

SmartSDLC -AI Enhanced Software Development Lifecycle Generative



S

Shalini.K

Team Members:

Sandhiya.S

PriyaDharshini.G

Rajeshwari.D

1.SYSTEM ANALYSIS

1.1.Introduction

The SmartSDLC – AI-Enhanced Software Development Lifecycle Generative is an advanced framework that integrates artificial intelligence technologies into the traditional software development lifecycle (SDLC). It aims to optimize planning, design, implementation, testing, and maintenance phases by using AI algorithms to automate repetitive tasks, predict risks, and enhance decision-making. This approach helps software teams reduce development time, improve product quality, and efficiently allocate resources while maintaining scalability and adaptability to various project types.

1.2.Objectives

- *The main objectives of the SmartSDLC system are:
- * To leverage AI-powered tools to enhance each phase of the software development lifecycle.
- * To automate tasks such as requirement analysis, test case generation, and code review.
- * To improve accuracy, predict potential defects, and ensure better software quality.
- * To enable data-driven decision-making and provide real-time insights to stakeholders.
- * To reduce development cost and time by optimizing workflows.
- * To support collaboration and knowledge sharing across teams using intelligent tools.

1.3.Existing System

In most traditional SDLC approaches, software development is a manual, time-consuming, and error-prone process. Common limitations include:

- * Lack of real-time feedback on code quality or potential bugs.
- * Requirement gathering and validation often involve human errors.
- * Testing is resource-intensive and may miss critical issues.
- * Maintenance and updates rely heavily on human intervention without predictive insights.
- * Difficulty in tracking project progress and identifying bottlenecks.

* Existing SDLC models like Waterfall, Agile, and DevOps provide structure and incremental delivery but lack AI-driven automation and predictive capabilities, leading to inefficiencies in complex or large-scale projects.

1.4. Proposed System

The SmartSDLC introduces an AI-driven, adaptive software development lifecycle that:

- * Uses machine learning models to predict risks and defects during early stages.
- * Automates requirement validation using natural language processing (NLP).
- * Generates optimized test cases based on code changes.
- * Integrates AI-driven analytics to monitor project progress and suggest improvements.
- * Offers intelligent code recommendations to developers.
- * Enhances collaboration by providing smart dashboards and workflow suggestions.
- * Supports continuous integration and deployment with automated monitoring.

This proposed system combines AI technologies with best practices in SDLC, making software development faster, more accurate, and cost-efficient while reducing human error and workload.

1.5. Tools and Technologies Used

The SmartSDLC system leverages a wide range of AI-powered tools and modern technologies:

Artificial Intelligence Technologies:

- * **Machine Learning (ML):** For defect prediction, anomaly detection, and optimization.
- * **Natural Language Processing (NLP):** For requirement analysis, documentation, and chatbot-based support.
- * **Computer Vision (Optional):** For UI testing automation.

Development Tools:

- * Python / R: For AI algorithms and data processing.
- * TensorFlow / PyTorch: For building and training ML models.
- * Scikit-learn: For lightweight ML models and analytics.

DevOps Tools:

- * Jenkins / GitHub Actions: For continuous integration and deployment.
- * Docker / Kubernetes: For containerization and orchestration.

Collaboration Tools:

- * JIRA / Trello: For project management.
- * Slack / Microsoft Teams: For communication and collaboration.

Database & Storage:

- * MySQL / PostgreSQL / MongoDB: For storing project data, logs, and analytics.
- * Cloud Platforms (AWS, Azure, Google Cloud): For scalable computing and storage.

Testing Tools:

- * Selenium / JUnit / PyTest: For automated testing and test case generation.
- * SonarQube: For static code analysis and code quality checks.

UI/UX Tools:

- * Figma / Adobe XD: For prototype design and user feedback analysis.

2.SYSTEM DESIGN

2.1. Project Description

The SmartSDLC project focuses on integrating artificial intelligence into every phase of the Software Development Lifecycle (SDLC) to automate and optimize processes. The system is designed to assist software development teams in requirement gathering, design, coding, testing, deployment, and maintenance by providing intelligent recommendations, predictive insights, and automation tools.

The system includes the following key components:

- * **Requirement Analysis Module:** Uses NLP to analyze, validate, and structure user requirements.
- * **Design Assistance Module:** Provides architecture templates and design suggestions based on historical data and best practices.
- * **Code Generation & Review:** AI-driven tools assist developers by suggesting code snippets and identifying possible errors or inefficiencies.
- * **Test Case Generator:** Automates test case creation based on code changes and requirements.
- * **Defect Prediction Engine:** Uses ML models to predict areas prone to bugs.
- * **Monitoring & Reporting Dashboard:** Real-time project metrics, risk alerts, and progress visualization.

This system is suitable for organizations aiming to streamline their software development while reducing cost, time, and human error.

2.2. Testing

Testing in SmartSDLC is both automated and adaptive, ensuring that the software meets functional and non-functional requirements.

Testing Types:

Unit Testing:

- * Frameworks like JUnit, PyTest, and Nose are used to test individual components.
- * Automated test cases are generated based on code logic.

Integration Testing:

- * Ensures different modules interact seamlessly.
- * Uses Selenium and Postman to test APIs and data exchanges.

Regression Testing:

- * Automated re-testing of existing functionalities after updates or bug fixes.
- * AI models prioritize tests based on code changes.

Performance Testing:

- * Tools like JMeter assess scalability, response times, and load handling.

Security Testing:

- * Uses static code analysis (SonarQube) to detect vulnerabilities like SQL injection, XSS, etc.

User Acceptance Testing (UAT):

- * Collects feedback through interactive dashboards and NLP-powered chatbots.

AI-Driven Testing Features:

- * Automated test case generation.
- * Predictive defect analysis.
- * Intelligent prioritization of critical tests.
- * Continuous monitoring of test results with alerts.

2.3.Sample Output

Test Case Generation

Requirement: User registration must validate email format.

Generated Test Cases:

1. Input valid email – Expect success.
2. Input invalid email – Expect error message “Invalid email format.”
3. Input empty email field – Expect error message “Email is required.”

2.4. Future Enhancements

The SmartSDLC platform is designed for scalability and can be further enhanced with the following improvements:

Self-Learning AI Models:

- * Incorporate feedback loops where AI improves over time by learning from past defects and corrections.

Integration with IoT and Edge Computing:

- * Support development for connected devices and real-time data processing.

Voice-Enabled Requirement Analysis:

- * Allow users to input requirements via voice commands for faster documentation.

AI-Powered Project Management:

- * Automatically allocate tasks, set deadlines, and balance workloads based on team availability and expertise.

Enhanced Security Analysis:

- * Incorporate deep learning models for detecting sophisticated security threats.

Support for Multi-Language Development:

- * Extend AI tools to support code review and suggestions in multiple programming languages.

User Behavior Analytics:

- * Track how users interact with applications and recommend UI/UX improvements.

Compliance Monitoring:

- * Automate checks for industry standards and regulatory compliance.

3.CODING

```
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 extract_text_from_pdf(pdf_file):
```

```
    if pdf_file is None:
```

```
        return ""
```

```
    try:
```

```
        pdf_reader = PyPDF2.PdfReader(pdf_file)
```

```
        text = ""
```

```
        for page in pdf_reader.pages:
```

```
            text += page.extract_text() + "\n"
```

```
        return text
```

```
    except Exception as e:
```

```
        return f"Error reading PDF: {str(e)}"
```

```
def requirement_analysis(pdf_file, prompt_text):
```

```
    # Get text from PDF or prompt
```

```
    if pdf_file is not None:
```

```
        content = extract_text_from_pdf(pdf_file)
```

```
        analysis_prompt = (
```

```
            f"Analyze the following document and extract key software requirements. "
```

```
            f"Organize them into functional requirements, non-functional requirements, and "
```

```
            f"technical specifications:\n\n{content}"
```

```
        )
```

```
    else:
```

```
        analysis_prompt = (
```

```
            f"Analyze the following requirements and organize them into functional requirements, "
```

```
            f"non-functional requirements, and technical specifications:\n\n{prompt_text}"
```

```
        )
```

```
    return generate_response(analysis_prompt, max_length=1200)
```

```

def code_generation(prompt, language):
    code_prompt = f"Generate {language} code for the following requirement:\n\n{prompt}\n\nCode:"
    return generate_response(code_prompt, max_length=1200)

# Create Gradio interface
with gr.Blocks() as app:
    gr.Markdown("# AI Code Analysis & Generator")

    with gr.Tabs():
        with gr.TabItem("Code Analysis"):
            with gr.Row():
                with gr.Column():
                    pdf_upload = gr.File(label="Upload PDF", file_types=[".pdf"])
                    prompt_input = gr.Textbox(
                        label="Or write requirements here",
                        placeholder="Describe your software requirements...",
                        lines=5
                    )
                    analyze_btn = gr.Button("Analyze")

                with gr.Column():
                    analysis_output = gr.Textbox(label="Requirements Analysis", lines=20)

            analyze_btn.click(
                requirement_analysis,
                inputs=[pdf_upload, prompt_input],
                outputs=analysis_output
            )

        with gr.TabItem("Code Generation"):
            with gr.Row():
                with gr.Column():
                    code_prompt = gr.Textbox(
                        label="Code Requirement",
                        placeholder="Describe what code you want to generate...",

```

```

        lines=5
    )
    language_dropdown = gr.Dropdown(
        choices=["Python", "JavaScript", "Java", "C++", "C#", "PHP", "Go", "Rust"],
        label="Programming Language",
        value="Python"
    )
    generate_btn = gr.Button("Generate Code")

    with gr.Column():
        code_output = gr.Textbox(label="Generated Code", lines=20)

    generate_btn.click(
        code_generation,
        inputs=[code_prompt, language_dropdown],
        outputs=code_output
    )

app.launch(share=True)

```

3.1.Code with Explanation

Explanation of the Code

1. Model Setup

- * Uses Hugging Face Transformers with the "ibm-granite/granite-3.2-2b-instruct" model.
- * Uses GPU if available (torch.cuda.is_available()), otherwise CPU.
- * Sets padding to the end-of-sequence token if undefined.

2. Main Functions

- * generate_response(prompt, max_length=1024): Generates text from the model given a prompt.
- * extract_text_from_pdf(pdf_file): Reads and extracts text from an uploaded PDF file using PyPDF2.
- * requirement_analysis(pdf_file, prompt_text): Analyzes the uploaded PDF or manual prompt and categorizes requirements.
- * code_generation(prompt, language): Generates code for a specified programming language based on the provided requirement.

3.Gradio Interface Structure

Code Analysis Tab

- * Upload a PDF or write requirements manually.

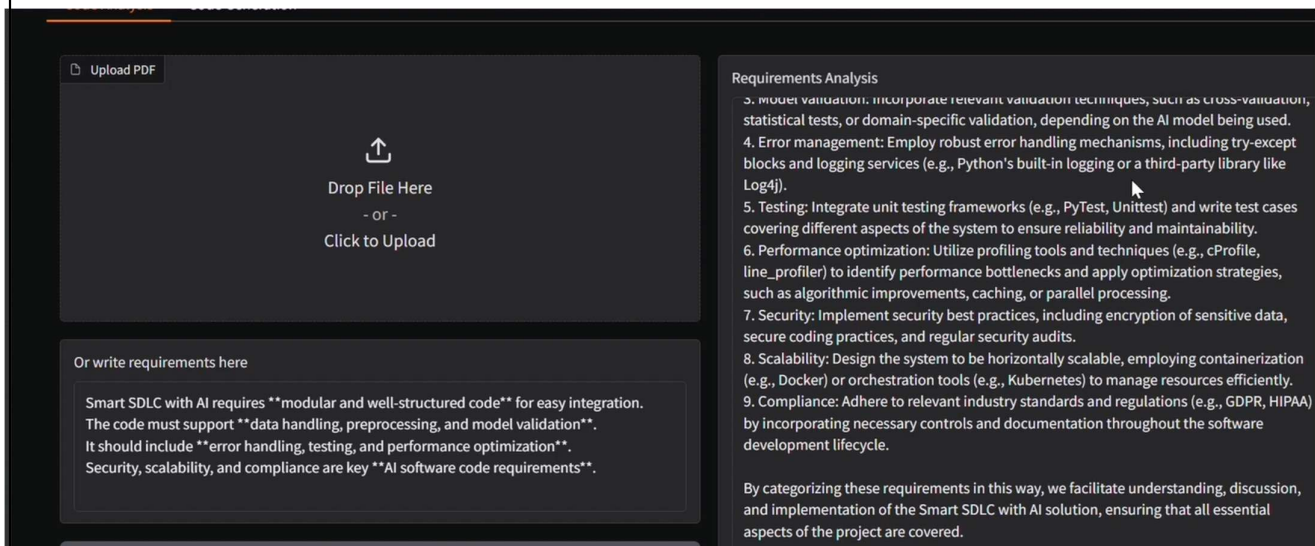
- * Analyze button triggers requirement extraction and categorization.
- * Displays structured requirements in a textbox.

Code Generation Tab

- * Enter the requirement and choose a programming language.
- * Generate button creates corresponding code snippet.
- * Displays the generated code.

3.2.Input and Output

Input : Enter “Create a function to sum even numbers” and select Python.



Output : Python code defining a function that sums even numbers in a list.

3.3.Screenshot

Fig 1:Requirement Analysis

Fig 2.Code generation

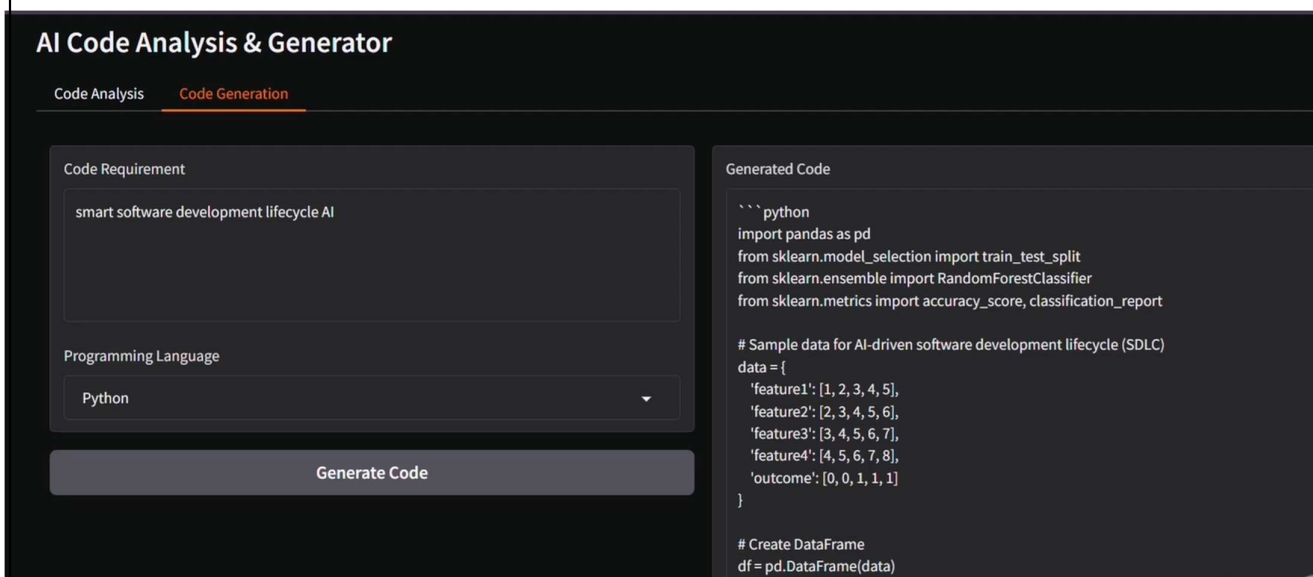
3.4.Advantages

- * Works fast and saves time.
- * Makes fewer mistakes.
- * Gives helpful advice.
- * Supports many languages and files.

3.5. Limitations

- * Needs clear information to work well.
- * Needs a good computer to run.
- * Not perfect for every project.
- * Can have safety issues with private data.

3.6.Application



The screenshot displays the 'AI Code Analysis & Generator' application interface. It features two tabs: 'Code Analysis' and 'Code Generation', with the latter being the active tab. The 'Code Requirement' section contains a text input field with the text 'smart software development lifecycle AI'. Below this, the 'Programming Language' is set to 'Python' via a dropdown menu. A 'Generate Code' button is positioned at the bottom of the input section. The 'Generated Code' section on the right shows the resulting Python code, which includes imports for pandas, sklearn, and train_test_split, followed by sample data for an AI-driven SDLC and a DataFrame creation command.

```
`` python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report

# Sample data for AI-driven software development lifecycle (SDLC)
data = {
    'feature1': [1, 2, 3, 4, 5],
    'feature2': [2, 3, 4, 5, 6],
    'feature3': [3, 4, 5, 6, 7],
    'feature4': [4, 5, 6, 7, 8],
    'outcome': [0, 0, 1, 1, 1]
}

# Create DataFrame
df = pd.DataFrame(data)
```

- * Making apps and websites faster.

- * Helping students learn coding.
- * Planning work in companies easily.
- * Studying and finding information quickly.
- * Saving time and money for new businesses.

4.CONCLUSION

The SmartSDLC makes software work easier and faster by using AI. It helps teams write code, plan projects, and learn new things with less effort. Even though it needs clear input and a good computer, it saves time and helps avoid mistakes. It's useful for students, businesses, and anyone who wants to create software more easily.

4.1.Reference

You can learn more about the tools and technologies used in this project from the following links:

- * **Gradio:** <https://gradio.app/>
- * **Transformers by Hugging Face:** <https://huggingface.co/transformers/>
- * **PyTorch:** <https://pytorch.org/>
- * **IBM Granite Model:** <https://huggingface.co/ibm-granite/granite-3.2-2b-instruct>