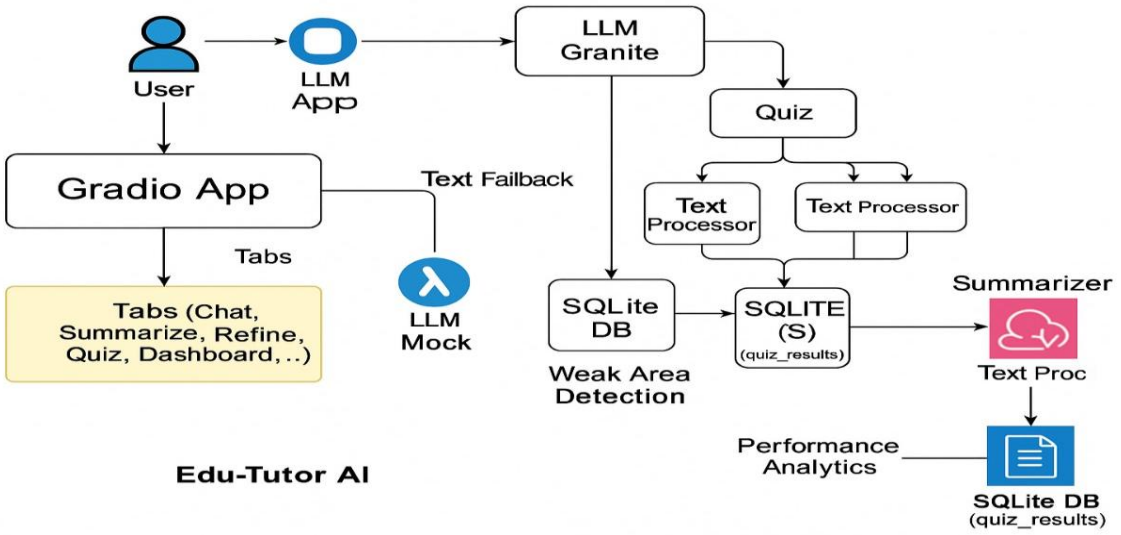
•**Gradio Interface Layer:** Offers the user-side web application with interactive elements (textboxes, buttons, radio buttons, dataframes, chatbots) grouped into tabs.

•**Application Logic Layer (Python):** Holds the functions for managing user input, prepping prompts, invoking the AI model, handling model output processing (e.g., quiz parsing), and communicating with the database.

•**AI Model Layer:** Has the ibm-granite/granite-3.3-2b-instruct model (or mock model) imported through Hugging Face Transformers for text generation operations.

•**Data Persistence Layer:** Quiz results are stored in an SQLite database (edututor.db) to allow tracking of performance.

•**Utility Libraries:** PyPDF2 for PDF parsing; accelerate and bitsandbytes for loading the model quickly and performing inference on GPU.



5. PROJECT PLANNING & SCHEDULING

5.1 Project Planning

•Phase 1: Setup and Core AI Integration (Done)

- o Install Python environment and Gradio, Transformers, PyPDF2, SQLite.
- o Integrate and test ibm-granite model (or mock) for simple text generation.
- o Implement chat.

•Phase 2: Core NLP Features (Done)

- o Develop text summarization.
- o Develop text refinement.
- o Develop word meaning and usage.
- o Develop sentence translation.

•Phase 3: Quiz System (Done)

- o Design quiz prompt for the AI model.
- o Implement parse_quiz_text for strong quiz extraction from raw output.
- o Develop interactive quiz UI using Gradio's dynamic components.
- o Implement quiz scoring logic.
- o Set up SQLite database for quiz results.
- o Implement record_quiz_result.

•Phase 4: Performance Tracking & UI Refinements (Done)

- o Implement get_performance_data and get_performance_insights.
- o Design "Performance Dashboard" tab with DataFrame, weak areas, and suggestions.

o Provide UI reset functions for quiz and overall clarity.

o Insert error handling and user feedback messages.

•**Phase 5: Testing & Deployment (Ongoing/Ongoing for Demo)**

o End-to-end test all functionalities.

o Fix parsing and model interaction bugs.

o Prepare deployment (e.g., Colab environment).

6. FUNCTIONAL AND PERFORMANCE TESTING

6.1 Performance Testing

•**Model Loading Time:** Observed while developing, the ibm-granite/granite-3.3-2b-instruct model loading time can be substantial, particularly without GPU acceleration.

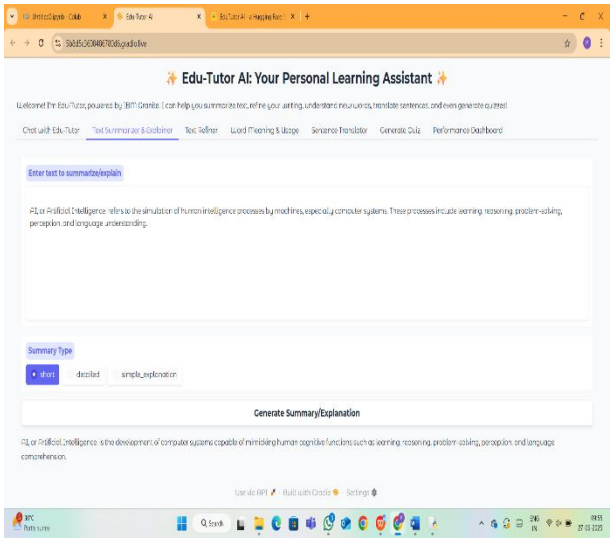
•**Response Latency:** Chat, summarization, refinement, etc., AI responses are streamed word-by-word with a STREAM_DELAY of 0.005 seconds, giving a responsive touch.

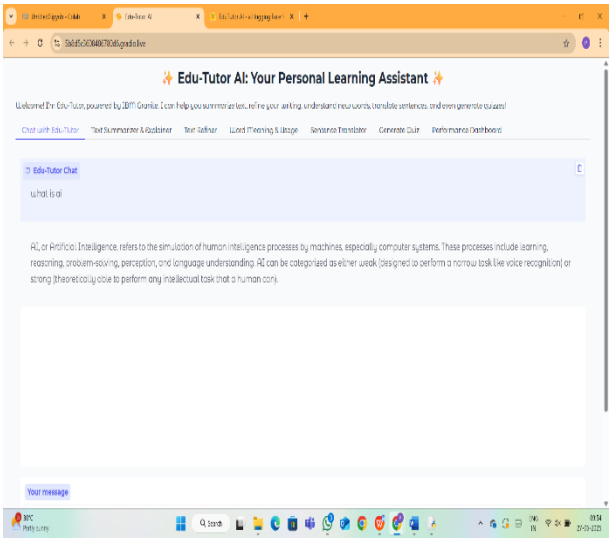
•**Quiz Generation Time:** Varies with question count and prompt complexity. The max_new_tokens for quiz generation is num_quiz_questions * 70 to control output length. Parse time is negligible after generation.

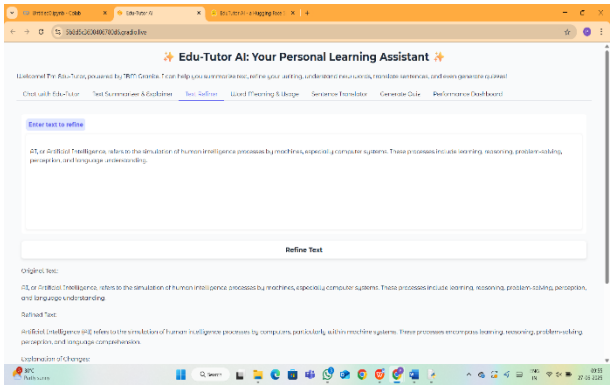
• **Database Operations:** SQLite operations for storing and retrieving quiz scores are extremely quick considering the local context and common data sizes for one user.

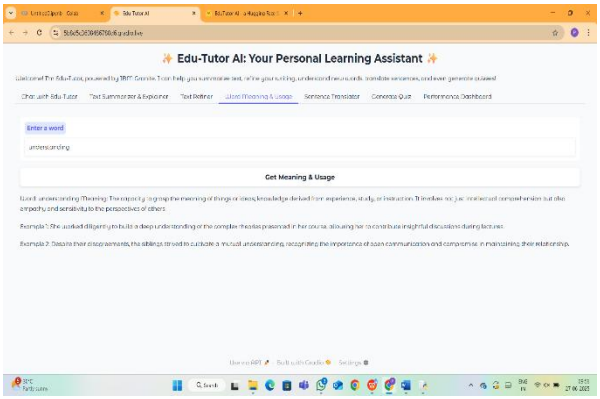
7. RESULTS

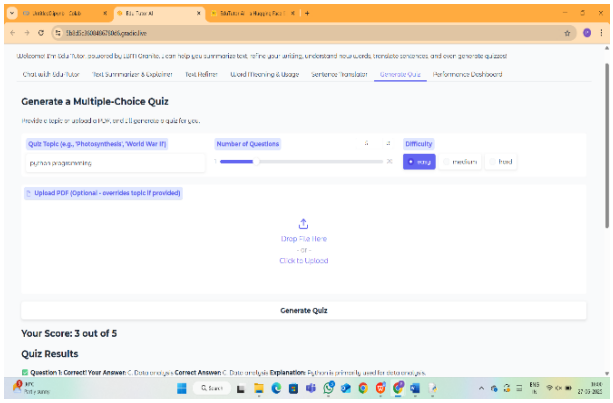
7.1 Output Screenshots

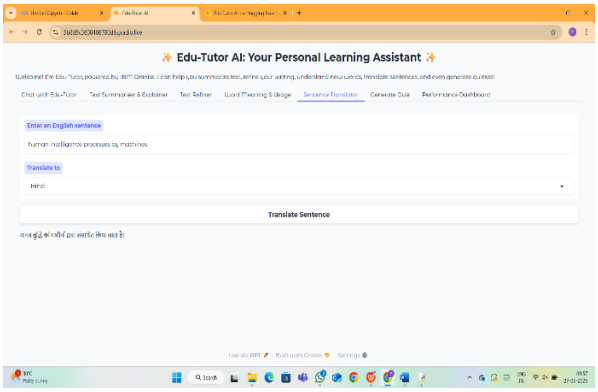


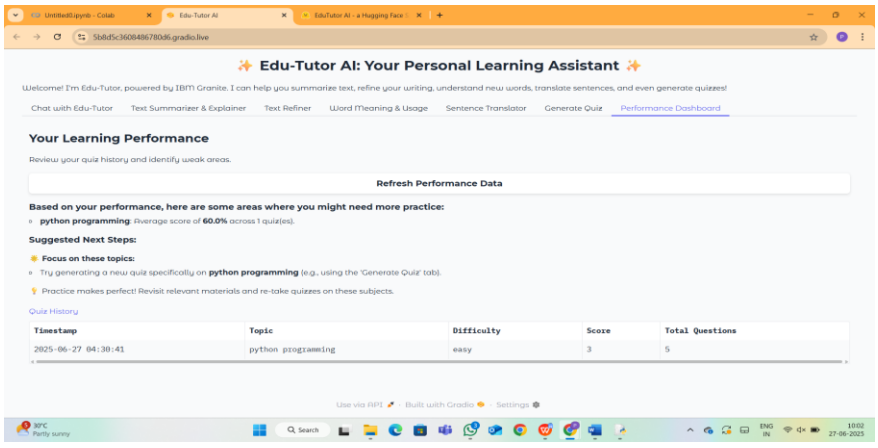












8. ADVANTAGES & DISADVANTAGES

Advantages:

•**Multi-functional:** Provides an extensive list of learning tools in a single program.

•**Personalized Learning:** Performance tracking and quiz generation enable intense practice on areas of weakness.

•**Interactive:** Chatbot and streaming responses improve the user experience.

•**Flexible Input:** Accommodates text input as well as PDF uploads for content analysis.

•**User-Friendly Interface:** Developed using Gradio, making it simple to use for even non-technical users.

•**Offline Persistence:** Quiz results are locally stored using SQLite.

Disadvantages:

•**Resource Hungry:** Operating a large language model such as Granite is very computationally expensive, both in terms of CPU power and GPU memory.

•**Model Dependent:** The accuracy and quality of answers depend directly on the underlying AI model.

•**Quiz Parsing Brittleness:** Although regex-based parsing of AI output is robust, it sometimes breaks if the model doesn't conform to the rigid output format, resulting in "Could not generate a valid quiz" errors.

•**Limited Scalability (Current Form):** Built for one user; multiple users running simultaneously may experience performance degradation without additional infrastructure.

•**No User Authentication:** No user login and profile management in the current implementation.

9. CONCLUSION

Edu-Tutor AI effectively provides an end-to-end personal learning assistant. It successfully combined various AI-driven capabilities for helping to comprehend, write, learn vocabulary, and assess oneself. The interactive Gradio environment puts such sophisticated tools at learners' fingertips, and the performance dashboard helps learners gain insights to monitor their progress and prioritize areas for improvement.

10. FUTURE SCOPE

• **Advanced Performance Analytics:** Incorporate more in-depth performance measures, trend analysis, and predictive insights into learning habits.

•**Adaptive Learning Paths:** Depending on performance, the AI may recommend certain learning resources or create customized practice exercises.

•**Multi-modal Input/Output:** Audio input/output support, image processing for educational content.

•**User Profiles and Progress Tracking:** Enable user authentication and personal progress tracking for multiple users.

•**Integration with External APIs:** Integrate with external educational databases or content platforms.

•**Improved Quiz Robustness:** Investigate more sophisticated parsing methods or model fine-tuning to provide more uniform quiz generation.

•**Deployment as a Web Service:** Deploy on a more robust cloud infrastructure (e.g., AWS, GCP, Azure) for wider accessibility and better scalability.

11. APPENDIX

Source Code

```
# NEW Cell 1: Install PyPDF2 aggressively and other core libs
!pip install -q PyPDF2
!pip install -q --upgrade PyPDF2 # Ensure it's the latest
!pip install -q transformers accelerate bitsandbytes gradio google-search-results
!pip install -q --upgrade transformers accelerate bitsandbytes gradio # Upgrade others

print("Core libraries installed.")

# Test the import of PyPDF2 immediately
try:
    import PyPDF2
    print("PyPDF2 imported successfully after installation.")
except ImportError:
    print("Error: PyPDF2 still cannot be imported. There might be a deeper issue.")

pip install gradio transformers torch accelerate PyPDF2

import gradio as gr
import os
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
from accelerate import Accelerator
import re
import time
import PyPDF2
import sqlite3
import json # Import json for pretty printing parsed data

# --- Global Constants ---
MAX_QUIZ_QUESTIONS_UI = 20 # Define this globally so UI can access it
WEAK_THRESHOLD_PERCENTAGE = 65 # Percentage below which a topic is considered a weak
area
quiz_context = {}
# --- Database Setup ---
DB_NAME = 'edututor.db'

def init_db():
    conn = sqlite3.connect(DB_NAME)
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS quiz_results (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            timestamp TEXT NOT NULL,
            topic TEXT NOT NULL,
            difficulty TEXT NOT NULL,
            score INTEGER,
            num_questions INTEGER
        )
```

```
    ''')
    conn.commit()
    conn.close()
    print(f"Database '{DB_NAME}' initialized successfully.")

def record_quiz_result(topic, difficulty, score, num_questions):
    conn = sqlite3.connect(DB_NAME)
    cursor = conn.cursor()
    timestamp = time.strftime('%Y-%m-%d %H:%M:%S')
    cursor.execute('''
        INSERT INTO quiz_results (timestamp, topic, difficulty, score, num_questions)
        VALUES (?, ?, ?, ?, ?)
    ''', (timestamp, topic, difficulty, score, num_questions))
    conn.commit()
    conn.close()
    print(f"Recorded quiz result: Topic='{topic}', Difficulty='{difficulty}',
Score={score}/{num_questions}")

def get_performance_data():
    """Fetches raw performance data from the database."""
    conn = sqlite3.connect(DB_NAME)
    cursor = conn.cursor()
    cursor.execute('SELECT timestamp, topic, difficulty, score, num_questions FROM quiz_results
ORDER BY timestamp DESC')
    data = cursor.fetchall()
    conn.close()
    return data

def get_performance_insights():
    """
    Analyzes quiz performance data and generates markdown for weak areas
    and suggested next steps, along with the dataframe for history.
    """
    performance_data = get_performance_data() # Fetch raw data

    # We return the actual data for the DataFrame, Gradio will handle updating its value
    df_data_for_update = performance_data

    if not performance_data:
        # Return empty data for DataFrame, and messages for markdown
        return [], "No quiz data available for analysis. Please take some quizzes first!", "No specific
suggestions at this time."

    topic_scores = {} # {topic: [score_percentage, ...]}
    for timestamp, topic, difficulty, score, num_questions in performance_data:
        if num_questions > 0:
            percentage = (score / num_questions) * 100
            if topic not in topic_scores:
                topic_scores[topic] = []
            topic_scores[topic].append(percentage)

    weak_areas_list = []
```

```
    for topic, percentages in topic_scores.items():
        avg_percentage = sum(percentages) / len(percentages)
        num_quizzes = len(percentages)
        if avg_percentage < WEAK_THRESHOLD_PERCENTAGE:
            weak_areas_list.append({
                "topic": topic,
                "avg_score_percentage": avg_percentage,
                "num_quizzes": num_quizzes
            })

    # Sort weak areas by average score (lowest first)
    weak_areas_list.sort(key=lambda x: x['avg_score_percentage'])

    # --- Weak Areas Markdown ---
    weak_areas_markdown = ""
    if not weak_areas_list:
        weak_areas_markdown = "🎉 **Excellent work!** No significant weak areas detected based on your quiz performance. Keep up the great work!"
    else:
        weak_areas_markdown = "### Based on your performance, here are some areas where you might need more practice:\n\n"
        for area in weak_areas_list:
            weak_areas_markdown += f"- **{area['topic']}**: Average score of **{area['avg_score_percentage']:.1f}%** across {area['num_quizzes']} quiz(es).\n"

    # --- Suggested Steps Markdown ---
    suggested_steps_markdown = ""
    if weak_areas_list:
        suggested_steps_markdown += "### Suggested Next Steps:\n\n"
        suggested_steps_markdown += "🌟 **Focus on these topics:**\n"
        for area in weak_areas_list:
            suggested_steps_markdown += f"- Try generating a new quiz specifically on **{area['topic']}** (e.g., using the 'Generate Quiz' tab).\n"
        suggested_steps_markdown += "\n💡 Practice makes perfect! Revisit relevant materials and re-take quizzes on these subjects."
    else:
        suggested_steps_markdown = "### Suggested Next Steps:\n\n"
        suggested_steps_markdown += "Keep exploring new topics or challenge yourself with a 'hard' difficulty quiz! You're doing great!"

    return df_data_for_update, weak_areas_markdown, suggested_steps_markdown


# --- Model Loading (Local Inference) ---
GRANITE_MODEL_NAME = "ibm-granite/granite-3.3-2b-instruct"

tokenizer = None
model = None
device = "cuda" if torch.cuda.is_available() else "cpu"

def load_granite_model():
```

```
    global tokenizer, model
    if model is not None and tokenizer is not None:
        return tokenizer, model

    print(f"Loading {GRANITE_MODEL_NAME} on {device}...")
    try:
        tokenizer = AutoTokenizer.from_pretrained(GRANITE_MODEL_NAME)
        model = AutoModelForCausalLM.from_pretrained(
            GRANITE_MODEL_NAME,
            device_map="auto",
            torch_dtype=torch.float16
        )
        model.eval()
        print(f"Successfully loaded model: {GRANITE_MODEL_NAME} locally!")
        print(f"Model is running on: {device}")
        return tokenizer, model
    except Exception as e:
        print(f"Error loading model {GRANITE_MODEL_NAME}: {e}")
        print("Possible reasons: Insufficient GPU RAM, model not found, or network issues during
download.")
        print("Consider trying a smaller model like 'ibm-granite/granite-3.0-2b-instruct' or upgrading
Colab runtime.")
        return None, None

# Try loading the actual model, if it fails, fallback to mock objects.
try:
    tokenizer, model = load_granite_model()
    if tokenizer is None or model is None:
        raise Exception("Model or tokenizer failed to load, falling back to mock.")
except Exception as e:
    print(f"INFO: Failed to load actual model ({e}). Using mock model for demonstration.")
    class MockModel:
        def generate(self, input_ids, max_new_tokens, temperature, top_p, do_sample,
pad_token_id, eos_token_id):
            text = self.tokenizer.decode(input_ids[0], skip_special_tokens=True)
            if "summarize" in text.lower():
                return self.tokenizer.encode("This is a mock summary of your text.")
            elif "refine" in text.lower():
                return self.tokenizer.encode("This is your mock refined text.")
            elif "meaning of" in text.lower():
                return self.tokenizer.encode("Mock meaning: A mock object is a simulated object.")
            elif "translate" in text.lower():
                return self.tokenizer.encode("Mock translation.")
            elif "quiz" in text.lower():
                mock_quiz_output = """
1. Question: What is the capital of France?
    A. Berlin
    B. Madrid
    C. Paris
    D. Rome

    Correct Answer: C
```

```
    Explanation: Paris is the capital and most populous city of France.

2. Question: Which planet is known as the Red Planet?
    A. Earth
    B. Mars
    C. Jupiter
    D. Venus

    Correct Answer: B
    Explanation: Mars is often referred to as the Red Planet because of its reddish appearance.

3. Question: What is the largest ocean on Earth?
    A. Atlantic Ocean
    B. Indian Ocean
    C. Arctic Ocean
    D. Pacific Ocean

    Correct Answer: D
    Explanation: The Pacific Ocean is the largest and deepest of Earth's five oceanic divisions.
"""
            return self.tokenizer.encode(mock_quiz_output)
        return self.tokenizer.encode("Hello! This is a mock response from Edu-Tutor AI.")

    tokenizer = AutoTokenizer.from_pretrained("gpt2") # Using a small, easily downloadable
tokenizer for mock
    model = MockModel()
    model.tokenizer = tokenizer # Attach tokenizer to mock model for encoding/decoding

# Ensure accelerator is prepared even for mock model (if using torch operations)
accelerator = Accelerator()
# Note: For the MockModel, accelerator.prepare might not be strictly necessary if it doesn't use
torch tensors internally
# but keeping it for consistency if you swap to a real model.
# If you get errors here with the mock model, you can remove this line for the mock setup.
# model, tokenizer = accelerator.prepare(model, tokenizer)


# --- Helper Functions for Model Inference ---

STREAM_DELAY = 0.005 # seconds per word

def generate_response(prompt, chat_history_for_model):
    if model is None or tokenizer is None:
        yield "Error: Model not loaded. Please check Colab runtime and GPU memory."
        return

    messages = []
    for human, ai in chat_history_for_model:
        messages.append({"role": "user", "content": human if human is not None else ""})
        messages.append({"role": "assistant", "content": ai if ai is not None else ""})
    messages.append({"role": "user", "content": prompt})
```

```
    input_text = tokenizer.apply_chat_template(messages, tokenize=False,
add_generation_prompt=True)
    input_ids = tokenizer(input_text, return_tensors="pt").to(device)

    current_response = ""
    try:
        input_token_len = input_ids.input_ids.shape[1]

        # Ensure pad_token_id and eos_token_id are set for generate
        if tokenizer.pad_token_id is None:
            tokenizer.pad_token_id = tokenizer.eos_token_id # Often EOS token is used as pad_token_id
        if tokenizer.eos_token_id is None: # Fallback if tokenizer doesn't provide one
             # For some models/tokenizers, you might manually set an arbitrary ID or handle it
differently
             # For gpt2, eos_token_id is typically 50256
            if hasattr(tokenizer, 'default_end_token_id'):
                tokenizer.eos_token_id = tokenizer.default_end_token_id
            elif tokenizer.vocab_size > 0:
                tokenizer.eos_token_id = tokenizer.vocab_size - 1 # Last token ID as a fallback


        full_output_tokens = model.generate(
            **input_ids,
            max_new_tokens=100,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.pad_token_id,
            eos_token_id=tokenizer.eos_token_id
        )
        full_response = tokenizer.decode(full_output_tokens[0][input_token_len:],
skip_special_tokens=True).strip()

        words = full_response.split(" ")
        for i in range(len(words)):
            current_response += words[i] + " "
            yield current_response
            time.sleep(STREAM_DELAY)

        return

    except Exception as e:
        yield f"Error during inference: {e}"

# Function to read text from PDF
def read_pdf_text(pdf_file_path):
    text = ""
    if pdf_file_path is None:
        return ""
    try:
        with open(pdf_file_path, 'rb') as file:
            reader = PyPDF2.PdfReader(file)
            for page_num in range(len(reader.pages)):
```

```
            page = reader.pages[page_num]
            text += page.extract_text()
        return text
    except Exception as e:
        return f"Error reading PDF: {e}"

# --- Quiz Parsing Function ---
def parse_quiz_text(raw_quiz_text):
    """
    Parses the raw quiz text generated by the LLM into a structured list of dictionaries.
    Each dictionary represents a question.
    """
    parsed_questions = []

    # Regex to find each question block
    # It looks for:
    # 1. An optional number (e.g., "1.")
    # 2. " Question:" followed by the question text
    # 3. Lines starting with A., B., C., D. for options
    # 4. "Correct Answer: [Letter]"
    # 5. "Explanation: [Text]"
    question_pattern = re.compile(
        r'(\d*\.?\s*Q(?:uestion)?:\s*(.*?)\n' # Optional number, "Question:", then question text
(group 2)
        r'\s*A\.\s*(.*?)\n' # Option A (group 3)
        r'\s*B\.\s*(.*?)\n' # Option B (group 4)
        r'\s*C\.\s*(.*?)\n' # Option C (group 5)
        r'\s*D\.\s*(.*?)\n' # Option D (group 6)
        r'\s*Correct Answer:\s*([A-D])\s*\n' # Correct Answer letter (group 7)
        r'\s*Explanation:\s*(.*?)(?=\n\d*\.?\s*Q(?:uestion)?:|\Z))', # Explanation (group 8),
lookahead for next question or end of string
        re.DOTALL # . matches newlines
    )

    matches = question_pattern.findall(raw_quiz_text)

    for match in matches:
        question_text = match[1].strip()
        options = {
            'A': match[2].strip(),
            'B': match[3].strip(),
            'C': match[4].strip(),
            'D': match[5].strip()
        }
        correct_answer = match[6].strip()
        explanation = match[7].strip()

        # Clean up question text if it has extra "Question:" or leading/trailing spaces
        # Also remove the leading number if present, as it's just for display initially
        question_text = re.sub(r'^\d+\.\s*Question:\s*', '', question_text,
flags=re.IGNORECASE).strip()
```

```
        parsed_questions.append({
            'question_text': question_text,
            'options': options,
            'correct_answer': correct_answer,
            'explanation': explanation
        })

    print(f"DEBUG: Parsed {len(parsed_questions)} questions.")
    return parsed_questions


# --- Quiz Generation and Scoring Functions ---

quiz_context = {
    "topic": "",
    "difficulty": "",
    "parsed_quiz": []
}


def generate_quiz_for_display(quiz_topic, num_quiz_questions_str, difficulty="medium",
pdf_file=None):
    # Initialize a list of gr.update objects for all expected outputs
    # The first element is for parsed_quiz_data_state, and should be gr.update()
    updates = [
        gr.update(), # 0: parsed_quiz_data_state (will be updated with actual data) - CORRECTED
        gr.update(visible=False, value=""), # 1: raw_quiz_output_display (clear and hide)
        gr.update(visible=False), # 2: submit_quiz_button
        gr.update(value="", visible=False), # 3: quiz_results_display
        gr.update(visible=False), # 4: start_new_quiz_button
        gr.update(visible=False) # 5: interactive_quiz_column - initially hidden for reset
    ]
    # Add updates for all possible question/radio pairs, initially hidden
    for i in range(MAX_QUIZ_QUESTIONS_UI):
        updates.append(gr.update(value="", visible=False)) # Q_md
        updates.append(gr.update(choices=[], value=None, visible=False)) # Q_radio

    # Yield initial reset state (important for clearing previous quiz display)
    yield tuple(updates)

    if model is None or tokenizer is None:
        print(f"DEBUG: Model or tokenizer not loaded at start of generate_quiz_for_display. Current
time: {time.time()}")
        updates[1] = gr.update(value="Error: Model not loaded. Please check Colab runtime and GPU
memory.", visible=True)
        yield tuple(updates)
        return

    try:
        num_quiz_questions = int(num_quiz_questions_str)
        if not (1 <= num_quiz_questions <= MAX_QUIZ_QUESTIONS_UI):
```

```
            raise ValueError(f"Number of questions must be between 1 and
{MAX_QUIZ_QUESTIONS_UI}.")
    except ValueError as e:
        print(f"DEBUG: Invalid number of questions: {e}. Current time: {time.time()}")
        updates[1] = gr.update(value=f"Invalid number of questions: {e}. Please enter a whole number
between 1 and {MAX_QUIZ_QUESTIONS_UI}.", visible=True)
        yield tuple(updates)
        return

    context_text = ""
    effective_topic = ""
    if pdf_file:
        context_text = read_pdf_text(pdf_file.name)
        if "Error reading PDF" in context_text:
            print(f"DEBUG: Error reading PDF: {context_text}. Current time: {time.time()}")
            updates[1] = gr.update(value=context_text, visible=True)
            yield tuple(updates)
            return
        if not context_text.strip():
            print(f"DEBUG: PDF file is empty or could not be read (empty context_text). Current time:
{time.time()}")
            updates[1] = gr.update(value="PDF file is empty or could not be read.", visible=True)
            yield tuple(updates)
            return
        effective_topic = f"PDF: {os.path.basename(pdf_file.name)}"
    elif quiz_topic and quiz_topic.strip():
        context_text = quiz_topic
        effective_topic = quiz_topic
    else:
        print(f"DEBUG: No topic or PDF provided. Current time: {time.time()}")
        updates[1] = gr.update(value="Please provide a topic for the quiz or upload a PDF.",
visible=True)
        yield tuple(updates)
        return

    if not context_text.strip():
        print(f"DEBUG: Final context_text is empty after processing. Current time: {time.time()}")
        updates[1] = gr.update(value="A valid topic or non-empty PDF content is required to generate
a quiz.", visible=True)
        yield tuple(updates)
        return

    difficulty_instruction = ""
    if difficulty == "easy":
        difficulty_instruction = "Make the questions relatively easy and straightforward."
    elif difficulty == "hard":
        difficulty_instruction = "Make the questions challenging and detailed, requiring deeper
understanding."
    else: # medium
        difficulty_instruction = "Make the questions of medium difficulty."
```

```
    quiz_prompt = f"""Generate {num_quiz_questions} multiple-choice questions about the following text or topic:
    {context_text}

    {difficulty_instruction}
    For each question, provide exactly 4 options (A, B, C, D), clearly state the single correct answer letter, and give an **EXTREMELY brief, 1-sentence explanation. Be very concise.**
    Focus on **very short questions** and **short options**. No introductory or concluding remarks. Just the questions.
    **IMPORTANT: Prepend each "Question:" with its number (e.g., "1. Question:", "2. Question:").**
    Format each question STRICTLY like this, with no extra text or numbering outside this pattern:

    [Number]. Question: [Short question text]
    A. [Option A]
    B. [Option B]
    C. [Option C]
    D. [Option D]

    Correct Answer: [Letter]
    Explanation: [Very brief explanation, 1 sentence max, no fluff]

    Example:
    1. Question: Capital of France?
    A. Berlin
    B. Madrid
    C. Paris
    D. Rome

    Correct Answer: C
    Explanation: Paris is the capital.
    """
    print(f"DEBUG: Quiz prompt length: {len(quiz_prompt)}. First 200 chars: {quiz_prompt[:200]}. Current time: {time.time()}")

    messages = [{"role": "user", "content": quiz_prompt}]
    input_text = tokenizer.apply_chat_template(messages, tokenize=False, add_generation_prompt=True)

    if not input_text.strip():
        print(f"DEBUG: Tokenizer applied chat template resulted in empty input_text. Current time: {time.time()}")
        updates[1] = gr.update(value="Error: Could not create a valid prompt for the quiz. This might happen if the topic or PDF content is too short or invalid.", visible=True)
        yield tuple(updates)
        return

    quiz_input_ids = None
    try:
        quiz_input_ids = tokenizer(input_text, return_tensors="pt").to(device)
        print(f"DEBUG: Tokenized input_ids shape: {quiz_input_ids.input_ids.shape}. Current time: {time.time()}")
```

```
    except Exception as e:
        print(f"DEBUG: Error tokenizing input for quiz: {e}. Current time: {time.time()}")
        updates[1] = gr.update(value=f"Error tokenizing input for quiz: {e}. Please ensure input text is valid.", visible=True)
        yield tuple(updates)
        return

    try:
        MAX_TOKENS_PER_QUIZ_QUESTION = 70
        quiz_max_new_tokens = num_quiz_questions * MAX_TOKENS_PER_QUIZ_QUESTION
        print(f"DEBUG: Generating with max_new_tokens={quiz_max_new_tokens}. Current time: {time.time()}")

        full_output_tokens = model.generate(
            **quiz_input_ids,
            max_new_tokens=quiz_max_new_tokens,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )
        raw_quiz_text = tokenizer.decode(full_output_tokens[0][quiz_input_ids.input_ids.shape[1]:], skip_special_tokens=True)

        # --- Temporarily show raw quiz text for debugging ---
        # You can set visible=False after testing
        updates[1] = gr.update(value=raw_quiz_text, visible=True, label="Raw Quiz Output (for debugging)")
        yield tuple(updates) # Yield this update so you can see the raw output while parsing happens

        print(f"DEBUG: Raw quiz text generated. Length: {len(raw_quiz_text)}. First 500 chars:\n{raw_quiz_text[:500]}. Current time: {time.time()}")

        parsed_quiz_data = parse_quiz_text(raw_quiz_text)
        print("DEBUG: Parsed quiz data:")
        print(json.dumps(parsed_quiz_data, indent=2))
        print(f"DEBUG: Number of questions parsed: {len(parsed_quiz_data)}. Current time: {time.time()}")

        if not parsed_quiz_data:
            print(f"DEBUG: No questions parsed from the generated text. Current time: {time.time()}")
            updates[1] = gr.update(value="Could not generate a valid quiz with the requested format. Please try again with a different topic or fewer questions.", visible=True)
            yield tuple(updates)
            return

        # Store for later use by scoring function
        quiz_context["topic"] = effective_topic
        quiz_context["difficulty"] = difficulty
        quiz_context["parsed_quiz"] = parsed_quiz_data

        # Update the initial components
        updates[0] = gr.update(value=parsed_quiz_data) # Update gr.State with the parsed data
```

```
        print(f"DEBUG: Value for parsed_quiz_data_state update: {len(parsed_quiz_data)} questions.")

        updates[1] = gr.update(visible=False, value="") # Hide the raw text output after successful parsing
        updates[2] = gr.update(visible=True) # Show the submit button
        updates[3] = gr.update(value="", visible=False) # Clear and hide previous results
        updates[4] = gr.update(visible=False) # Hide start new quiz button initially
        updates[5] = gr.update(visible=True) # Make the interactive quiz column visible here

        # Populate quiz questions and options.
        output_idx = 6 # Start index for question/radio pairs after the initial 6 common components
        for i, q in enumerate(parsed_quiz_data):
            if i < MAX_QUIZ_QUESTIONS_UI: # Ensure we don't exceed placeholder count
                question_label = f"{i+1}. {q['question_text']}"
                choices = [f"A. {q['options']['A']}", f"B. {q['options']['B']}", f"C. {q['options']['C']}", f"D. {q['options']['D']}"]

                # Update question Markdown and Radio button using gr.update
                updates[output_idx] = gr.update(value=question_label, visible=True)
                updates[output_idx + 1] = gr.update(choices=choices, value=None, label=f"Select your answer for Q{i+1}", visible=True)
            else:
                # This should ideally not happen if num_quiz_questions <= MAX_QUIZ_QUESTIONS_UI
                # But for safety, ensure any remaining placeholders are hidden.
                updates[output_idx] = gr.update(value="", visible=False)
                updates[output_idx + 1] = gr.update(value=None, visible=False, choices=[])
            output_idx += 2

        # Hide any remaining unused placeholder quiz elements
        for i in range(len(parsed_quiz_data), MAX_QUIZ_QUESTIONS_UI):
            updates[output_idx] = gr.update(value="", visible=False)
            updates[output_idx + 1] = gr.update(value=None, visible=False, choices=[])
            output_idx += 2

        print(f"DEBUG: Gradio UI updates prepared for interactive quiz. Current time: {time.time()}")
        yield tuple(updates) # Yield the populated quiz UI

    except Exception as e:
        print(f"DEBUG: Error during quiz generation inference or parsing: {e}. Current time: {time.time()}")
        updates[1] = gr.update(value=f"Error generating or parsing quiz: {e}. Please check Colab console for details.", visible=True)
        yield tuple(updates)


# Make sure quiz_context is defined globally at the top of your script, e.g.:
# quiz_context = {}

def submit_quiz(*user_answers_raw): # Removed parsed_quiz_data as an input argument
    """
    Scores the quiz based on user answers and provides detailed feedback.
```

```
    *user_answers_raw will be a tuple where each element is the selected option string for a question.
    """
    # --- DEBUGGING PRINTS: Access quiz_data from global quiz_context ---
    print(f"DEBUG: submit_quiz called.")

    # Get parsed quiz data directly from the global quiz_context
    parsed_quiz_data = quiz_context.get("parsed_quiz", [])

    print(f"DEBUG: Type of parsed_quiz_data (from quiz_context): {type(parsed_quiz_data)}")
    print(f"DEBUG: Content of parsed_quiz_data (from quiz_context): {parsed_quiz_data}")
    print(f"DEBUG: Length of parsed_quiz_data (from quiz_context): {len(parsed_quiz_data) if parsed_quiz_data is not None else 'None'}")
    # --- END DEBUGGING PRINTS ---

    # Prepare outputs list, matching the order of submit_quiz_outputs
    outputs = []

    # First, handle the common UI elements:
    outputs.append(gr.update(visible=False)) # submit_quiz_button (will be hidden)
    outputs.append(gr.update(value="", visible=False)) # quiz_results_display (cleared and hidden, will be updated)
    outputs.append(gr.update(visible=True)) # start_new_quiz_button (will be updated) - Always show this after submit
    outputs.append(gr.update(visible=False)) # interactive_quiz_column (will be hidden)

    # Initialize updates for all quiz question and radio button components
    # These must be included in the outputs tuple, even if they are hidden
    for _ in range(MAX_QUIZ_QUESTIONS_UI):
        outputs.append(gr.update(value="", visible=False)) # quiz_question_md[i] - No 'choices' here
        outputs.append(gr.update(value=None, visible=False, choices=[])) # quiz_options_radio[i] - important to clear choices too

    # --- Now check if the list retrieved from quiz_context is empty ---
    if not parsed_quiz_data: # Check if the list 'parsed_quiz_data' is empty
        print("DEBUG: parsed_quiz_data (from quiz_context) is empty. Returning 'No quiz data found' message.")
        outputs[1] = gr.update(value="No quiz data found. Please generate a quiz first.", visible=True)
        outputs[2] = gr.update(visible=True) # Show start new quiz button
        return tuple(outputs)

    score = 0
    feedback_markdown = "## Quiz Results\n\n"

    for i, question in enumerate(parsed_quiz_data): # Iterate directly over the list
        user_answer_display = "No answer provided"
        user_choice_letter = None

        if i < len(user_answers_raw) and user_answers_raw[i] is not None:
            user_answer_display = user_answers_raw[i]
            # Extract just the letter (A, B, C, D) from user_answer_raw (e.g., "A. Option Text")
            if isinstance(user_answers_raw[i], str) and len(user_answers_raw[i]) >= 1:
```

```
            user_choice_letter = user_answers_raw[i][0]

        correct_answer_letter = question['correct_answer']

        is_correct = (user_choice_letter == correct_answer_letter)
        if is_correct:
            score += 1
            feedback_markdown += f"✅ **Question {i+1}: Correct!**\n"
        else:
            feedback_markdown += f"❌ **Question {i+1}: Incorrect.**\n"

        feedback_markdown += f"**Your Answer:** {user_answer_display}\n"
        feedback_markdown += f"**Correct Answer:** {correct_answer_letter}.
{question['options'].get(correct_answer_letter, 'Option not found')}\n"
        feedback_markdown += f"**Explanation:** {question['explanation']}\n\n---\n\n"

    total_questions = len(parsed_quiz_data) # Get length directly from the list
    final_score_text = f"## Your Score: {score} out of {total_questions}\n\n"
    feedback_markdown = final_score_text + feedback_markdown

    # Record result to database - ensure score and total_questions are integers
    record_quiz_result(quiz_context["topic"], quiz_context["difficulty"], int(score),
int(total_questions))

    # Update the relevant output components in the 'outputs' list
    outputs[0] = gr.update(visible=False) # submit_quiz_button
    outputs[1] = gr.update(value=feedback_markdown, visible=True) # quiz_results_display
    outputs[2] = gr.update(visible=True) # start_new_quiz_button
    outputs[3] = gr.update(visible=False) # interactive_quiz_column (hide this after submission)

    # All other question/radio updates are already set to hidden/cleared by the initial `outputs` list
setup.

    return tuple(outputs)
def start_new_quiz_ui_reset():
    """Resets the quiz UI to its initial state, hiding quiz and results elements."""
    outputs = [
        gr.update(value="", visible=True), # quiz_topic_input
        gr.update(value="3", visible=True), # num_questions_input
        gr.update(value="medium", visible=True), # quiz_difficulty
        gr.update(value=None, visible=True), # pdf_upload_input
        gr.update(visible=True), # generate_quiz_button
        gr.update(value="", visible=False), # raw_quiz_output_display (ensure hidden, now a Textbox)
        gr.update(visible=False), # submit_quiz_button (hidden)
        gr.update(value="", visible=False), # quiz_results_display (hidden)
        gr.update(visible=False), # start_new_quiz_button (hidden)
        gr.update(visible=False) # interactive_quiz_column (hide this too)
    ]
    # Hide all question fields and reset their values
    for i in range(MAX_QUIZ_QUESTIONS_UI):
        outputs.extend([
            gr.update(value="", visible=False), # quiz_question_md[i] - No 'choices' here
```

```
        gr.update(value=None, visible=False, choices=[]) # quiz_options_radio[i] - important to clear choices too
    ])

  return tuple(outputs)


# --- Summarizer, Refiner, Word, Translate Functions (Unchanged - ensure they use gr.update for outputs) ---
def summarize_text(input_text, summary_length="short"):
  if model is None or tokenizer is None:
    yield gr.update(value="Error: Model not loaded. Please check Colab runtime and GPU memory.")
    return
  if not input_text:
    yield gr.update(value="Please provide text to summarize.")
    return

  if summary_length == "short":
    prompt_instruction = "Summarize the following text concisely:"
    max_output_tokens = 100
  elif summary_length == "detailed":
    prompt_instruction = "Provide a detailed summary of the following text:"
    max_output_tokens = 250
  else: # simple_explanation
    prompt_instruction = "Explain the following text in simple terms, suitable for a beginner:"
    max_output_tokens = 200

  summary_prompt = f"{prompt_instruction}\n\n{input_text}"

  messages = [{"role": "user", "content": summary_prompt}]
  input_text_tokenized = tokenizer.apply_chat_template(messages, tokenize=False, add_generation_prompt=True)
  summary_input_ids = tokenizer(input_text_tokenized, return_tensors="pt").to(device)

  current_summary = ""
  try:
    full_output_tokens = model.generate(
      **summary_input_ids,
      max_new_tokens=max_output_tokens,
      temperature=0.7,
      do_sample=True,
      pad_token_id=tokenizer.eos_token_id
    )
    summary_raw = tokenizer.decode(full_output_tokens[0][summary_input_ids.input_ids.shape[1]:], skip_special_tokens=True)

    words = summary_raw.split(" ")
    for i in range(len(words)):
      current_summary += words[i] + " "
      yield gr.update(value=current_summary) # Ensure outputs are gr.update
```

```
            time.sleep(STREAM_DELAY)

        return

    except Exception as e:
        yield gr.update(value=f"Error during summarization: {e}") # Ensure outputs are gr.update


def refine_text(input_text):
    if model is None or tokenizer is None:
        yield gr.update(value="Error: Model not loaded. Please check Colab runtime and GPU
memory.")
        return
    if not input_text:
        yield gr.update(value="Please provide text to refine.")
        return

    refine_prompt = f"""Review the following text for grammar, spelling, punctuation, and clarity.
Provide corrected sentences and suggest improvements for style and conciseness. Explain any
major changes.

    Original Text:
    {input_text}

    Refined Text:
    """

    messages = [{"role": "user", "content": refine_prompt}]
    input_text_tokenized = tokenizer.apply_chat_template(messages, tokenize=False,
add_generation_prompt=True)
    refine_input_ids = tokenizer(input_text_tokenized, return_tensors="pt").to(device)

    current_refined_text = ""
    try:
        full_output_tokens = model.generate(
            **refine_input_ids,
            max_new_tokens=400,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )
        refined_text_raw =
tokenizer.decode(full_output_tokens[0][refine_input_ids.input_ids.shape[1]:],
skip_special_tokens=True)

        words = refined_text_raw.split(" ")
        for i in range(len(words)):
            current_refined_text += words[i] + " "
            yield gr.update(value=current_refined_text) # Ensure outputs are gr.update
            time.sleep(STREAM_DELAY)

        return
```

```
    except Exception as e:
        yield gr.update(value=f"Error refining text: {e}") # Ensure outputs are gr.update

# Function for Word Meaning & Usage
def get_word_meaning_and_usage(word):
    if model is None or tokenizer is None:
        yield gr.update(value="Error: Model not loaded.")
        return
    if not word:
        yield gr.update(value="Please enter a word.")
        return

    prompt = f"""Provide the meaning of the word '{word}' and demonstrate its usage in two
different example sentences.
    Format your response clearly.

    Word: {word}
    Meaning:
    Example 1:
    Example 2:
    """
    messages = [{"role": "user", "content": prompt}]
    input_text_tokenized = tokenizer.apply_chat_template(messages, tokenize=False,
add_generation_prompt=True)
    input_ids = tokenizer(input_text_tokenized, return_tensors="pt").to(device)

    current_output = ""
    try:
        full_output_tokens = model.generate(
            **input_ids,
            max_new_tokens=150,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )
        raw_output = tokenizer.decode(full_output_tokens[0][input_ids.input_ids.shape[1]:],
skip_special_tokens=True)

        words = raw_output.split(" ")
        for i in range(len(words)):
            current_output += words[i] + " "
            yield gr.update(value=current_output) # Ensure outputs are gr.update
            time.sleep(STREAM_DELAY)
        return
    except Exception as e:
        yield gr.update(value=f"Error getting word meaning: {e}") # Ensure outputs are gr.update

# Function for Sentence Translation
def translate_sentence(sentence, target_language="Hindi"):
    if model is None or tokenizer is None:
        yield gr.update(value="Error: Model not loaded.")
```

```
        return
    if not sentence:
        yield gr.update(value="Please enter a sentence to translate.")
        return

    prompt = f"Translate the following English sentence into {target_language}:\n\nEnglish:
{sentence}\n{target_language}:"
    messages = [{"role": "user", "content": prompt}]
    input_text_tokenized = tokenizer.apply_chat_template(messages, tokenize=False,
add_generation_prompt=True)
    input_ids = tokenizer(input_text_tokenized, return_tensors="pt").to(device)

    current_output = ""
    try:
        full_output_tokens = model.generate(
            **input_ids,
            max_new_tokens=100,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )
        raw_output = tokenizer.decode(full_output_tokens[0][input_ids.input_ids.shape[1]:],
skip_special_tokens=True)

        words = raw_output.split(" ")
        for i in range(len(words)):
            current_output += words[i] + " "
            yield gr.update(value=current_output) # Ensure outputs are gr.update
            time.sleep(STREAM_DELAY)
        return
    except Exception as e:
        yield gr.update(value=f"Error translating sentence: {e}") # Ensure outputs are gr.update


# Initialize database
init_db()

# Define the chat functions OUTSIDE the Blocks for proper scoping
def user_message(user_message, history):
    # This now yields a gr.update to set the textbox value to empty
    # and updates the chatbot history to show the user's message immediately.
    return gr.update(value=""), history + [[user_message, None]]

def bot_response(history):
    if not history:
        # Yield an update for the chatbot to display an error
        yield gr.update(value=[["", "Error: Chat history is empty."]])
        return

    user_message_text = history[-1][0]

    # Generate the full response using the helper function
```

```
    full_bot_response = ""
    for chunk in generate_response(user_message_text, history[:-1]):
        full_bot_response = chunk
        history[-1][1] = full_bot_response
        yield history

def clear_chat():
    return None, None

# Make sure quiz_context = {} is defined globally at the top of your script, e.g.:
# quiz_context = {}

# --- Gradio UI ---
with gr.Blocks(title="Edu-Tutor AI", theme=gr.themes.Soft()) as demo:
    gr.Markdown("# <center>✨ Edu-Tutor AI: Your Personal Learning Assistant ✨</center>")
    gr.Markdown("Welcome! I'm Edu-Tutor, powered by IBM Granite. I can help you summarize text,
refine your writing, understand new words, translate sentences, and even generate quizzes!")

    with gr.Tab("Chat with Edu-Tutor"):
        chatbot = gr.Chatbot(label="Edu-Tutor Chat", bubble_full_width=False, layout="panel",
render_markdown=True, height=500)
        msg = gr.Textbox(label="Your message", placeholder="Ask me anything...", lines=2)
        with gr.Row():
            submit_button = gr.Button("Send Message")
            clear_button = gr.ClearButton([msg, chatbot])

        submit_button.click(user_message, [msg, chatbot], [msg, chatbot], queue=False).then(
            bot_response, chatbot, chatbot
        )
        msg.submit(user_message, [msg, chatbot], [msg, chatbot], queue=False).then(
            bot_response, chatbot, chatbot
        )
        clear_button.click(clear_chat, [], [msg, chatbot])

    with gr.Tab("Text Summarizer & Explainer"):
        with gr.Row():
            summary_input = gr.Textbox(label="Enter text to summarize/explain", lines=10,
placeholder="Paste your text here...")
        with gr.Row():
            summary_type = gr.Radio(["short", "detailed", "simple_explanation"], label="Summary
Type", value="short")
        with gr.Row():
            summarize_button = gr.Button("Generate Summary/Explanation")
        summary_output = gr.Markdown(label="Output")

        summarize_button.click(summarize_text, inputs=[summary_input, summary_type],
outputs=summary_output)

    with gr.Tab("Text Refiner"):
        refine_input = gr.Textbox(label="Enter text to refine", lines=10, placeholder="Paste your text
here...")
        refine_button = gr.Button("Refine Text")
```

```
        refine_output = gr.Markdown(label="Refined Text")

        refine_button.click(refine_text, inputs=refine_input, outputs=refine_output)

    with gr.Tab("Word Meaning & Usage"):
        word_input = gr.Textbox(label="Enter a word", placeholder="e.g., 'ubiquitous'")
        word_button = gr.Button("Get Meaning & Usage")
        word_output = gr.Markdown(label="Meaning and Examples")

        word_button.click(get_word_meaning_and_usage, inputs=word_input,
outputs=word_output)

    with gr.Tab("Sentence Translator"):
        translate_input = gr.Textbox(label="Enter an English sentence", placeholder="e.g., 'Hello, how
are you?'")
        target_language_dropdown = gr.Dropdown(["Hindi", "Spanish", "French", "German",
"Japanese", "Tamil"], label="Translate to", value="Hindi")
        translate_button = gr.Button("Translate Sentence")
        translate_output = gr.Markdown(label="Translated Sentence")

        translate_button.click(translate_sentence, inputs=[translate_input,
target_language_dropdown], outputs=translate_output)

    with gr.Tab("Generate Quiz"):
        gr.Markdown("## Generate a Multiple-Choice Quiz")
        gr.Markdown("Provide a topic or upload a PDF, and I'll generate a quiz for you.")
        with gr.Row():
            quiz_topic_input = gr.Textbox(label="Quiz Topic (e.g., 'Photosynthesis', 'World War II')",
lines=1, placeholder="Enter a topic or upload a PDF")
            num_questions_input = gr.Slider(minimum=1, maximum=MAX_QUIZ_QUESTIONS_UI,
value=3, step=1, label="Number of Questions")
            quiz_difficulty = gr.Radio(["easy", "medium", "hard"], label="Difficulty", value="medium")
        pdf_upload_input = gr.File(type="filepath", label="Upload PDF (Optional - overrides topic if
provided)", file_types=[".pdf"])
        generate_quiz_button = gr.Button("Generate Quiz")

        # This Textbox is for debugging raw output. It will be hidden in production.
        raw_quiz_output_display = gr.Textbox(label="Raw Quiz Output (for debugging)", lines=10,
visible=False)

        # State to store the parsed quiz data (This component is still needed, even if not directly an
input to submit_quiz)
        parsed_quiz_data_state = gr.State(value=[])

        # Column to hold the interactive quiz questions
        with gr.Column(visible=False) as interactive_quiz_column:
            quiz_question_md = []
            quiz_options_radio = []
            for i in range(MAX_QUIZ_QUESTIONS_UI):
                quiz_question_md.append(gr.Markdown(value="", visible=False))
                quiz_options_radio.append(gr.Radio(choices=[], value=None, label="", visible=False))
```

```
    submit_quiz_button = gr.Button("Submit Quiz", visible=False)

quiz_results_display = gr.Markdown(value="", visible=False, label="Quiz Results")
start_new_quiz_button = gr.Button("Start a New Quiz", visible=False)

# Define the outputs for generate_quiz_for_display
# Order MUST match the `updates` list in the function.
generate_quiz_outputs = [
    parsed_quiz_data_state, # State for quiz data
    raw_quiz_output_display, # Raw output for debugging
    submit_quiz_button, # Submit button visibility
    quiz_results_display, # Quiz results display (clear/hide)
    start_new_quiz_button, # Start new quiz button (hide)
    interactive_quiz_column # Interactive quiz column visibility
]
# Dynamically add all question/radio components to outputs
for i in range(MAX_QUIZ_QUESTIONS_UI):
    generate_quiz_outputs.append(quiz_question_md[i])
    generate_quiz_outputs.append(quiz_options_radio[i])

generate_quiz_button.click(
    generate_quiz_for_display,
    inputs=[quiz_topic_input, num_questions_input, quiz_difficulty, pdf_upload_input],
    outputs=generate_quiz_outputs
)

# Define the outputs for submit_quiz
# Order MUST match the `outputs` list in the function.
submit_quiz_outputs = [
    submit_quiz_button,
    quiz_results_display,
    start_new_quiz_button,
    interactive_quiz_column # Hide the quiz column after submission
]
# Dynamically add all question/radio components to outputs for hiding/clearing
for i in range(MAX_QUIZ_QUESTIONS_UI):
    submit_quiz_outputs.append(quiz_question_md[i])
    submit_quiz_outputs.append(quiz_options_radio[i])

submit_quiz_button.click(
    submit_quiz,
    inputs=list(quiz_options_radio), # Corrected: parsed_quiz_data_state removed
    outputs=submit_quiz_outputs
)

# Start New Quiz Button
start_new_quiz_button.click(
    start_new_quiz_ui_reset,
    inputs=[],
    outputs=[quiz_topic_input, num_questions_input, quiz_difficulty, pdf_upload_input,
             generate_quiz_button, raw_quiz_output_display, submit_quiz_button,
             quiz_results_display, start_new_quiz_button, interactive_quiz_column] +
```

```
                [item for sublist in zip(quiz_question_md, quiz_options_radio) for item in sublist]
        )

    with gr.Tab("Performance Dashboard"):
        gr.Markdown("## Your Learning Performance")
        gr.Markdown("Review your quiz history and identify weak areas.")

        with gr.Column():
            refresh_performance_button = gr.Button("Refresh Performance Data")
            weak_areas_display = gr.Markdown("### Weak Areas:\n\nNo data yet.")
            suggested_steps_display = gr.Markdown("### Suggested Next Steps:\n\nTake some quizzes
to get personalized suggestions!")
            quiz_history_dataframe = gr.DataFrame(
                headers=["Timestamp", "Topic", "Difficulty", "Score", "Total Questions"],
                row_count=5, # Show at least 5 rows
                col_count=(5, "fixed"), # 5 fixed columns
                wrap=True,
                label="Quiz History",
                visible=True
            )

        # Load initial data when the tab is selected (or when refreshed)
        demo.load(
            get_performance_insights,
            inputs=[],
            outputs=[quiz_history_dataframe, weak_areas_display, suggested_steps_display]
        )

        # Refresh button click
        refresh_performance_button.click(
            get_performance_insights,
            inputs=[],
            outputs=[quiz_history_dataframe, weak_areas_display, suggested_steps_display]
        )

demo.launch(debug=True, share=True)
```

Dataset Link

Not applicable, as the AI model (ibm-granite/granite-3.3-2b-instruct) is pre-trained and used for inference.

GitHub & Project Demo Link

- GitHub Repository: https://github.com/Likhitha-Gunneri/EduTutor-AI-Personalized-Learning-with-Generative-AI-and-LMS-Integration

- Project Demo: https://drive.google.com/file/d/1T3GJBJtv7A4EPl9aeLxu73ycRnGXFNO5/view?usp=sharing