

# SmartSDLC – AI-Enhanced Software Development Lifecycle Documentation

---

## 1.Introduction

- Project Title: SmartSDLC – AI-Enhanced Software Development Lifecycle

TEAM MEMBERS 1     -ISHWARYA S  
TEAM MEMBERS 2     -KIRUBA SHALINI S  
TEAM MEMBERS 3     -HARINI SHREE S

## 2.Project Overview

- Purpose:  
SmartSDLC is an AI-powered platform designed to automate and streamline the Software Development Lifecycle (SDLC). It leverages IBM Watsonx Granite models, LangChain, FastAPI, and Streamlit to enhance each phase of software engineering, including requirements analysis, code generation, test case creation, bug fixing, and documentation. By integrating AI-driven automation, SmartSDLC reduces manual workload, accelerates development timelines, and improves software quality.
- Features:
  - Requirement Upload & Classification  
Key Point: Structured requirement management  
Functionality: Extracts text from uploaded PDFs and classifies sentences into SDLC phases (Requirements, Design, Development, Testing, Deployment).
  - AI Code Generator  
Key Point: Automated code creation  
Functionality: Generates clean, production-ready code from natural language prompts or structured user stories.
  - Bug Fixer  
Key Point: Error detection and correction  
Functionality: Identifies and fixes syntax and logic errors in code snippets, returning optimized versions.
  - AI-Driven Test Case Generation

Key Point: Automated testing support

Functionality: Generates unit and integration test cases from generated or user-provided code.

- Code Summarization & Documentation

Key Point: Improved maintainability

Functionality: Summarizes and documents code to improve readability and project understanding.

- Chatbot Assistance

Key Point: Real-time developer support

Functionality: Provides interactive guidance and assistance for SDLC-related queries.

- GitHub Integration

Key Point: Workflow automation

Functionality: Automates pushing code, opening issues, and syncing documentation with GitHub repositories.

- Use Case Scenarios:

- Requirement Upload & Classification: Upload raw requirement PDFs and receive structured user stories grouped by SDLC phase.

- AI Code Generator: Generate working Python or JavaScript code from natural language prompts.

- Bug Fixer: Submit buggy code and receive AI-optimized corrections.

- Test Case Generation: Automatically generate test cases for faster validation.

- Chatbot Support: Access an AI-powered assistant to answer SDLC queries and guide workflows.

### 3. Architecture

- Frontend (Streamlit): Interactive dashboard for requirements, code generation, bug fixing, testing, and chatbot interaction.

- Backend (FastAPI): API layer handling routing, authentication, AI requests, and service orchestration.

- AI Integration (IBM Watsonx + LangChain): Natural language processing, code generation, bug fixing, and summarization.

- Modules: Requirement analysis, code generation, bug fixing, test case generation, code summarization, GitHub workflows.

- Deployment: Local hosting with Uvicorn (backend) and Streamlit (frontend).

### 4. Setup Instructions

Prerequisites:

- Python 3.10+

- FastAPI, Uvicorn

- Streamlit

- IBM Watsonx API access
- LangChain
- PyMuPDF (fitz)
- Git & GitHub

#### Installation Process:

- Install Python 3.10 and pip
- Create virtual environment: `python -m venv myenv`
- Activate environment and install dependencies from `requirements.txt`
- Configure `.env` file with API keys and model IDs
- Start FastAPI backend: `uvicorn app.main:app --reload`
- Run Streamlit frontend: `streamlit run frontend/Home.py`

## 5. Folder Structure

- `app/` – FastAPI backend
  - `routes/` – API endpoints for AI, chat, auth, feedback
  - `services/` – Core AI service logic
  - `models/`, `utils/` – Supporting modules
- `frontend/` – Streamlit UI components
  - `Home.py` – Entry dashboard
  - `pages/` – Modular pages (requirements, code gen, bug fixer, etc.)
- `ai_story_generator.py` – Requirement classification
- `code_generator.py` – Code and test case generation
- `bug_resolver.py` – Bug fixing
- `doc_generator.py` – Code summarization
- `conversation_handler.py` – Chatbot logic
- `github_service.py` – GitHub workflow automation

## 6. Running the Application

- Start FastAPI backend with Uvicorn
- Run Streamlit dashboard
- Navigate via dashboard menu
- Upload requirements or enter prompts
- Generate code, fix bugs, create tests, and access chatbot
- Sync outputs with GitHub and export documentation

## 7. API Documentation

- `POST /upload-pdf` – Uploads requirements for classification
- `POST /generate-code` – Generates production-ready code
- `POST /fix-bugs` – Accepts buggy code and returns corrected version
- `POST /generate-tests` – Creates test cases

- POST /summarize-code – Summarizes uploaded code
- POST /chat – Chatbot interactions
- POST /feedback – Submits user feedback
- GET /docs – Swagger UI for API exploration

## 8. Authentication

- Token-based authentication (JWT)
- Role-based access (admin, developer, tester)
- Hashed user login and registration
- Planned: OAuth2 integration and session management

## 9. User Interface

- Home Dashboard: Feature overview and navigation
- Requirement Classifier: Upload and classify requirements
- Code Generator: Prompt-based code creation
- Bug Fixer: Code correction interface
- Test Generator: Auto-generated test cases
- Chatbot: Real-time AI guidance
- Feedback Form: Collects user feedback
- GitHub Sync: Push code, open issues, sync docs

## 10. Testing

- Unit Testing: For backend AI services
- API Testing: Swagger UI and Postman
- Manual Testing: For requirement classification, bug fixing, and chatbot
- Edge Cases: Large PDF uploads, malformed prompts, incorrect API keys

## 11. Known Issues

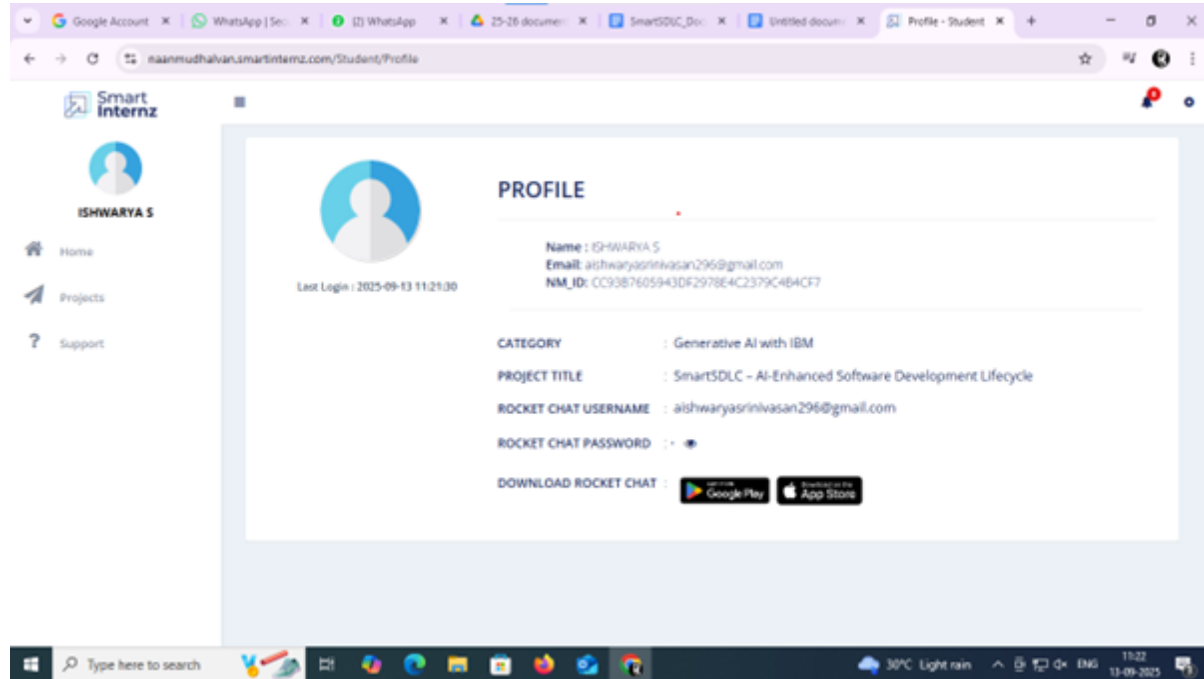
- Limited offline support
- Dependency on IBM Watsonx API availability
- Occasional latency for large PDFs
- Basic test case generation (needs extension)

## 12. Future Enhancements

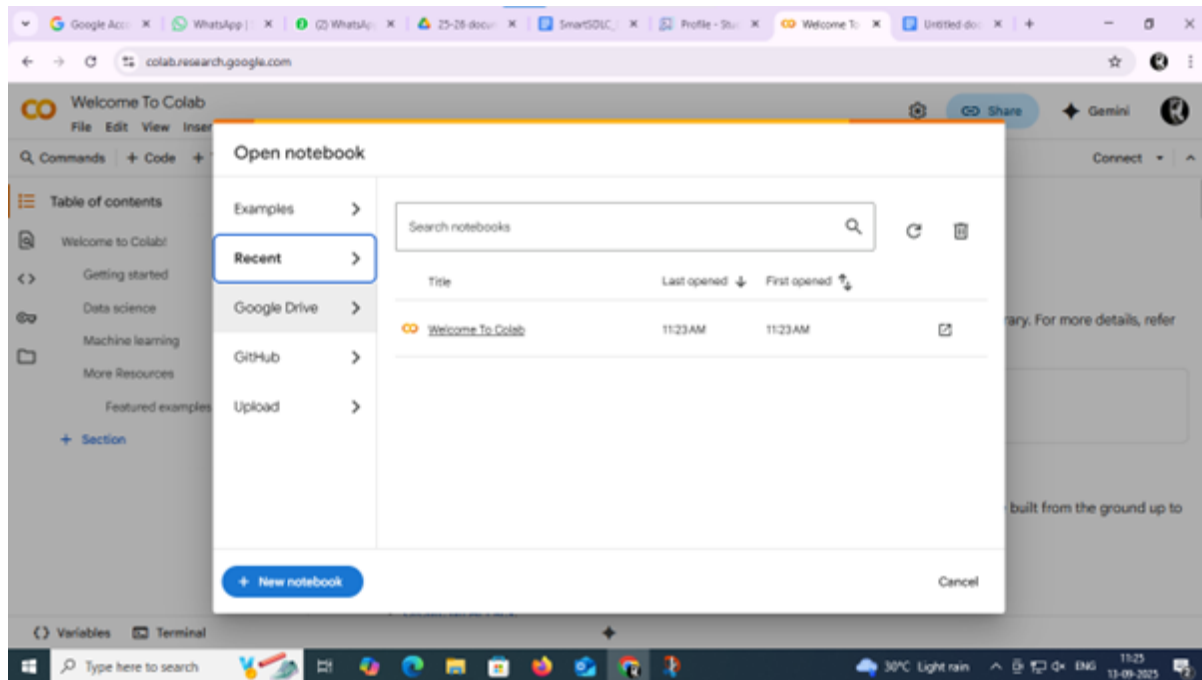
- CI/CD pipeline integration
- Multi-language support for code generation
- Advanced bug detection with deep learning
- Cloud deployment (AWS, IBM Cloud, Azure)

- Collaboration features for team workflows
- Enhanced test generation with coverage analysis

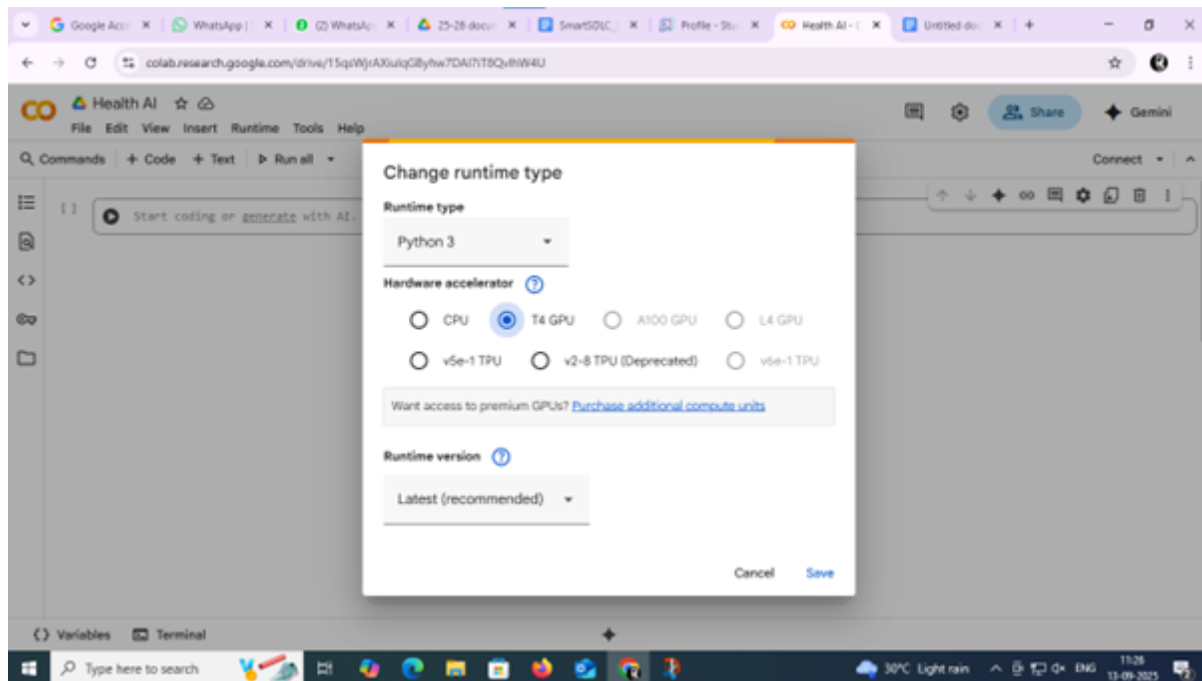
## 13.Screenshots



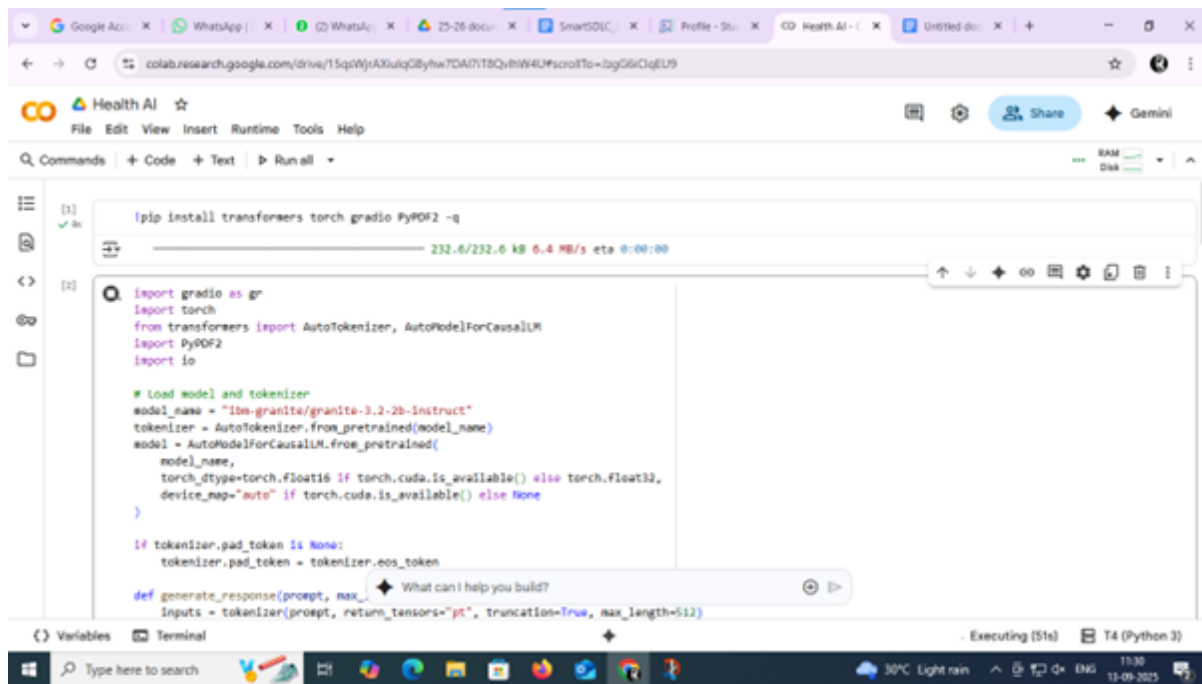
Click on access resources, then click on project workspace to place the demo link.



Open Google Colab, then click on new notebook.



Choose runtime and click change runtime type to “T4 GPU”.



The screenshot shows a Google Colab notebook interface. The top bar includes the Google Assistant icon, a star for bookmarks, and a 'Share' button. The menu bar contains 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu, there are tabs for 'Commands', '+ Code', '+ Text', and 'Run all'. The main code editor displays the following Python code:

```
[1] |pip install transformers torch gradio PyPDF2 -q

[2] 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_
Inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=512)
```

The bottom status bar indicates 'Executing (S1s)' and 'T4 (Python 3)'.

Then run the above code in the cell.

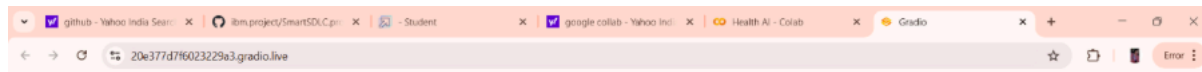


The screenshot shows the same Google Colab notebook after the code has been executed. The code editor now displays the output of the execution, which includes progress bars for various components:

- tokenizer\_config.json: 8.88k/? [00:00<00:00, 792kB/s]
- vocab.json: 777k/? [00:00<00:00, 10.3MB/s]
- merges.txt: 442k/? [00:00<00:00, 25.6MB/s]
- tokenizer.json: 3.48M/? [00:00<00:00, 99.2MB/s]
- added\_tokens.json: 100% [00:00<00:00, 67.0/67.0 [00:00<00:00, 6.79kB/s]
- special\_tokens\_map.json: 100% [00:00<00:00, 791/791 [00:00<00:00, 75.9kB/s]
- config.json: 100% [00:00<00:00, 786/786 [00:00<00:00, 78.9kB/s]
- "torch\_dtype" is deprecated! Use "dtype" instead!
- model.safetensors.index.json: 29.8k/? [00:00<00:00, 2.68MB/s]
- Fetching 2 files: 100% [00:00<00:00, 2/2 [01:30<00:00, 98.87kB/s]
- model-00001-of-00002.safetensors: 100% [00:00<00:00, 5.00G/5.00G [01:30<00:00, 40.0MB/s]
- model-00002-of-00002.safetensors: 100% [00:01<00:00, 67.1MB/67.1M [00:01<00:00, 67.6MB/s]
- Loading checkpoint shards: 100% [00:00<00:00, 2/2 [00:00<00:00, 8.83MB/s]
- generation\_config.json: 100% [00:00<00:00, 1.0/1.0 [00:00<00:00, 1.0MB/s]

The bottom status bar now shows '✓ 11:32 AM' and 'T4 (Python 3)'.

Thus the above code is executed and the output will be generated.



## AI Code Analysis & Generator

Code Analysis Code Generation

Upload PDF

Drop File Here

- OR -

Click to Upload

Or write requirements here

what is meant by smart sdic and how does it differ from the traditional sdic?

Analyze

Requirements Analysis

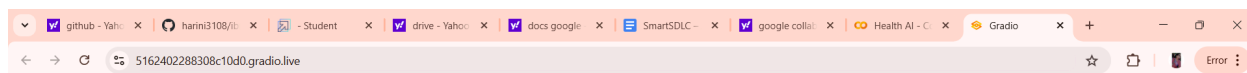
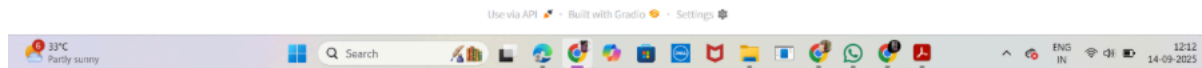
Supporting environment on code integration, testing, and deployment systems:

- Collaboration Tools: Integrate collaboration tools, like version control systems, issue trackers, and communication platforms, to facilitate seamless teamwork throughout the SDLC.
- Reporting and Visualization: Smart SDLC must provide customizable and interactive reporting, as well as visual analytics to help stakeholders monitor progress and assess project health.

4. Differences:

- Predictive Analysis: Smart SDLC employs AI for predictive analysis, enabling organizations to anticipate risks, prioritize tasks, and optimize resource allocation. Traditional SDLC lacks this capability.
- Continuous Improvement: Smart SDLC incorporates machine learning for continuous feedback loops and iterative improvements along the SDLC, while Traditional SDLC is static and relies on periodic reviews.
- Decision Support: Smart SDLC offers AI-driven decision support, suggesting optimal practices, and automating routine decision-making within SDLC stages, unlike Traditional SDLC's human-centric decision-making process.
- Automation: Smart SDLC automates more SDLC processes, including code reviews, testing, and deployment, with the help of AI and ML technologies, whereas Traditional SDLC relies more on manual intervention.

In conclusion, Smart SDLC represents a more advanced and intelligent approach to software development, incorporating AI and machine learning for enhanced predictive analysis, continuous improvement, decision support, and process automation. This contrasts with the traditional SDLC, which is manual, predictive, and lacks the advanced capabilities of AI integration.



## AI Code Analysis & Generator

Code Analysis Code Generation

Code Requirements

```
import random

print("Welcome to the Number Guessing Game!")
number = random.randint(1, 100)

while True:
    guess = int(input("Guess a number between 1 and 100: "))
    if guess < number:
        print("Too low!")
    elif guess > number:
        print("Too high!")
    else:
        print("Congratulations! You guessed it!")
        break
```

Programming Language

Python

Generate Code

Generated Code

```
python
import random

print("Welcome to the Number Guessing Game!")
number = random.randint(1, 100)

while True:
    guess = int(input("Guess a number between 1 and 100: "))
    if guess < number:
        print("Too low!")
    elif guess > number:
        print("Too high!")
    else:
        print("Congratulations! You guessed it!")
        break
```

