Project Report:
Trifusion – AI-Enabled Recording Assistant for Transcription, Summarization, Q&A and Quiz Generation From classroom Lectures.

Introduction

At a time when university and college classrooms are churning out more knowledge nuggets than can ever be actually learned due to the painstaking efforts to record, transcribe, compile knowledge into create study materials. The AI-enabled classroom Recording Assistant, as described, solves this by capturing (record audio) classroom instruction for converting, in real-time, into fully-interactive study material.

These and the other features offered in a Streamlit interface were developed to frustrate effort, to make engagement active and to capture classroom knowledge - that is in a format that is easy to find, search and reuse.

Problem Statement 4

AI-Powered Interactive Learning Assistant for Classrooms:

- Build a Multimodal AI assistant for classrooms to dynamically answer queries using text, voice, and visuals while improving student engagement with personalized responses.

Our Objective

The AI-enabled recording assistant was developed for one purpose, to establish a bulk writing system that gives learners an efficient and automated way to record a classroom-like lecture and combine that classroom-like lecture into usable study material.

- **Record Live Lectures** - the intention is to facilitate utilising any notes from the lecture, that means there is no longer a need to be stuck in a recording studio or listen to a previously prepared audio which has to be stored in a separate folder from the recording to notes.
- **From live lecturing**,. the audible speech transforms it to readable text
- **AI** will be able to summarize, condense and shorten text into more readable summaries.
- **AI** will assimilate the content and construct multiple-choice quizzes as part of the revision process.

- **Interactive Questions and Answers Chat** - this will enable users to ask adaptive general questions corresponding to the lecture's topic and obtain an effective corresponding instant detailed answer.

In the end, Trifusion hopes ultimately gives learners and educators the time to not even consider the processes to record and assemble their content, and instead concentrate on and listen to what is about to be discussed during their lecture and focus on everything associated with exploring, learning, or practicing the content.

Solution Overview

Trifusion integrates AI's, and Automatic Speech Recognition + Summarization and automated models in a single monolithic (single) Streamlit app. This singular workflow has allowed for automated transcripts, summaries, exercises (quizzes), and Q&A workflow (from the classes) or to turn raw audio into structured, interactive study material with little manual effort.

Methodology

Iterative and Modular Development

Instead of developing everything, the project used an iterative and feedback loop development cycle:

Stepwise modular design:

- Audio Capture & Input Processing
- Transcription
- Summarization
- Generative AI (Q & A and Quiz)
- Streamlit User Interface

Build and iterate through each module:

- Begin by prototyping each module at a time (almost exclusively for example, transcribe alone).
- Test each module for accuracy, speed, and usability.Collect team feedback after each phase.

Sequential integration:

- Once each module is stable on its own, integrate the modules into one Streamlit interface.
- Introduce state management (i.e. st.session_state) to allow data to flow across tabs.
- Test for seamless integration and regular usability testing.

Feedback loop:

- Real world testing each time allowed real world analytic usability problems to come to the surface.
- The team created their models and selections, chunk sizes, and caching and the user interface as best as possible in order to maximize performance.

Outcome:
A lightweight and usable classroom assistant based on good practices of iterative approaches of design and real world testing

Architecture

1. **Audio Capture & Input Processing:**
   - **1.1. Live Recording:** utilizes the PvRecorder library that will capture the lecture audio as live via the microphone used by the user. The files are .wav files are timestamped (e. g. recording_20250712_123456.wav) when saved in the recordings subfolder.

```
# Code path: record_audio() function in the main app
def record_audio(device_index):
    recorder = PvRecorder(device_index=device_index,
frame_length=512)
    recorder.start()
    st.info("Recording... click Stop to finish.")
    frames = []
    stop = st.button("Stop Recording")
    if stop:
        recorder.stop()
        frames = recorder.read_all()
        audio_data = np.array(frames, dtype=np.int16)
        filename =
f"recording_{datetime.now().strftime('%Y%m%d_%H%M%S')}.wav"
        filepath = os.path.join(SAVE_DIR, filename)
        sf.write(filepath, audio_data, samplerate=16000)
        return filepath
    return None
```

   - **1.2. File Upload**: allows the user to upload a pre-existing recording of the lecture which must necessarily be in .wav or .mp3 formats and gives the user an added feature to gain consistency in the preauditory processing regardless of pre-recorded or live.

```
# Code path: file upload handling section in "Record & Transcribe"
page
uploaded = st.file_uploader("select or upload audio file",
type=["wav", "mp3"])
if uploaded:
    filename =
f"uploaded_{datetime.now().strftime('%Y%m%d_%H%M%S')}.wav"
    path = os.path.join(SAVE_DIR, filename)
    with open(path, "wb") as f:
        f.write(uploaded.read())
    st.session_state.audio_path = path
    st.success("Uploaded successfully!")
```

- The process includes dual input, providing a way for the user to capture the live lectures or offline lectures that were pre-recorded lectures.
- Both methods save files in:

```
recordings/
    recording_20250712_123456.wav
    uploaded_20250712_124501.wav
```

Streamlit UI:

2. Transcription

2.1. Feature Extraction: Audio files are sliced into ~30-second segments and converted into input features using WhisperProcessor (from Hugging Face).

```
# Code path: transcribe_with_encoder_decoder()
input_features = transcriber_processor(
    chunk, sampling_rate=16000, return_tensors="pt"
).input_features
```

2.2. Encoding(Openvino Optimized): The Whisper encoder takes the features and provides hidden states (vector representations) after being accelerated with OpenVINO to reduce latency and therefore achieve faster real-time transcription.

```
# Code path: transcribe_with_encoder_decoder()
encoder_output_np = encoder_model(
    inputs={"input_features": input_features.numpy()}
)[encoder_model.outputs[0]]
encoder_hidden_states = torch.tensor(encoder_output_np,
dtype=torch.float32)
```

2.3. Decoding: The WhisperForConditionalGeneration decoder takes the hidden states and forced decoder IDs (for language and task) and produces text tokens.

```
# Code path: transcribe_with_encoder_decoder()
forced_decoder_ids =
transcriber_processor.get_decoder_prompt_ids(language="en",
task="transcribe")
generated_ids = decoder_model.generate(

encoder_outputs=BaseModelOutput(last_hidden_state=encoder_hidden_sta
tes),
    forced_decoder_ids=forced_decoder_ids,
    max_length=448,
    do_sample=False
)
```

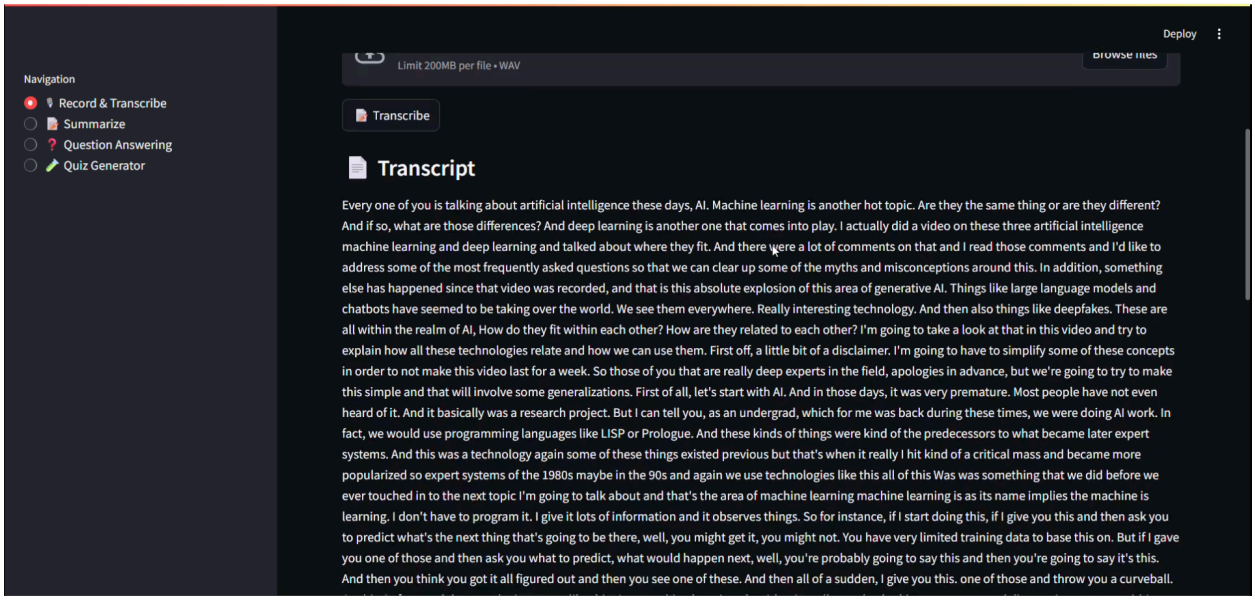
2.4. Reconstruction: The text tokens are decoded into human-readable text using batch_decode. Then, the chunks are combined in order to create the final transcript.

```
# Code path: transcribe_with_encoder_decoder()
transcription = transcriber_processor.batch_decode(generated_ids,
skip_special_tokens=True)[0]
transcriptions.append(transcription)
```

- This split encoder–decoder architecture may create a scalable solution for long lectures while providing reasonable accuracy.
- The final transcript is:

```
return " ".join(transcriptions)
```

Streamlit UI:



3. Summarization

- The transcript will be broken up into chunks (≤1024 tokens) to fit in model input size.

```
# Code path: summarize_text() function
chunks = [text[i:i+max_chunk_length] for i in range(0, len(text),
max_chunk_length)]
```

- Each chunk will be summarized independently with facebook/bart-large-cnn, a transformer model that has had success in the task of abstractive summarization.

```
# Code path: summarize_text() function
```

```
input_ids = summary_tokenizer(chunk, return_tensors="pt",
truncation=True, max_length=1024).input_ids
output = summary_model.generate(input_ids, max_length=150)
summary = summary_tokenizer.decode(output[0],
skip_special_tokens=True)
```

- The individual partial summaries will be put together for a summary of the lecture.

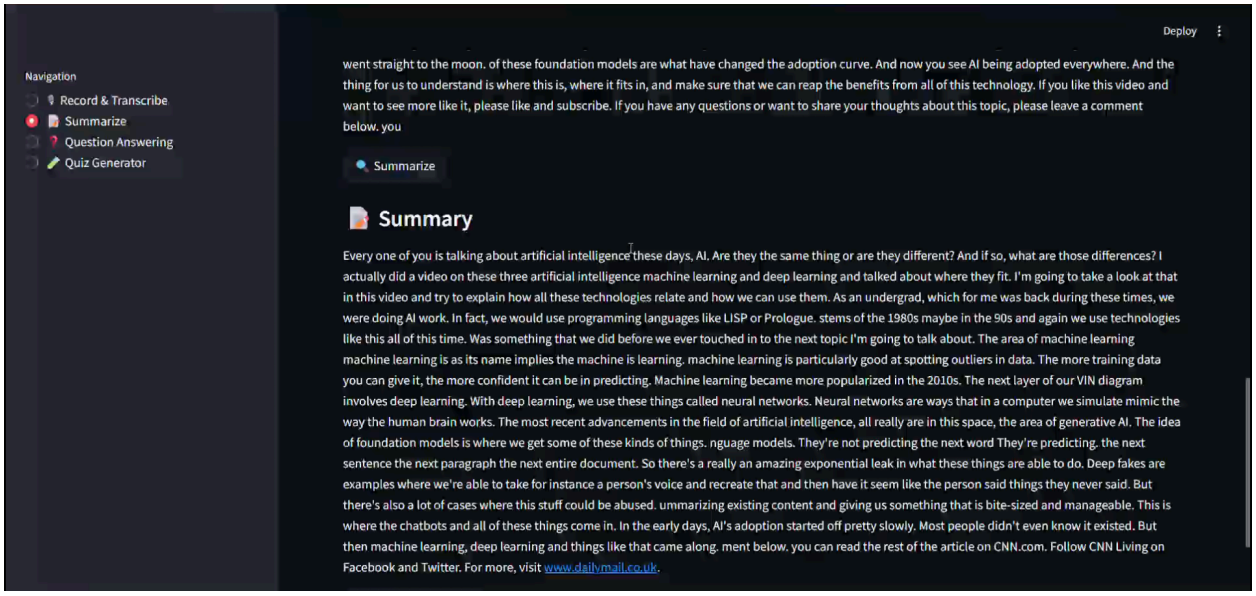
- This chunking process allows us to be sure that the summary is still meaningful and coherent even for lectures that are long.

- Partial summaries will be combined for a single summary for the whole lecture that is coherent. This chunking makes the final summary meaningful and coherent even for long lectures.

```
# Code path: summarize_text() function
return " ".join(summaries)
```

```
def summarize_text(text, max_chunk_length=1024):
    chunks = [text[i:i+max_chunk_length] for i in range(0, len(text),
max_chunk_length)]
    summaries = []
    for chunk in chunks:
        input_ids = summary_tokenizer(chunk, return_tensors="pt",
truncation=True, max_length=1024).input_ids
        output = summary_model.generate(input_ids, max_length=150)
        summary = summary_tokenizer.decode(output[0],
skip_special_tokens=True)
        summaries.append(summary)
    return " ".join(summaries)
```

Streamlit UI:



4. Generative AI (Q&A and Quiz)

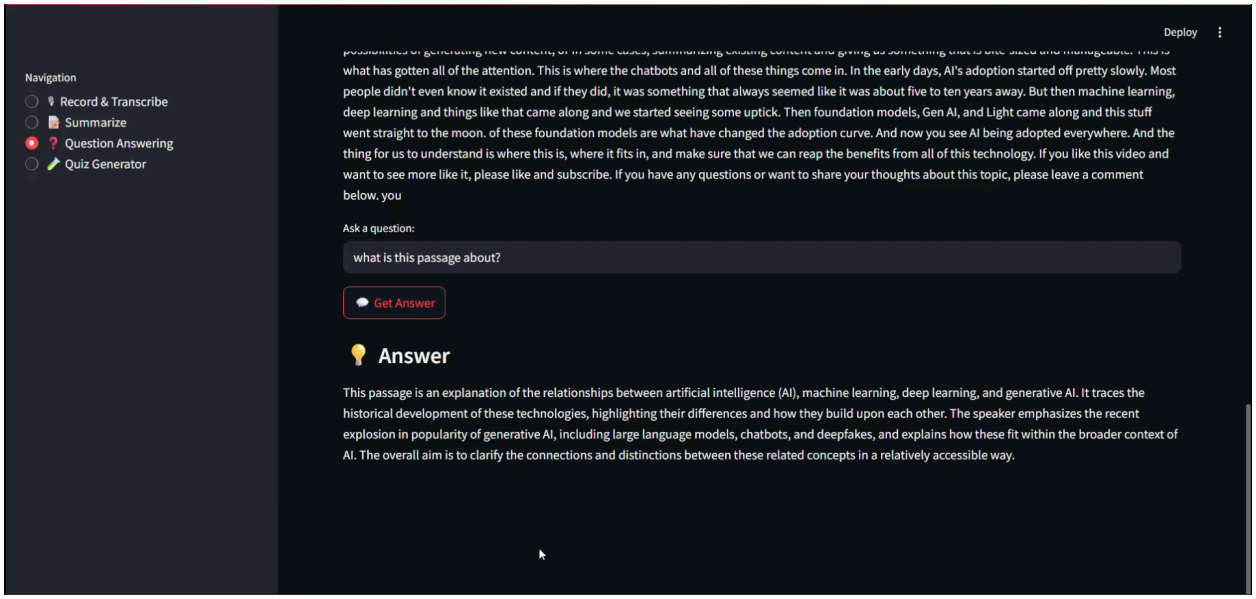
Q&A CHAT

- Users can type any question about the lecture.
- The system integrates:the full transcript, the user's question
- and sends this prompt to the gemini API, which uses the Gemini model (gemini-1.5-flash) to generate a specific, contextual response.

```
if page == "❓ Question Answering" and "transcribed_text" in
st.session_state:
    st.write("### 📄 Transcript")
    st.write(st.session_state.transcribed_text)

    q = st.text_input("Ask a question:")
    if st.button("💬 Get Answer") and q:
        with st.spinner("Gemini thinking..."):
            prompt = f"""Based on the transcript below, answer this
question:

Transcript:
{st.session_state.transcribed_text}

Question: {q}"""
```

```
            answer = gemini_answer(prompt)
            st.subheader("💡 Answer")
            st.write(answer)
```

Streamlit UI:



Quiz Generation

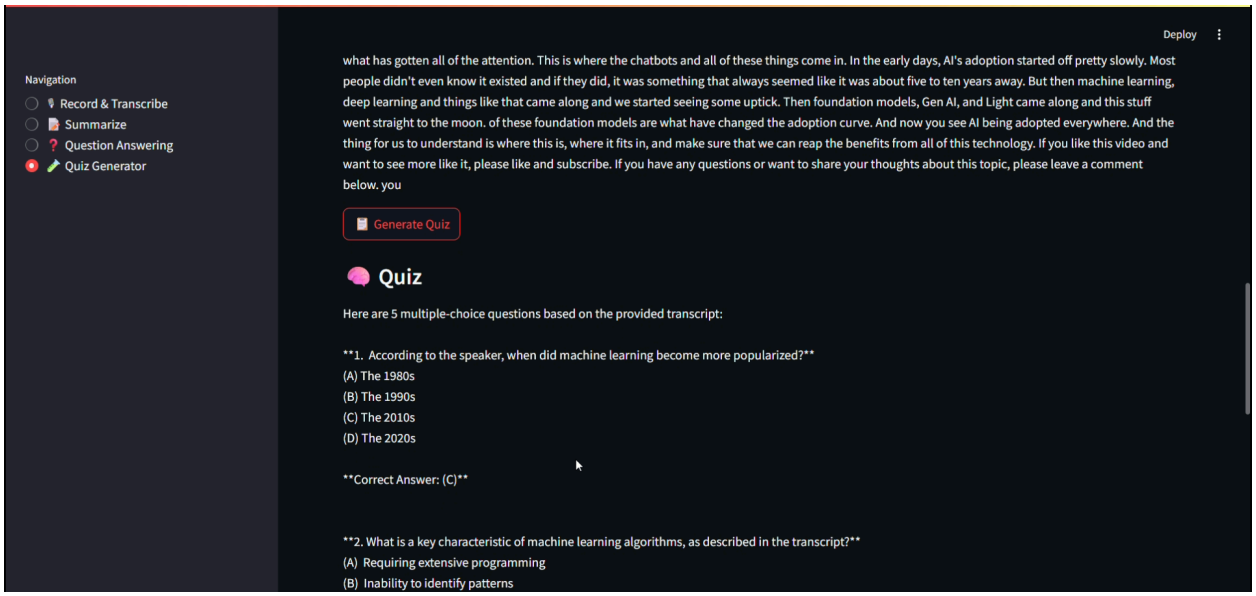
- The system forwards the complete transcript, to the same API, with a different prompt:
- "Ask Gemini to produce 5 multiple-choice questions (with four answer options), and to indicate the correct answer."

```
if page == "🧪 Quiz Generator" and "transcribed_text" in st.session_state:
    st.write("### 📄 Transcript")
    st.write(st.session_state.transcribed_text)

    if st.button("📋 Generate Quiz"):
        with st.spinner("Generating quiz with Gemini..."):
            prompt = f"""Generate 5 multiple-choice quiz questions with 4
options each (A-D) based on the following transcript.
Mention the correct answer after each question.

Transcript:
```

```
{st.session_state.transcribed_text}
"""
            quiz = gemini_answer(prompt)
            st.session_state.quiz = quiz
        st.subheader("🧠 Quiz")
        st.text(quiz)
        st.download_button("⬇ Export Quiz", quiz, file_name="quiz.txt")
```

- The logic is easy in the app; The complexity is hidden in the Gemini LLM hosted by GOOGLE GEMINI.
- It keeps the system light, while at the same time is able to dynamically produce extensive, adaptive educational content.

Streamlit UI:



5. Streamlit User Interface

Users navigate through four distinct tabs:

- **Record & Transcribe** - Users can record live audio or upload – then transcribe and download corresponding transcript.
- **Summarize -** Users can summarize the transcript and download it.
- **Contextual Questions -** Users can ask custom questions and receive corresponding answers.
- **Quiz Builder -** Users can create multiple choice quizzes and export them.

Model loading & caching

- To ensure system responsiveness and not create any delays with each user interaction, all of the models load once (on startup), and the models are stored in cache for re-use.
- **Why cache?**

  Loading large transformer models (for example, Whisper, BART...) is time-consuming.

  Accessing the loaded models with each user action (transcription, summarization...) is a lot quicker.

- The application will not download or reinitialize once for every button click
- It uses Streamlit's **@st.cache_resource** decorator, which guarantees that each model is only loaded once per session and stored in memory.
- The cache will be automatically refreshed when the code or environment changes.

Load Whisper encoder (OpenVINO):

```
@st.cache_resource
def load_openvino_encoder():
    core = Core()
    encoder =
core.compile_model("whisper-small-openvino/openvino_encoder_model.xml"
, "CPU")
    processor =
WhisperProcessor.from_pretrained("openai/whisper-small")
    return encoder, processor
```

Load Whisper decoder (Hugging Face):

```
@st.cache_resource
def load_hf_decoder():
    return
WhisperForConditionalGeneration.from_pretrained("openai/whisper-small
")
```

Load BART summarizer:

```
@st.cache_resource
def setup_summarizer():
    tokenizer =
BartTokenizer.from_pretrained("facebook/bart-large-cnn")
    model =
```

```
BartForConditionalGeneration.from_pretrained("facebook/bart-large-cnn
")
    return tokenizer, model
```

- **State management -** Uses **st.session_state** to retain transcript, summary and quiz data across tabs for easy transitions between them.
- **Local + cloud processing -** Transcription and summarization occur locally **(OpenVINO + Hugging Face)** while Quiz and Question Answering leverage cloud API **(GOOGLE GEMINI + Gemini)**.
- **File management -** All recordings and produced text documents are archived in a dedicated folder **(recordings)** for ease of access.
- **Development Approach -** Step-by-step wayfinder delivers audio content with ability to record, transcribe, summarize, ask contextual questions, generate quizzes **(including multiple choice)** without need for technical software expertise.

Technologies Used

List and briefly describe the main libraries, frameworks, APIs, and tools that were used in the project.

Example:

- **Streamlit:** for the interactive web app interface.
- **OpenVINO Runtime:** for accelerating Whisper encoder inference.
- **Hugging Face Transformers:** for Whisper decoder & summarization models.
- **PvRecorder:** for recording live audio from a microphone.
- **Librosa:** for processing and chunking audio.
- **GOOGLE GEMINI API (Gemini model):** for cloud-based Q&A and quiz generation.
- **SoundFile & NumPy:** for saving and managing audio data.

Team Members

ROLE	NAME	EMAIL	CONTRIBUTIONS
TEAM LEADER	A.Pardha Saye	Pardha Saye A 22…	App flow, audio logic, integration , testing documentation and Q&A generation openvino conversion

TEAM MEMBER	G.Siva Manikanta	Gudla Siva Manik…	recording and transcription quiz gen logic and documentation
TEAM MEMBER	N.Sai Siddharadha	Narayanapurapu …	Gemini integration, Q&A and quiz logic and ui and documentation
MENTOR	Dr Kanaka Raghu Sreerama	skatraga@gitam.edu	Guidance and Support

References

Streamlit - https://docs.streamlit.io/
OpenVINO IR format - https://docs.openvino.ai/2025/documentation/openvino-ir-format.html
OpenVINO™ Toolkit documentation - https://docs.openvino.ai/archives/index.html
Hugging Face Transformers documentation - https://huggingface.co/docs/transformers/index
PvRecorder GitHub repository - http://github.com/Picovoice/pvrecorder
Torch documentation - https://docs.pytorch.org/docs/stable/index.html