SDLC Project Documentation — Educational AI Assistant

Version: 1.0

Generated: Detailed elaboration of architecture, implementation, setup, testing, and maintenance.

Team members

1.Gayathri. A
2.Reshma.S
3.Hulee.M

Executive Summary

This document provides a comprehensive, SDLC-aligned explanation of the Educational AI Assistant project whose source code you supplied. The assistant is a small web-based app that loads a HuggingFace causal language model, exposes two high-level features (Concept Explanation and Quiz Generation), and serves a lightweight UI via Gradio for interactive usage.

Table of Contents

1. Project Overview and Scope

Projectpurpose: TheEducationalAIAssistant helps learners understand technical and conceptual topics by producing clear, structured explanations and generating short quizzes for practice. It is intended for local or small-cloud deployment as an educational utility for students, tutors, and trainers. Scope: - Input: free-text concept/topic names supplied by users via a Gradio UI text box. - Output: generated textual explanations (multi-paragraph) and short quizzes (varied question types). - Model: A causal LLM hosted via HuggingFace model hub (example: ibm- granite/granite-3.2-2b-instruct). - Interface: Gradio-based single-page app with tabs for the two main features.

2. Functional Requirements

- FR1 — ConceptExplanation:Accept a concept string and return a multi-paragraph explanation with examples.
- FR2 — Quiz Generation: Generate a 5-question quiz for a given topic including answer keys (mix of MCQ, True/False, short answer).
- FR3 — Robust Input Handling: Reject empty inputs gracefully and provide helpful messages.
- FR4 — Model Loading: Load tokenizer + model with proper device management (CPU/GPU).
- FR5 — UI: Provide a responsive Gradio interface with separate tabs for explanation and quiz generation.
- FR6 — Extensibility: Code modules should be modular so new functionality (e.g., saving quizzes) can be added.

3. Non-Functional Requirements

-NFR1 — Usability: The UI must be simple enough for non-technical users. - NFR2 — Performance: Reasonable response latency on CPU (< 20s for a 2B model may be optimistic; GPU recommended for interactive use). - NFR3 — Reliability: The app must handle model loading failures and provide clear error messages. - NFR4 — Portability: Must run on local developer machines and cloud VMs with minimal changes. - NFR5 — Maintainability: Clear module boundaries and documentation for future contributors.

4. System Architecture and Components

High-levelcomponents: - Frontend:Gradioweb app. Responsible for collecting user input and displaying output. Simple tabbed interface (Concept explanation, Quiz generator). - Backend/Runtime: Python process that loads the language model and runs inference. Key modules: * Model loader and tokenizer initialization * Prompt construction and response generation * Feature-specific wrappers (explain, quiz) - Dependency Layer: HuggingFace Transformers + Torch for inference, PyPDF2 used (in README) but optional for PDF parsing, Gradio for UI. - Optional Persistence: A database (SQLite/Postgres) to save generated quizzes and user history. - Deployment Environments: Local machine, Docker container, or cloud VM (AWS/GCP/Azure) with GPU for better latency.

Component: Model Loader

The model loader's responsibilities: - Instantiate the tokenizer and model from a model repo (e.g., `ibm-granite/...`). - Ensure `pad_token` is set when missing (many causal LM tokenizers require a pad token). - Move the model to the correct device (CUDA when available). - Provide a simple `generate_response(prompt, max_length)` wrapper that handles tokenization, generation parameters (temperature, do_sample, max_length), and decoding. Implementation notes: - Watch GPU memory constraints — a 2B parameter model will require several gigabytes of GPU memory. - For CPU-only environments, set lower `max_length` and reduce sampling complexity.

Component: Feature Wrappers (Explain / Quiz)

Two small functions wrap the model for feature-level logic: - concept_explanation(concept): constructs a clear prompt that asks the model to explain a concept with examples and returns the model text. - quiz_generate(concept): constructs a prompt to produce a 5-question quiz with multiple formats and returns the model text. Good prompting practices: - Be explicit about the format you expect (number questions, mark answers at the end, desired difficulty). - If results are inconsistent, add a short instruction to not output extraneous commentary.

5. Data Flow and Sequence of Operations

Typical request flow (Concept explanation): 1. User enters a concept into Gradio UI and clicks 'Explain'. 2. Gradio sends input to backend function `concept_explanation`. 3. `concept_explanation` builds a prompt and calls `generate_response`. 4. `generate_response` tokenizes the prompt, executes `model.generate(...)`, then decodes and post-processes the text. 5. Gradio receives the returned string and displays it to the user. For Quiz generation: 1. User enters topic and clicks 'Generate Quiz'. 2. `quiz_generate` constructs a quiz-specific prompt and calls `generate_response`. 3. Returned text is shown in the UI. Optionally, the app can parse and structure the returned quiz into JSON for storage.

6. Detailed Implementation and Code Walkthrough

Below is a clear walkthrough of the primary source (README.md) you provided with the main code excerpt included. After the code excerpt we explain each block and provide guidance for improving robustness and maintainability.

Primary source code (SDLC.md) — snippet

```
!pip install transformers torch gradio PyPDF2 -q import
gradio as gr import torch from transformers import
AutoTokenizer, AutoModelForCausalLM import PyPDF2 import io
```

```
# Load model and tokenizer
model_name = "ibm-granite/granite-3.2-2b-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32,
    device_map="auto" if torch.cuda.is_available() else None
)

if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

def generate_response(prompt, max_length=1024):
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=512)

    if torch.cuda.is_available():
        inputs = {k: v.to(model.device) for k, v in inputs.items()}

    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_length=max_length,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )

    response = tokenizer.decode(outputs[0], skip_special_tokens=True)
    response = response.replace(prompt, "").strip()
    return response

def concept_explanation(concept):
    prompt=f"Explain the concept of{concept}in detail with examples:"
    return generate_response(prompt,max_length=800)
```

```
def quiz_generate(concept):
    prompt=f"Generate 5 quiz questions about {concept} with different question types(multiple choice,true/false,s
    return generate_response(prompt,max_length=1200)

# Create Gradio interface
with gr.Blocks() as app:
    gr.Markdown("# Educational AI Assistant")
    with gr.Tabs():
        with gr.TabItem("Concept explanation"):
          concept_input=gr.Textbox(label="Enter a concept",placeholder="e.g.,Machine Learning")
          explain_btn=gr.Button("explain")
          explanation_output=gr.Textbox(label="Explanation",lines=10)

          explain_btn.click(concept_explanation, inputs=concept_input, outputs=explanation_output)

        with gr.TabItem("Quiz Generator"):
          quiz_input=gr.Textbox(label="Enter a Topic",placeholder="e.g.,Machine Learning")
          quiz_btn=gr.Button("Generate Quiz")
          quiz_output=gr.Textbox(label="Quiz Questions&answers",lines=15)

          quiz_btn.click(quiz_generate, inputs=quiz_input, outputs=quiz_output)

app.launch(share=True)
```

Key function explanations

-generate_response(prompt, max_length=1024): * Tokenizes prompt with `tokenizer(..., truncation=True, max_length=512)`. * Moves tensors to model device if using CUDA. * Calls `model.generate(...)` with `temperature`, `do_sample`, and `pad_token_id` set. * Decodes outputs and strips the prompt from the returned text to keep only the model's completion. * Notes: Truncation needs careful tuning to avoid cutting user instruction; consider using a sliding window or chunking long inputs. - concept_explanation(concept): * Creates a prompt like "Explain the concept of {concept} ..." * Calls `generate_response` and returns the explanation. - quiz_generate(concept): * Creates a quiz prompt asking for 5 questions of mixed types and includes answers. * Returns the generated quiz; consider parsing to structured JSON if you want machine-readable output.

7. Setup, Installation and Deployment

Local setup (recommendedfor development): 1.Create a virtual environment: python -m venv venv source venv/bin/activate # on Windows: venv\Scripts\activate 2. Install dependencies: pip install -r requirements.txt (If you don't have a requirements.txt, run `pip install transformers torch gradio PyPDF2`). 3. Run the app: python main.py # or the script that launches the Gradio app Notes for GPU: - Install a torch build that matches your CUDA version. Example: `pip install torch --index-url https://download.pytorch.org/whl/cu118` for CUDA 11.8. Docker deployment (example): - Create a Dockerfile that installs Python, dependencies and exposes the Gradio port. Use an official Python base image and optionally NVIDIA CUDA images for GPU support. Cloud deployment notes: - Use a cloud VM with a GPU (g4/g5/gpu family) for interactive latency-sensitive use-cases. - Use a reverse proxy (NGINX) and optional HTTPS termination. Environment variables: - If model repo or API keys are private, store tokens in a `.env` file and load them securely.

8. Testing Strategy and Test Cases

Testinglevels: - Unit tests:Test generate_response with mocked model outputs; test prompt constructors; validate error conditions. - Integration tests: Run the Gradio functions end-to-end with a small mock model or a lightweight deterministic model. - Manual tests: Validate UI flows for empty input, long input, and incorrect input types. Sample test cases: 1. Empty concept input -> expected: user-friendly error message. 2. Short concept (e.g., 'DNS') -> expected: concise explanation with at least one example. 3. Long concept string (>1024 tokens) -> expected: predictable behavior (truncate or ask user to shorten). 4. Quiz generation formatting -> expected: exactly 5 questions with answers at the end. 5. Model down/unavailable -> expected: graceful message & retry instructions. Automation tips: - Use pytest for unit tests and GitHub Actions for CI runs. - Mock heavy model calls in CI to reduce resource usage.

9. Security, Privacy and Compliance Considerations

- Secrets management: Keep model access tokens, API keys, and credentials out of the code repository. Use environment variables or secrets management services. - Rate limiting: If the app is made public, put limits to avoid abuse and unexpected cost or exhaustion. - Data retention: Decide whether to store user queries or outputs. If storing, anonymize and document retention policy. - Input validation: Sanitize user-provided file uploads (if added) and limit file sizes. - Model output trust: LLMs can hallucinate; present a disclaimer and encourage verification for critical information.

10. Performance, Scaling and Optimization

- GPU inference: Use CUDA-enabled GPUs to reduce latency. Use model parallelism or quantized weights for larger models. - Batch requests: If many requests are expected, batch them where possible or use an inference server (like TorchServe, Triton, or an LLM-hosting solution). - Caching: Cache repeated prompts and responses for deterministic or frequently asked concepts. - Model size tradeoffs: Smaller distilled models reduce latency but may lower quality. - Monitoring: Track response latency, error rates, and resource utilization.

11. Maintenance, Monitoring and Logging

- Logging: Capture errors, prompt/response lengths, and user actions (with privacy considerations). - Alerts: Configure alerts for high error rates or model OOMs (out-of-memory). - Upgrades: Keep dependencies updated and pin versions in requirements.txt to ensure reproducible environments. - Documentation: Maintain README, changelog, and contribution guidelines for new developers.

12. Future Enhancements

- Structured outputs: Have the model return JSON for quizzes (question objects with options and correct answer indices). - Persistence: Save quizzes and user interactions in a database to track progress. - Multi-modal I/O: Add voice input/output, and support for PDF/CSV ingestion to generate topic-specific quizzes and summaries. - Admin interface: Allow teachers to curate or edit generated content before publishing to learners. - Metrics dashboard: Track adoption, common queries, and average comprehension scores (if quizzes are answered and stored).

13. Appendix: Sample Prompts and Outputs

1) Prompt (concept_explanation): "Explain the concept of Convolutional Neural Networks in simple terms, provide a small example and one application." Expected output: A short multi-paragraph explanation describing CNN layers, intuitive example (image filter), and an application like image classification. 2) Prompt (quiz_generate): "Generate 5 quiz questions about Python list comprehensions with varying types (MCQ, T/F, short answer). Provide the answers at the end." Expected output: 5 numbered questions, options for MCQs, and answers listed clearly at the end. 3) Edge-case prompt: Empty string should return: "Please enter a concept or topic to explain."

SDLC Source code

```
!pip install transformers torch gradio PyPDF2 -q import
gradio as gr import torch from transformers import
AutoTokenizer, AutoModelForCausalLM import PyPDF2 import io


# Load model and tokenizer
model_name = "ibm-granite/granite-3.2-2b-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32,
    device_map="auto" if torch.cuda.is_available() else None
)

if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

def generate_response(prompt, max_length=1024):
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=512)

    if torch.cuda.is_available():
        inputs = {k: v.to(model.device) for k, v in inputs.items()}

    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_length=max_length,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )

    response = tokenizer.decode(outputs[0], skip_special_tokens=True)
    response = response.replace(prompt, "").strip()
    return response

def concept_explanation(concept):
    prompt=f"Explain the concept of{concept}in detail with examples:"
    return generate_response(prompt,max_length=800)

def quiz_generate(concept):
    prompt=f"Generate 5 quiz questions about {concept} with different question types(multiple choice,true/false,s
    return generate_response(prompt,max_length=1200)

# Create Gradio interface
with gr.Blocks() as app:
    gr.Markdown("# Educational AI Assistant")
    with gr.Tabs():
        with gr.TabItem("Concept explanation"):
          concept_input=gr.Textbox(label="Enter a concept",placeholder="e.g.,Machine Learning")
          explain_btn=gr.Button("explain")
          explanation_output=gr.Textbox(label="Explanation",lines=10)

          explain_btn.click(concept_explanation, inputs=concept_input, outputs=explanation_output)

        with gr.TabItem("Quiz Generator"):
          quiz_input=gr.Textbox(label="Enter a Topic",placeholder="e.g.,Machine Learning")
          quiz_btn=gr.Button("Generate Quiz")
          quiz_output=gr.Textbox(label="Quiz_Questions&answers",lines=15)

          quiz_btn.click(quiz_generate, inputs=quiz_input, outputs=quiz_output)

app.launch(share=True)
```