# Project Title

Software Development Life Cycle (SDLC)

# Project Documentation

## 1. Introduction

• Project title : Software Development Life Cycle (SDLC)  
• Team member : Harini.M  
• Team member : Lubna lutfiyya.I  
• Team member : Jothi.E  
• Team member : Harini.C  
  
The Software Development Life Cycle (SDLC) is a systematic process for building software that ensures quality and correctness. In this project, AI has been applied to automate two core phases of SDLC: Requirements Analysis and Code Generation. By leveraging Large Language Models (LLMs), our system reads requirements, organizes them into categories, and generates code snippets.

## 2. Project Overview

Purpose

The purpose of this project is to develop an AI-driven platform that simplifies requirement management, policy summarization, forecasting, and anomaly detection while promoting eco-friendly practices. The system is designed with a conversational interface and multimodal input support, making it easy for both technical and non-technical users to interact with data and generate actionable insights.

Features

- Conversational Interface: AI-powered chatbot for natural language interaction.

- Policy Summarization: Generates concise summaries of lengthy policy documents.

- Resource Forecasting: Predicts demand and utilization of resources.

- Eco-Tip Generator: Provides personalized eco-friendly suggestions.

- Citizen Feedback Loop: Collects feedback and suggestions from users/citizens.

- KPI Forecasting: Forecasts key performance indicators for proactive decision-making.

- Anomaly Detection: Identifies unusual or irregular data patterns.

- Multimodal Input Support: Accepts text, speech, images, and PDF inputs.

- Requirement Analysis: Categorizes requirements into functional, non-functional, and technical specifications.

- Code Generation: Auto-generates code in multiple languages (Python, Java, JavaScript, C++, C#, PHP, Go, Rust).

- PDF Support: Extracts requirements directly from uploaded PDF documents.

- User Interface (Streamlit / Gradio): Clean two-tab interface for easy navigation.

Requirement Analysis

- Functional Requirements:

- Conversational queries

- Policy summarization

- Forecasting (resources, KPIs)

- Code generation

- Eco-tip generation

- Non-Functional Requirements:

- Usability

- Performance

- Security

- Scalability

- Technical Requirements:

- AI/ML model integration

- PDF parsing libraries

- Streamlit/Gradio-based UI

- Multi-language code support

Code Generation

- Supports multiple languages:

- Python, Java, JavaScript, C++, C#, PHP, Go, Rust

- Generates boilerplate/sample code based on extracted requirements.

PDF Support

- Direct upload of requirement documents in PDF format.

- Extracts and categorizes requirements automatically.

- Displays results in a structured format.

User Interface

- Tab 1: Requirement Analysis

- Displays extracted and categorized requirements.

- Tab 2: Project Tools

- Forecasting dashboards

- Anomaly detection insights

- Eco-tip suggestions

- Code generation options

## 3. Architecture

The architecture of this project is divided into multiple layers:  
  
Frontend (Gradio):  
• An interactive UI with tabs for Requirement Analysis and Code Generation.  
  
Backend (Python + Transformers):  
• Handles requirement analysis and code generation using IBM Granite LLM.  
  
LLM Integration (IBM Granite):  
• Granite LLM model is used for natural language understanding, text classification, and generating code.  
  
PDF Handling (PyPDF2):  
• Extracts and processes text from uploaded requirement documents.  
  
Core Functions:  
• generate\_response(): Generates responses from the LLM.  
• extract\_text\_from\_pdf(): Extracts content from PDFs.  
• requirement\_analysis(): Analyzes and classifies requirements.  
• code\_generation(): Generates code in selected programming language.

## 4. Setup Instructions

Prerequisites:  
o Python 3.9 or later  
o Libraries: torch, transformers, gradio, PyPDF2  
o Internet access for downloading model weights  
  
Installation Process:  
1. Install dependencies using pip:  
 pip install torch transformers gradio PyPDF2  
2. Run the script in Google Colab or local environment.  
3. Launch the Gradio app with app.launch(share=True).  
4. Access the public link generated by Gradio to use the interface.

## 5. Folder Structure

app/ – Contains backend logic (analysis, code generation)  
ui/ – Contains Gradio interface design  
sdlc\_app.py – Main program script  
models/ – Placeholder for model handling scripts

## 6. Running the Application

➢ Step 1: Launch the Python script in Colab or locally.  
➢ Step 2: Gradio dashboard will open in browser.  
➢ Step 3: Navigate between Code Analysis and Code Generation tabs.  
➢ Step 4: Upload PDF or input requirements → Results appear as structured categories.  
➢ Step 5: Enter requirement description and select programming language → Generated code snippet appears.  
➢ Step 6: Copy results for documentation or prototype development.

## 7. API Documentation

The current version runs via Gradio and does not expose REST APIs. However, the backend functions can be extended into endpoints such as:  
• POST /analyze – Analyze requirements  
• POST /generate – Generate code in selected language  
• POST /upload – Upload and process PDF documents

## 8. Authentication

The system is open for demonstration. Future deployments can include:  
• Token-based authentication (JWT)  
• Role-based access (admin, user)  
• OAuth2 authentication  
• User session management and history tracking

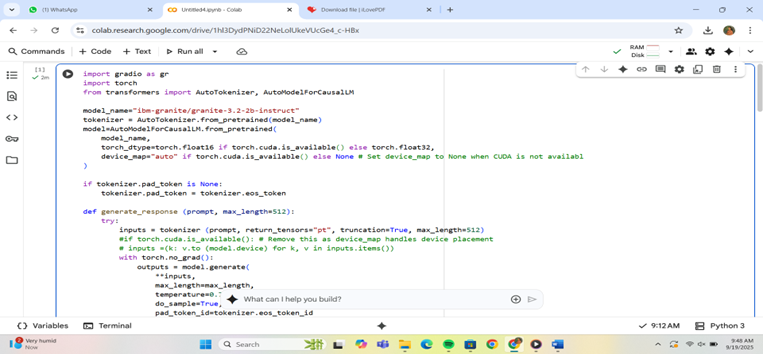
## 9. User Interface

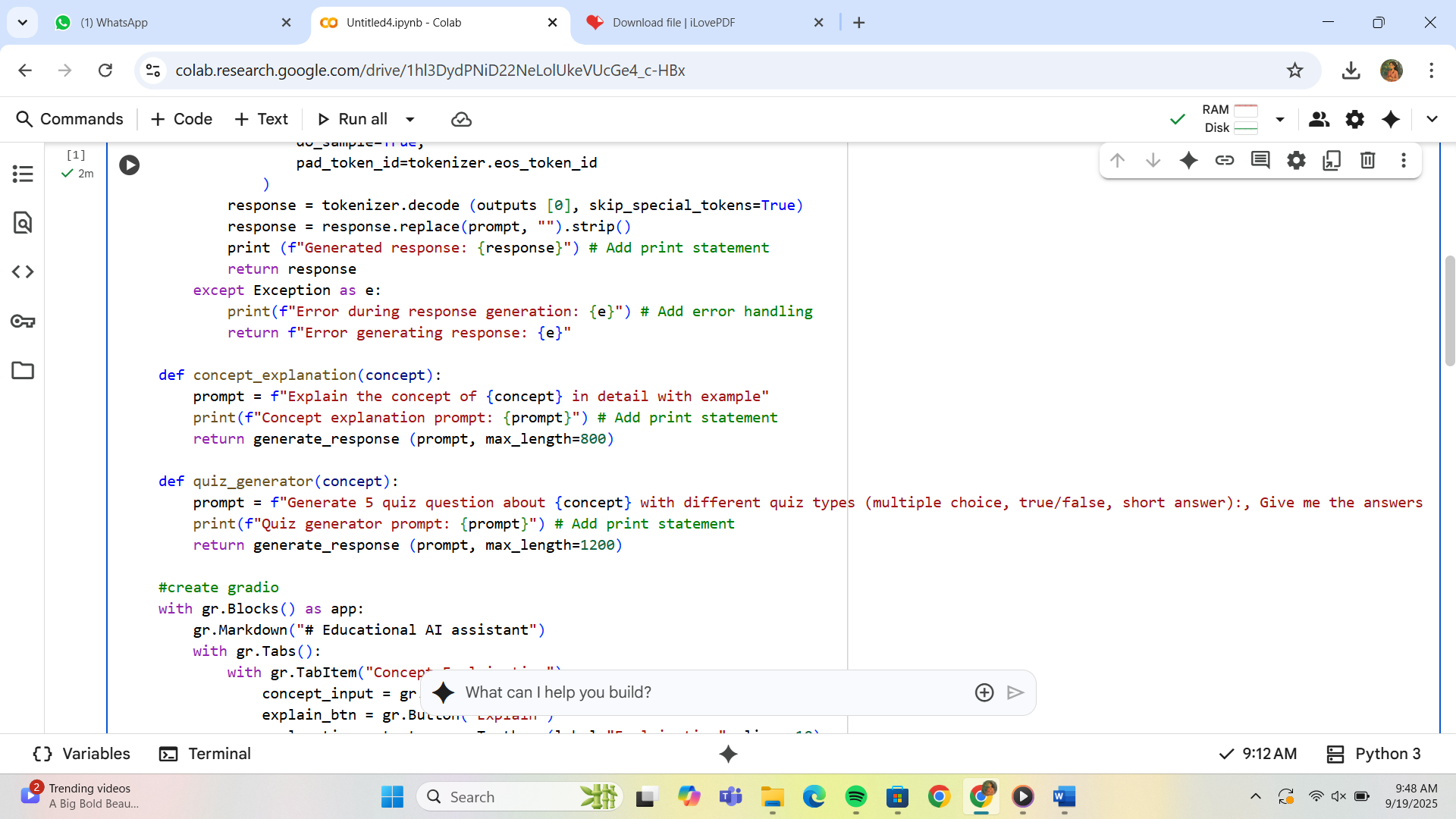
The user interface is built with Gradio and contains:  
• Tab 1: Requirement Analysis – Upload PDF or type requirements, view structured analysis.  
• Tab 2: Code Generation – Input requirements, select programming language, generate code.  
The design is minimalistic, user-friendly, and suitable for students and professionals.

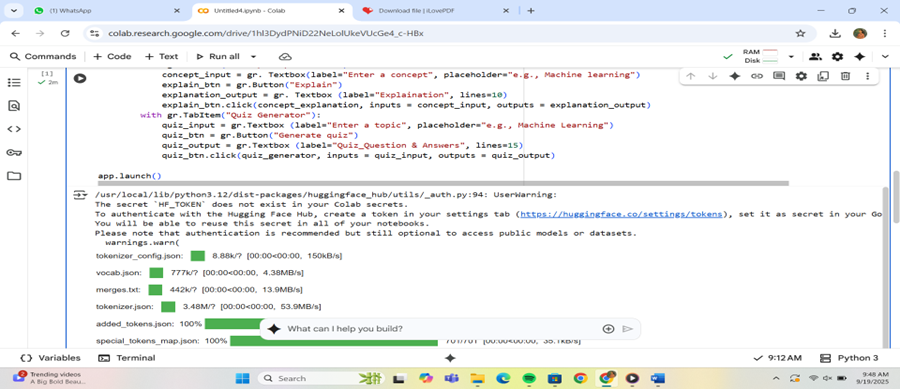
## 10. Testing

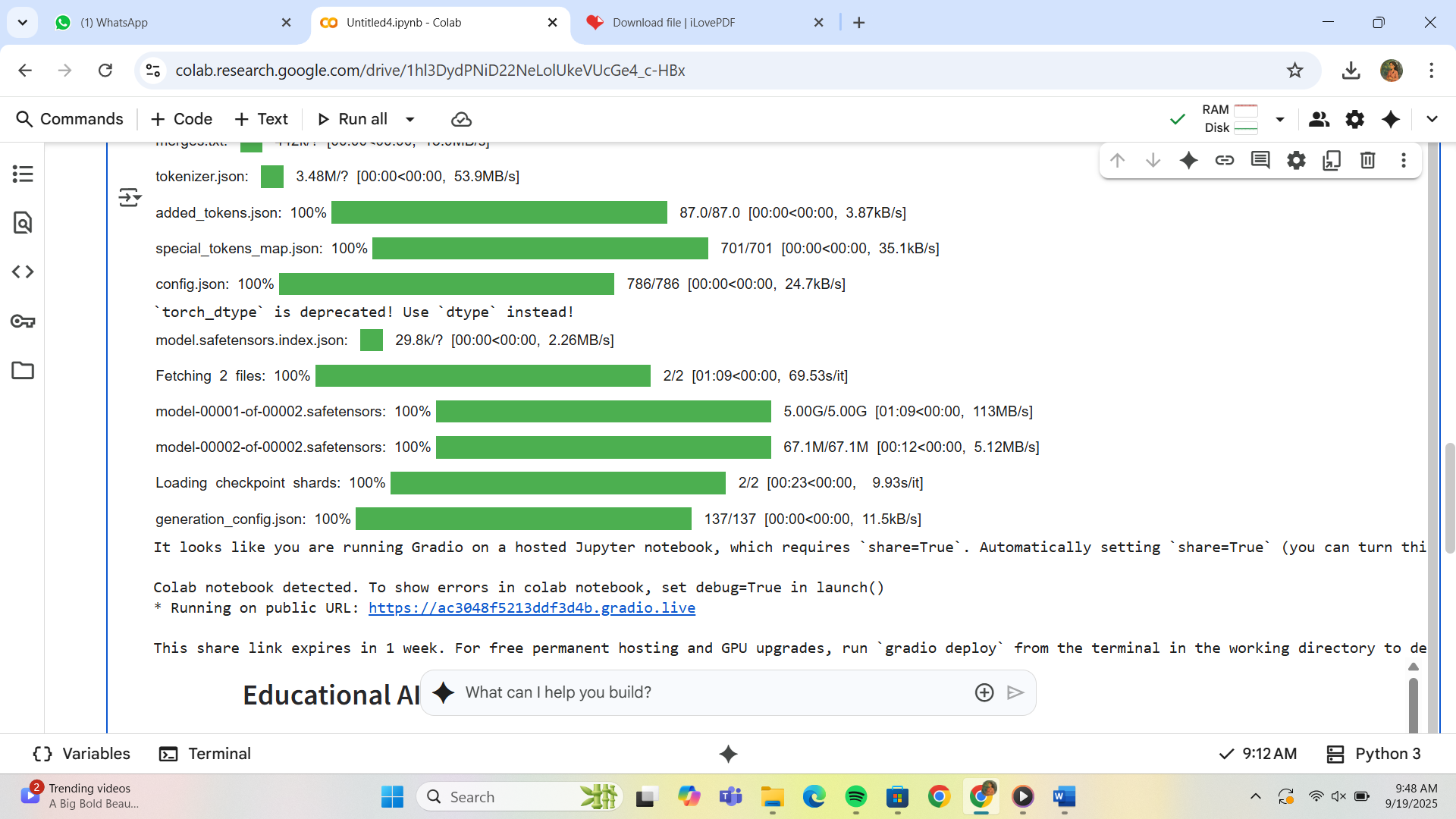
Testing strategy included:  
• Unit Testing – Verifying individual functions like generate\_response and extract\_text\_from\_pdf.  
• Functional Testing – Validating requirement analysis and code generation.  
• Manual Testing – Uploading PDFs, typing inputs, checking responses.  
• Edge Case Handling – Testing empty inputs, very large files, unsupported formats.

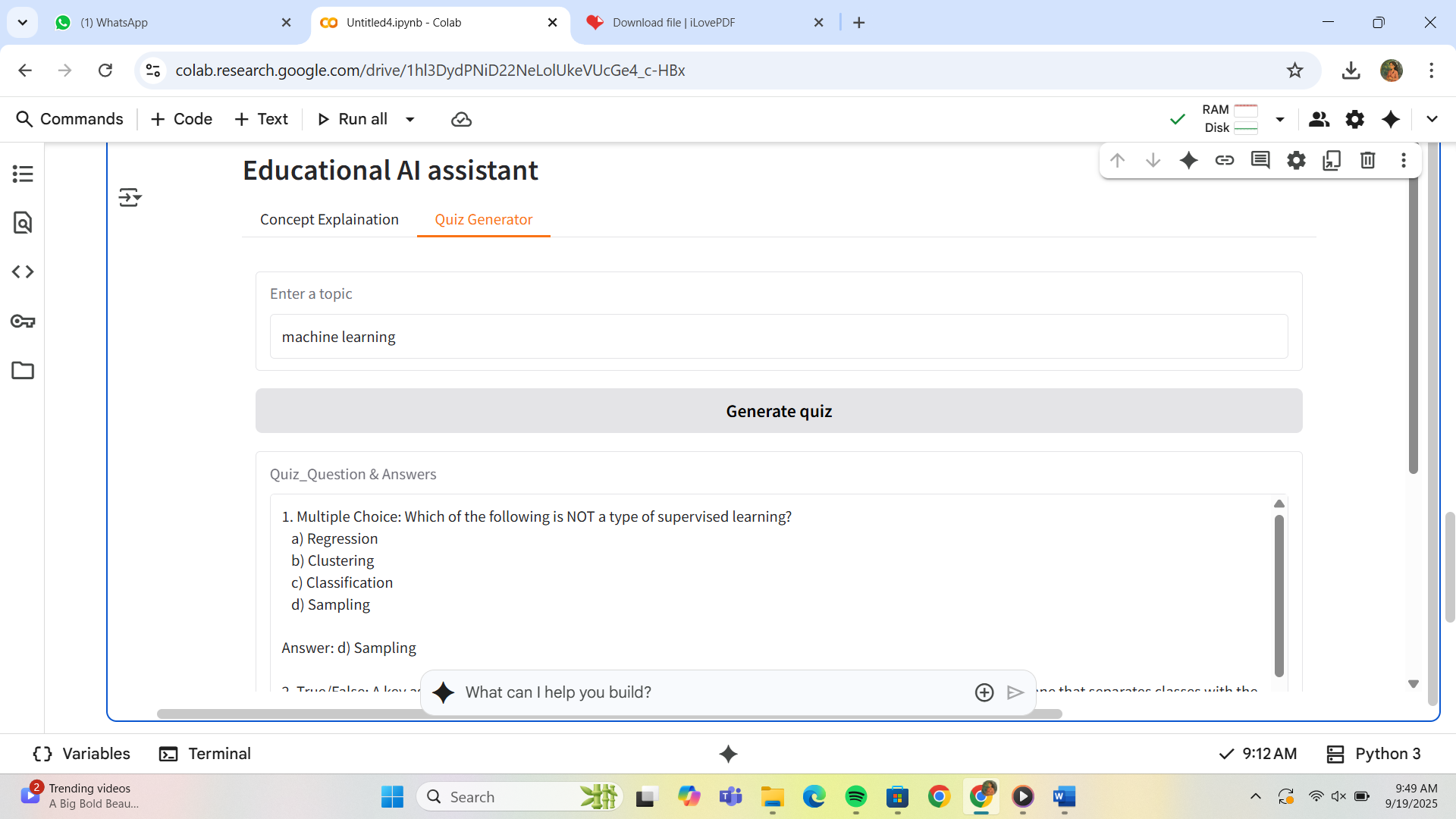
## 11. Screenshots

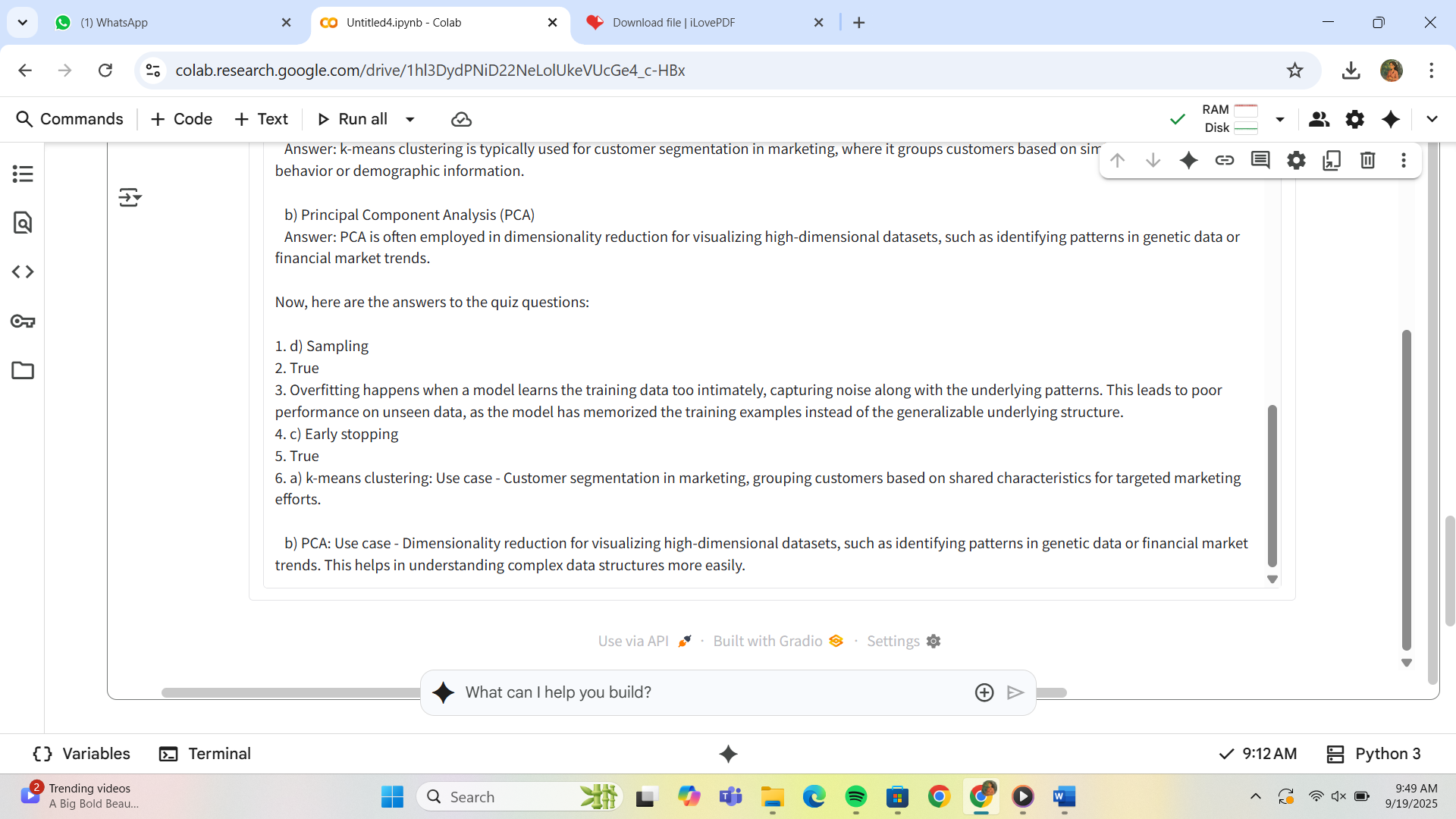












## 12. Known Issues

• Generated code may not always be production-ready.  
• Large models require significant computational power.  
• Output quality depends on LLM capabilities and prompt design.

## 13. Future Enhancement

• Add syntax highlighting and formatting for generated code.  
• Enable export of results into Word/PDF reports.  
• Provide REST APIs for integration with other platforms.  
• Add authentication and session-based history.  
• Enhance visualization of requirement analysis results.