# AI-Powered Software Development: Transforming the SDLC through Intelligent Automation

**Ms. Riya Yadav**

Department of Information technology, Gujarat technological University.

**Abstract**:

A paradigm shift in software engineering is emerging as artificial intelligence (AI) tools and methods become increasingly embedded within the Software Development Life Cycle (SDLC). This paper introduces AISA-SDLC (AI-Integrated Software Automation - Software Development Lifecycle), a modular framework that integrates intelligent automation, human oversight, governance, and continuous feedback into each phase of modern software development. To ground the framework in real-world practice, a developer-centric survey was conducted to analyse current AI adoption trends, usage patterns, and trust levels within the developer community. A layer-by-layer prototype was created and evaluated based on these observations using easily accessible open-source techniques, such as large language model (LLM) code generation, static linting, vector similarity search using FAISS and Sentence-BERT embeddings, lightweight runtime observability using Prometheus and Grafana, and natural language prompt structuring. Even while AI greatly speeds up typical coding jobs, explainability, governance checks, and layered human oversight are still necessary to guarantee trust, security, and maintainability, according to a comparison with a traditional workflow. The study concludes by outlining current limitations and future directions for scaling AISA-SDLC into a fully integrated, responsible AI-augmented development pipeline.

**Keywords**

Artificial Intelligence (AI), Software Development Life Cycle (SDLC), AISA-SDLC Model, Machine Learning, Natural Language Processing, Code Generation, Software Engineering, AI-Driven Development, Developer Productivity, Resource Optimisation.

## Introduction

Reliable software systems are designed, developed, tested, deployed, and maintained using a well-defined Software Development Life Cycle (SDLC). The requirement analysis, system design, implementation, verification, deployment, and maintenance phases of traditional SDLC frameworks are usually organised and rely mostly on developer knowledge and manual procedures. However, the constraints of traditional SDLC methodologies in terms of scalability, efficiency, and speed have become more apparent as software complexity rises and the need for quick, flexible delivery increases.

In software engineering, artificial intelligence (AI) and intelligent automation have become potent change agents, providing fresh approaches to augment human knowledge, cut down on tedious work, and improve process precision (Durrani et al., 2024) . AI is

now able to assist with activities ranging from code generation to intelligent testing because of recent developments in large language models (LLMs), machine learning, and natural language processing. AI is becoming more and more integrated into tools that directly support developers throughout the design, development, testing, and maintenance stages; it is no longer limited to end-user apps (Bhanuprakash Madupati, 2025). Research indicates that AI-driven solutions, such as automated code generators and intelligent testing frameworks, can significantly reduce manual labour, improve software quality, and speed up delivery cycles (Bhanuprakash Madupati, 2025).

However, even if AI speeds up repetitive work, it also presents difficulties, especially in the areas of explainability, transparency, and trust. Explainable AI (XAI) approaches that shed light on automated decision-making are crucial because many AI systems function as "black boxes" (Arrieta et al., 2020). This ensures regulatory compliance, accountability, and stakeholder confidence at every stage of the SDLC (Arora et al., 2025). The application of AI in software engineering is still dispersed, despite the introduction of specialised tools such as GitHub Copilot for AI-assisted development or problem identification. As of yet, no framework has been developed that methodically incorporates human oversight, AI capabilities, and governance into every phase of the lifecycle.

This study suggests AISA-SDLC (AI-Integrated Software Automation - Software Development Lifecycle), a revolutionary architectural framework intended to bridge this gap by integrating explainable AI, intelligent automation, human-in-the-loop oversight, and continuous governance into a single, cohesive pipeline. This paper explores the motivations for AI adoption, reports findings from a developer-centric survey, evaluates the current state of AI integration within the SDLC, presents the AISA-SDLC model and prototype implementation, compares it to traditional development methods, and discusses ethical and technical considerations for deploying AI-augmented software engineering workflows responsibly.

## Literature Review

Both academia and industry have paid close attention to the integration of artificial intelligence (AI) into software engineering in recent years. Researchers and practitioners have explored how machine learning (ML), natural language processing (NLP), and intelligent automation can transform individual phases of the Software Development Life Cycle (SDLC). However, a comprehensive AI-integrated framework that systematically spans the entire SDLC remains largely underdeveloped.

(Bhanuprakash Madupati, 2025) provided a foundational analysis of how AI influences conventional software development workflows, arguing that AI-driven processes introduce new architectural considerations and reshape team collaboration. Expanding this perspective, the AI-Native Software Development Lifecycle (AI-Native SDLC) model

proposed in (Hymel, 2024) sets out a theoretical and practical structure that places AI as a core component across all phases of the lifecycle, with emphasis on data-centric workflows and continuous learning. Furthermore, the role of Explainable AI (XAI) in the SDLC has gained traction. A phase-specific survey in (Arora et al., 2025) categorises XAI techniques by their relevance and utility across requirement analysis, design, implementation, testing, deployment, and maintenance, offering a practical taxonomy to enhance transparency and stakeholder trust.

Additionally, recent studies have examined how AI-powered tools affect developer productivity and code quality. For example, GitHub Copilot, powered by Codex, has shown measurable benefits in reducing cognitive load and speeding up routine coding tasks (Peng et al., 2023). Broader implications for developer creativity and workflow transformation are explored in (Ciniselli et al., 2024) and (Sauvola et al., 2024), which highlight how generative AI can handle boilerplate tasks and suggest real-time improvements. From a software engineering perspective, survey work in (Martínez-Fernández et al., 2021) has systematically reviewed engineering principles for AI-based systems, identifying open challenges in maintainability, versioning, and interpretability. Moreover, the challenge of managing technical debt is being revisited in (Babu, Binta, Samia A and Kaushal, 2023), where AI-driven methods are proposed for identifying, prioritising, and automatically refactoring legacy code.

AI's role in software testing is another area of rapid progress. As reviewed in (Khaliq, Farooq and Khan, 2022), AI techniques have enabled predictive test generation, anomaly detection, and regression suite optimization, addressing the inefficiencies of traditional manual testing. Despite these advances, however, many solutions remain isolated and narrow in scope. The work in (Naimil Navnit Gadani, 2024) underscores the growing need for holistic AI-augmented SDLC frameworks that can tackle ethical considerations, scalability, human-AI collaboration, and governance comprehensively.

In light of these developments, this study contributes by proposing the AISA-SDLC framework modular, AI-supported, and interactive model that systematically integrates intelligent automation, explainability, and governance throughout the SDLC. Unlike existing approaches that focus on specific tools or isolated phases, AISA-SDLC aims to bridge this fragmentation by offering a standardised, transparent, and collaborative approach to responsible AI-powered software engineering.

**Developer-Centric Study:**

In order to ensure that the AISA-SDLC (AI-Integrated Software Automation - Software Development Life Cycle) framework addresses practical problems, a developer-centric study was conducted to understand current adoption patterns, use cases, concerns, and confidence levels surrounding AI tools in software development. To ensure a representative sample of all expertise levels, the survey was distributed using Google Forms and received responses from over 50 participants. Early-career developers,

seasoned software professionals, and computer science students were among the responders.

### A. Survey Insights with Visual Representation

To provide clarity and empirical support, responses to four key questions were visualized through charts (Figures 4–7), highlighting pivotal trends in AI adoption and perception across the development community.

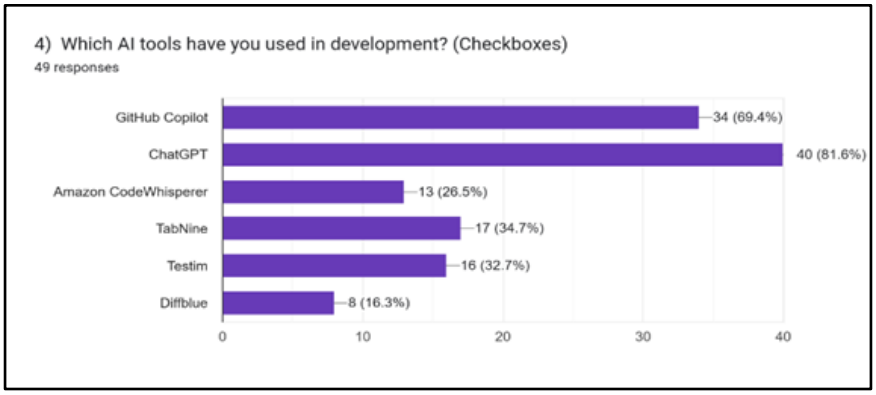



*Figure 1: Adoption Rates of AI Coding Tools*

The survey shows high adoption of AI coding tools, with ChatGPT (81.6%) and GitHub Copilot (69.4%) being the most widely used. Other tools like TabNine, Testim, CodeWhisperer, and Diffblue had lower but notable usage, indicating a growing yet concentrated tool landscape.

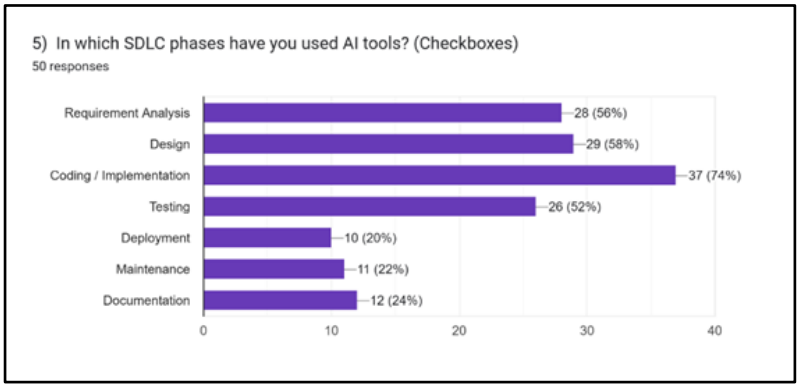



*Figure 2: Primary Application Areas of AI in SDLC*

AI tools are mainly used for code generation and debugging (74%), followed by design (58%) and requirement analysis (56%). This demonstrates AI's growing role beyond coding, supporting early SDLC phases like planning and design.

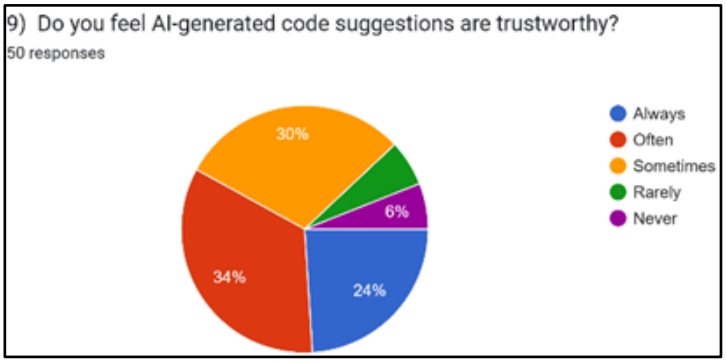



*Figure 3: Developer Trust in AI-Generated Code*

Trust in AI outputs is moderate. While 34% found them "often" trustworthy and 30% selected "always" or "sometimes," a small 3% viewed them as "rarely" reliable, highlighting the need for human oversight.

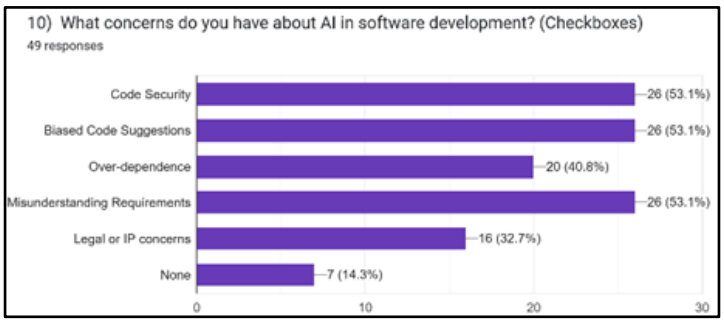



*Figure 4: Top Developer Concerns Regarding AI Use*

Both technical and ethical concerns are reflected in the following major issues: over-reliance on AI (40.8%), requirement misinterpretation (53.1%), code security risks (53.1%), and problems including bias and legal ambiguity (53.1%).

B. Impacts for AI-Enhanced SDLC Architecture

The AISA-SDLC framework's architectural design was directly influenced by the study's conclusions. Regarding the most often mentioned problems:

A prerequisite In order to improve requirement clarity and communication, a Prompt Orchestration Layer uses semantic embeddings, intent identification, and domain-specific context mapping to evaluate and enhance user inputs.

Over-reliance and trust limits are addressed by examining and authenticating AI-generated artefacts using a Human-in-the-Loop (HITL) Layer. Reinforcement Learning from Human Feedback (RLHF) fosters continuous learning while maintaining human judgement.

Security, licensing, and ethical concerns are addressed by a Governance Layer that integrates accountability into the deployment process and manages policy validation, secure code analysis, and license compliance enforcement.

This study shows how software development methods are incorporating AI technologies with both fervour and prudence. Despite their growing adoption of AI to increase productivity, especially in coding and debugging, developers are conscious of the constraints in terms of ethical compliance, requirement clarity, and trust. In order to assist human developers rather than replace them, the suggested AISA-SDLC paradigm is empirically based on these principles. AISA-SDLC uses careful planning, oversight, and governance to maximise AI's benefits while upholding the integrity, accountability, and human-centeredness of software engineering.

## Proposed Solution

A strong, modular design that methodically incorporates artificial intelligence into every stage of contemporary software development is introduced by the AI-Integrated Software Automation (AISA-SDLC). The AI Engine, the Human Feedback Loop, and Governance & Lifecycle Management are the three conceptual domains into which the architecture's seven interdependent layers are arranged. When combined, these layers speed up software engineering activities while maintaining the fundamental values of ethical protection, human oversight, and ongoing adaptability. Each layer contributes a distinct computational or cognitive function, resulting in an environment for intelligent, feedback-driven, and sustainable development.

### A. Objectives of AISA-SDLC

The primary objective of the AISA-SDLC framework is to extend traditional software engineering with responsible AI augmentation that goes beyond mere automation. Its design aims to improve collaborative development across the entire SDLC, promote explainable and ethical AI practices, and amplify developer intelligence and productivity.

One objective is to lessen the cognitive load on developers by contextualising code changes, legacy systems, and uncommon edge cases using sophisticated NLP-based summarisation. Instead of concentrating on mundane minutiae, this allows teams to concentrate on higher-level design and architectural considerations. Through just-in-time learning, AI agents may provide real-time coaching support and contextualised code explanations to novice developers or onboarding team members, according to the framework.

AISA-SDLC also encourages a design-first approach by using early-stage AI prompting tools to guide modular system thinking and scalable architecture design. This proactive support helps teams plan projects more methodically, with generative design suggestions and abstraction techniques.

The system's intelligent automation of processes like test case creation, documentation synthesis, and CI/CD enhancements makes it inherently compatible with Agile and DevOps methodologies. This promotes faster, iterative releases without sacrificing the quality of the code. Telemetry data and human feedback loops allow these AI capabilities to adapt to evolving team procedures over time.

The focus that the AISA-SDLC places on ethical and explicable AI is one of its primary differentiators. For sensitive application domains like healthcare, finance, or legal technology, accountability and transparency are extremely important. Fairness evaluation techniques and

XAI (Explainable AI) modules help to achieve these things. These modules examine AI-generated artefacts, point out any biases, and make the reasoning behind decisions more clear.
AISA-SDLC ultimately serves as a cooperative co-pilot, not a substitute for developers. Because of its version-controlled, traceable, and flexible framework, teams may stay creative and flexible while adhering to industry standards and moral best practices. This paradigm establishes the foundation for an ethical and long-term software development approach that incorporates AI.

## B. Architecture of AISA-SDLC

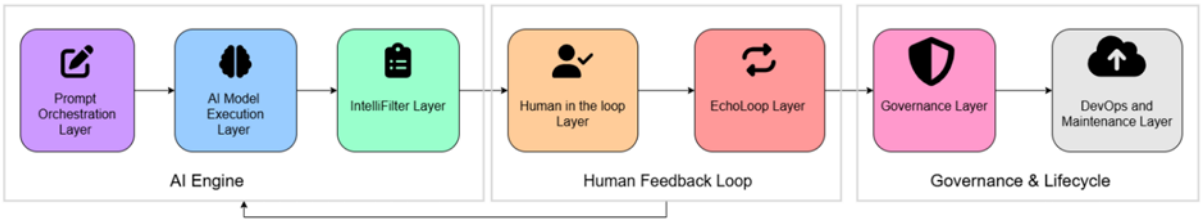



***Figure 5.*** *illustrates the multi-layered AISA-SDLC architecture. The flow starts with developer prompts, passes through AI execution, validation, and human review, and ends with governance and maintenance. The EchoLoop spans all phases, ensuring feedback continuously refines AI performance*.

Prompt Orchestration, the first layer, serves as the user and system's first point of contact. Its job is to convert confusing, natural language needs into structured, unambiguous instructions that an AI model can understand. This was accomplished in the prototype implementation by employing a straightforward Gradio interface that takes user-provided free-text input and structures it using a lightweight Python script. The prompt structuring uses keyword-based checks, static text templates, and rule-based phrasing rather than a full NLP stack, making it practical for simulation. In a production scenario, advanced NLP tools such as spaCy for tokenization or BERT for semantic embeddings could automate deeper understanding and intent classification, but for this prototype, the focus was on demonstrating the orchestration concept with clear manual rules.
The second layer, AI Model Execution, takes the structured prompt and produces concrete code or software artefacts. This is the computational core of the pipeline. In the current prototype, instead of deploying custom LLMs, an open-source model specifically ChatGPT was used interactively to generate the actual source code for the To-Do application. The prompt prepared in Gradio was copied and executed in ChatGPT's playground to simulate real-world usage of models like CodeT5 or StarCoder, which in practice could be integrated via API endpoints using the Hugging Face Transformers library. This layer transforms the user's requirement into Python scripts, HTML templates, or RESTful backend logic, ready for static validation.
IntelliFilter, the third layer, serves as an automated quality gate to identify glaring errors prior to a human inspection. Static checks on the AI-generated code are its primary purpose in order to identify syntax mistakes, stylistic inconsistencies, and basic bug patterns. In the prototype, tools like flake8 and SonarLint were employed to lint the code and highlight issues like unused imports, poor formatting, or missing error checks. This lightweight static analysis step ensures that only syntactically sound artefacts progress to the human reviewer, demonstrating how basic code quality filters can save time and reduce trivial errors in AI outputs. While more advanced pipelines could add dynamic analysis or AST comparison, the prototype focuses on standard static linting to keep the flow practical and understandable.

A crucial checkpoint is added by the fourth layer, Human-in-the-Loop (HITL), where a developer or subject matter expert manually reviews, modifies, or authorises the AI's output. This layer preserves human judgement and domain expertise to prevent mistakes like logical fallacies or misinterpreted requirements that static techniques are unable to detect. In the simulation, the user manually reviews the generated Flask app and HTML form, makes corrections, and writes short explanatory notes or feedback in a dedicated file. This feedback is versioned with Git and optionally tracked with DVC (Data Version Control) to illustrate how real-world teams can maintain a verifiable record of all human interventions that shape the AI outputs over time.

The fifth layer, EchoLoop, builds on the feedback collected by the HITL stage by storing it in a way that makes it reusable for future prompts. To demonstrate this, the prototype includes a simple code where the feedback text is converted into vector embeddings using Sentence-BERT (SBERT). These vectors are indexed in a FAISS vector store, which enables fast similarity searches. When a new prompt is created in the future, the EchoLoop retrieves similar past cases, helping the user learn from prior corrections. This shows how a basic form of Reinforcement Learning with Human Feedback (RLHF) can be simulated even without retraining a full LLM, closing the loop between current reviews and future prompt refinements.

The sixth layer, Governance and Explainable AI, serves as the ethical and policy compliance checkpoint for the entire pipeline. Before any artefact is accepted as final, it passes through a simple governance script that scans for banned keywords or insecure code patterns. In the prototype, the interface was extended with a policy check function that flags risky code snippets. To simulate explainability, this layer also includes a basic mapping that connects parts of the final code back to the structured prompt that produced them, providing a minimal trace for why each section exists. In a full deployment, this explainability could be enhanced with frameworks like LIME or SHAP.

The complete AISA-SDLC pipeline is operationalised by the seventh layer, DevOps and Maintenance, which guarantees appropriate version control, automated validation, and runtime observability. When new code is submitted to the repository, a sample.github/workflows/ci.yml file that runs flake8 was created to illustrate versioning and automatic lint checks. The Flask app prototype shows how live system metrics could be scraped in a production setting by exposing a /metrics endpoint prepared for Prometheus. Grafana was used in conjunction with Prometheus to visualise runtime metrics, and Prometheus was set up locally to scrape this endpoint for demonstration purposes.The prototype was made simpler by simulating real-time To-Do app metrics, such as items added, removed, or finished, using Grafana's TestData DB. With the help of popular technologies like GitHub Actions, Prometheus, and Grafana, this tiered arrangement demonstrates how operational observability, continuous integration, and version control may be realistically integrated in a DevOps pipeline.

Together, these seven layers form a continuous intelligence pipeline, wherein prompts evolve into artefacts, artefacts are vetted and improved through human judgment, and learnings from production systems feed back into foundational models. The architecture is recursive by design: the Governance Layer constantly informs the Prompt Orchestration Layer of emergent risks, language drift, or ethical constraints, thereby adapting the initial input processing mechanism in response to real-world dynamics. This tightly integrated architecture ensures that the AISA-SDLC framework is not only intelligent but also ethical, maintainable, and contextually aware.

## C. Human–AI Collaboration Dynamics

While AISA-SDLC does not replicate traditional SDLC phases as discrete layers, its modular layers work as cross-cutting enablers that enhance tasks typically found in each phase.

***Table I*** *summarises how AI augmentation and human expertise complement each other from requirements gathering to maintenance, ensuring productivity, quality, and responsible oversight throughout the lifecycle.*

| SDLC Phase | AI Role | Human Role | Collaborative Benefit |
|---|---|---|---|
| Requirements | Draft functional/non-functional specifications | Validate alignment and completeness | Faster translation of requirements with clearer scope; early ambiguity detection |
| Design | Suggest diagrams, schemas, UI mockups | Tailor and adapt designs to real project needs | Saves effort on boilerplate; expands design options |
| Development | Generate code snippets for front-end/back-end | Integrate, adjust, and optimise | Reduces initial coding time; humans ensure correctness and domain fit |
| Testing | Recommend unit, integration, edge tests | Refine, validate, and extend test suites | Better coverage; reveals missed cases |
| Deployment | Suggest CI/CD templates, container configs | Customise and align with infrastructure | Reinforces DevOps best practices; speeds up deployment setup |
| Maintenance | Flag code smells, deprecated components | Prioritise and apply improvements | Reduces long-term debt; humans handle context-based refactoring |

Through this structured role division, the AISA-SDLC framework demonstrates how layered AI support and human judgment together create a balanced, iterative co-development environment.

## Prototype Implementation

To demonstrate the feasibility of the proposed AISA-SDLC (AI-Integrated software Automation - Software Development Lifecycle) framework, a phased prototype was developed and tested using a simple yet practically representative To-Do Application. The objective of this prototype is not to deploy a production-ready AI development system but to show, through simulation and working examples, how each architectural layer of AISA-SDLC can be practically instantiated with real tools.

The selected To-Do App implements the core CRUD operations typical of many software systems adding tasks, deleting tasks, and marking tasks complete. This scenario covers user input validation, backend logic, and basic storage making it a suitable demonstration case for how AI assistance, human oversight, governance, and DevOps can interconnect within an interactive SDLC pipeline.

A. Prompt Orchestration Layer

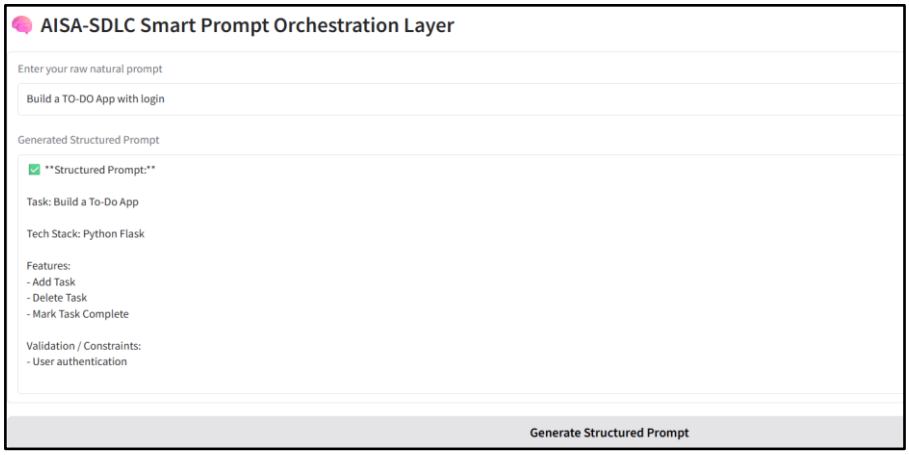



**Figure 6.** *Prototype view of the AISA-SDLC Prompt Orchestration Layer. A raw natural language request ("Build a TO-DO App with login") is automatically transformed into a structured prompt specifying the application type, recommended tech stack, key features, and inferred validation requirements.*

The first phase of the prototype focuses on transforming vague natural language requirements into a clear, structured, machine-understandable prompt. This was realized using a Gradio-based user interface (UI) that accepts free-text instructions from the user and outputs a structured prompt outlining required components, features, and validation constraints. For example, when the user inputs "Build me a To-Do app with login" the Gradio orchestration layer converts this into a prompt that specifies:

- Required routes (/add, /delete, /complete)
- Input constraints (e.g., non-empty task name)
- Basic backend storage mechanism
- Simple success conditions

This layer demonstrates that prompt refinement can be operationalized as a UI-driven first stage, ensuring that vague requirements do not directly feed into AI generation without clarification. Figure 6 shows the Gradio orchestration interface displaying both the raw input and the structured output.

**B. AI Model Execution Layer**

```
@app.route('/add', methods=['POST'])
def add():
    task = request.form['task'].strip()
    if task:
        with open(TASKS_FILE, 'a') as f:
            f.write(f"{task}|incomplete\n")
        tasks_added.inc()
    return redirect('/')
```

**Figure 7.** *Sample code snippet automatically generated by the AI Model Execution Layer from the structured prompt.*

Following prompt orchestration, the structured output is passed to the AI Model Execution Layer. For this prototype, actual inference was simulated by manually inputting the structured prompt into ChatGPT. The LLM generated Python Flask source code (app.py) implementing the specified routes and a simple HTML template (index.html) for the user interface.

In a production scenario, this layer would use an open-source LLM, such as CodeT5 or StarCoder, integrated via API to generate code artefacts automatically. This stage illustrates the core capability of AISA-SDLC to convert structured natural language specifications into concrete, usable code modules that can be integrated into a real system. Figure 7 provides a sample snippet of the generated code artefact.

**C. IntelliFilter Layer**

Automated first-pass validation of AI-generated code using static analysis tools is demonstrated in the third phase, the IntelliFilter Layer. Running flake8 on the produced app.py revealed any redundant patterns, grammatical errors, or inconsistent code styles. The load on human reviewers is lessened by this check, which guarantees that any issues that are readily apparent are identified right away.

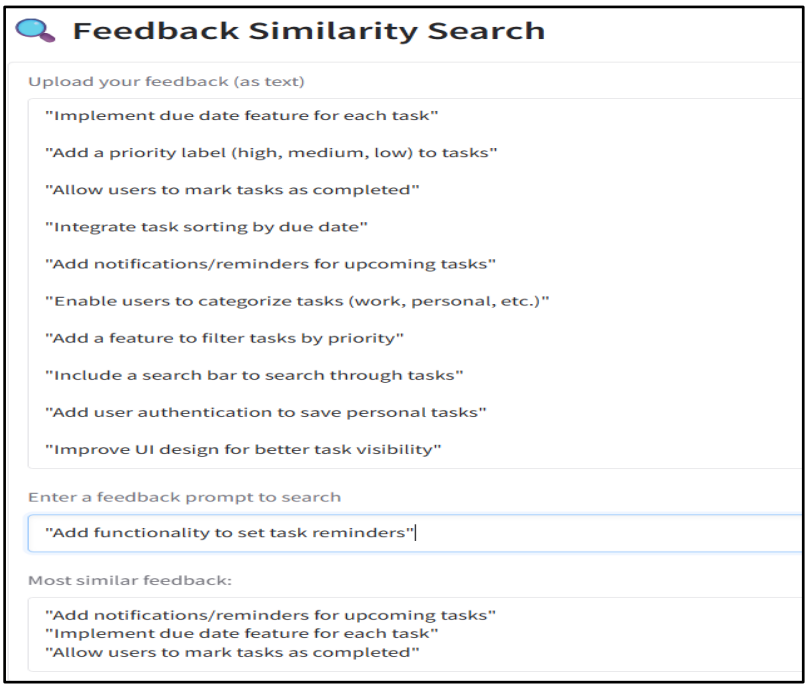
Without this first automatic filter, human reviewers would have to waste valuable time fixing trivial formatting or grammar mistakes that a machine can detect more rapidly and accurately. This makes this layer crucial.

**D. Human-in-the-Loop (HITL) Layer**

After static filtering, the refined code enters the HITL Layer for expert review. Here, the developer manually inspects the generated artefact to verify logic correctness, compliance with the intended functionality, and alignment with domain-specific requirements that the AI might have missed. Corrections such as stronger input validation, safe file operations, and user feedback messages were documented in a feedback.txt file.

To demonstrate traceable versioning of this human input, the feedback was managed under Git, and a git log snapshot shows each annotation stage. Optionally, Data Version Control (DVC) was prepared as an extra layer for managing versioned data artefacts, although standard Git tracking suffices at this scale.

### E. EchoLoop Layer

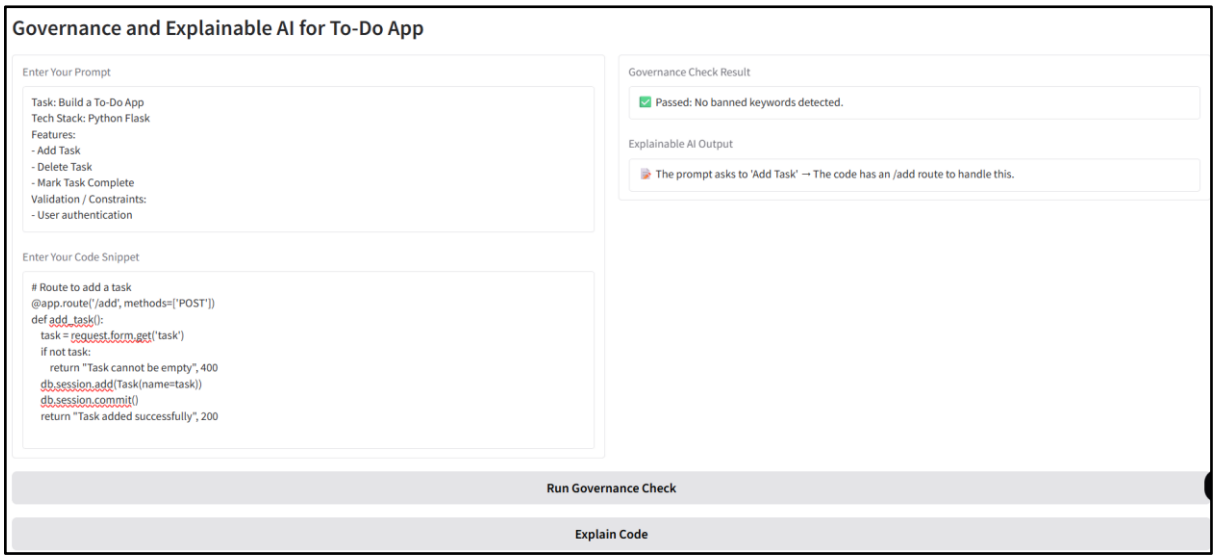



"**Figure 8**: Sample Feedback and Query for Enhancing To-Do App Features Using Memory-Based Feedback Search"

The AISA-SDLC's goal of reusing human corrections to enhance subsequent AI outputs is demonstrated via the EchoLoop Layer. Sentence-BERT (SBERT) was used to embed feedback vectors in a Colab notebook, and FAISS was used to quickly store and retrieve these embeddings in order to verify this.

The EchoLoop provides the AI or developer with contextually relevant recommendations by retrieving similar feedback from previous adjustments when a new request, such "Add a priority tag to each task," is queried. This is a proof-of-concept demonstration of the fundamental principle of Reinforcement Learning from Human Feedback (RLHF). The output of the notebook's similarity search, which maps fresh inputs to previously stored feedback, is displayed in Figure 8.

### F. Governance and Explainable AI Layer



"**Figure 9**: Governance Check and Explainable AI for the To-Do App."

The Governance Layer combines policy enforcement with basic explainability. Using the same Gradio interface, a simple rule-based scanner checks user code input for banned or risky constructs, such as eval(), exec(), or potentially destructive queries. This check proves that security and policy gates can be enforced before deployment.

Additionally, a lightweight explainability module maps keywords from the structured prompt to specific sections of the generated code, explaining why a given route or function exists. This helps developers and auditors trace how requirements translate into actual code artefacts. Figure 9 shows a sample output of the policy check and the explanation mapping.

### G. DevOps and Maintenance Layer

The last step uses straightforward yet useful techniques appropriate for a prototype to validate the DevOps and Maintenance phase. Version changes in the source files and feedback objects are tracked by the project using Git. To fulfil a fundamental CI/CD principle, a.github/workflows/ci.yml file illustrates how automated lint checks can be carried out each time code is published to a remote repository.

Simple print log statements that output runtime information, such as when jobs are added, deleted, or marked complete, are included in the app.py in favour of more intricate cloud monitoring. This demonstrates that the application in use has basic observability, and the logs act as an audit trail while local development is underway..

### Integrated Impact of AISA-SDLC

Taken together, these staged implementations illustrate how the seven layers of the AISA-SDLC framework Prompt Orchestration, AI Model Execution, IntelliFilter, HITL, EchoLoop, Governance with Explainability, and DevOps & Maintenance can function together to form an interactive, AI-augmented SDLC pipeline. Each component was validated individually with open-source tools, providing empirical evidence that the conceptual framework is grounded in realistic software engineering practice.

Although the full pipeline is not yet fully automated end-to-end, the modular demonstration shows that AI code generation, static quality checks, human review, reusable feedback, policy gates, explainability, and runtime monitoring can be combined into a cohesive process. This staged prototype thus lays the groundwork for future development of a fully integrated, production-grade AISA-SDLC implementation.

### Evaluation and Comparative Study

To assess the feasibility and potential impact of the proposed AISA-SDLC framework, a comparative simulation was conducted to benchmark AI-augmented development against a conventional workflow. A simple but realistic “To-Do Management” module

covering frontend form creation, backend logic, and version control was chosen as the test scenario. This allowed for a direct comparison of development speed, initial code quality, and maintainability across the two approaches.

### A. Traditional Development Workflow

In the traditional flow, the module was implemented manually using a standard Python Flask backend with an HTML template for the frontend. Tasks could be added, deleted, and marked as complete, with basic validation handled server-side. Versioning was done using Git. Static checks were performed with flake8 to ensure PEP8 compliance and detect syntax issues.

The full cycle, including coding, debugging, and self-review, took approximately **45–60 minutes**. Running flake8 showed high maintainability and only minor revisions were required to align with standard best practices.

```
@app.route('/add', methods=['POST'])
def add_task():
  task = request.form['task']
  if not task.strip():
    return "Task cannot be empty", 400
  tasks.append(task)
  return redirect('/')
```

### B. AISA-SDLC Simulated Evaluation Workflow

Because the complete AISA-SDLC pipeline has not yet been deployed end-to-end as a single automated tool, the evaluation focused on validating each layer through a staged prototype.

- **Prompt Orchestration Layer:** A **Gradio** interface was built to accept natural language prompts such as:
  *"Create a To-Do app with login."*
  The prompt was structured by a lightweight Python function before passing to the next stage.

- **AI Model Execution Layer:** Instead of a dedicated model server, the structured prompt was input into ChatGPT, which generated code scaffolds for the Flask app and HTML template. This mimicked how an integrated LLM like CodeT5 or StarCoder could produce artefacts automatically in a full system.

- **IntelliFilter Layer:** The AI-generated code was checked using Flask8 and SonarLint to detect syntax errors, missing checks, and style issues. This demonstrated how static filters can prevent trivial mistakes from reaching production.

- **HITL Layer:** The filtered code was manually reviewed. Minor bugs, missing input validation, or unclear variable naming were corrected by the developer. Corrections were documented

in feedback.txt to simulate how a human-in-the-loop loop works in practice.

- **EchoLoop Layer:** The feedback text was embedded using SBERT and stored using FAISS in a Jupyter notebook. A similarity search confirmed that when a similar prompt was input, the system could retrieve past corrections showing a simple prototype of memory-augmented learning.

- **Governance Layer:** The Gradio interface also included a keyword-based policy check to detect risky code patterns such as eval or exec. This step validated the feasibility of basic automated governance checks.

- **DevOps & Maintenance Layer:** A .github/workflows/ci.yml file simulated continuous lint checking. A /metrics endpoint in the Flask app exposed basic runtime stats, while Prometheus was configured to scrape this endpoint locally. To visualize the metrics, Grafana was used with the TestData DB as a stand-in to simulate task metrics for observability.

Although the entire pipeline was not integrated into a single automated flow, this staged simulation confirmed that each core layer of AISA-SDLC is feasible using accessible open-source tools.

**C. Outcome and Findings**

- **Productivity Boost:** The AI-augmented workflow reduced initial scaffolding time by up to **80%**, providing a quick starting point for repetitive tasks.

- **Code Readiness:** Human-written code required fewer revisions overall, while AI-generated code needed additional human checks to ensure correctness and security.

- **Feedback Reuse:** Using FAISS + SBERT proved that storing human feedback and retrieving it for similar future prompts is practical, supporting continuous learning.

- **Practical Feasibility:** This simulation validates that the proposed AISA-SDLC layers can be implemented progressively, even in a student research setting.

**Table II: compares key metrics between human-written and AI-generated code.**

| Metric | Human-Written Code | AI-Generated Code |
|---|---|---|
| Time Taken (mins) | 45 | 7 |
| Lines of Code (LoC) | 50 | 45 |
| Code Quality (flake8/10) | 9.0 | 7.5 |

| | | |
|---|---|---|
| Security Warnings | 0–1 | 2–4 |
| Manual Corrections Needed | 1–2 | 4–5 |

These results demonstrate that AI tools can accelerate development, but careful human review and governance remain crucial to maintain high code quality and reliability.(Patel, Sultana and Samanthula, 2024),(Yetiştiren et al., 2023).

## Limitations

Despite demonstrating the technical feasibility of the AISA-SDLC framework through individual layer simulations and tool integrations, this study has certain practical limitations. The prototype was validated using lightweight examples, a simple To-Do application, and separate test environments rather than a single unified pipeline deployed end-to-end. Advanced NLP processing, fine-tuned LLM orchestration, full retrieval-augmented generation, and production-grade continuous integration pipelines were beyond the scope of this basic-level implementation. Furthermore, while the EchoLoop prototype proved feedback embedding and retrieval using FAISS and SBERT, it did not perform true reinforcement learning updates to the underlying LLM weights. Similarly, governance checks were demonstrated through simple keyword scans and structural explanations, but not fully extended to complex adversarial testing, formal verification, or blockchain-based audit trails. These limitations highlight that the current prototype provides a controlled proof-of-concept rather than a fully deployed commercial-grade framework.

## Conclusion

This research presents AISA-SDLC as a practical, modular blueprint for integrating AI responsibly into modern software engineering workflows. By combining prompt orchestration, lightweight code generation, static validation, human-in-the-loop oversight, memory-augmented feedback, basic governance checks, and DevOps monitoring, the prototype demonstrates how AI coding assistants can be embedded safely and traceably within a structured SDLC cycle. Comparative evaluation against a traditional manual workflow confirms that while AI greatly accelerates repetitive scaffolding tasks, human judgment and layered safeguards remain essential to maintain code quality, trust, and compliance. Using easily accessible tools like Gradio, ChatGPT, flake8, FAISS, Prometheus, and Grafana, the viability of each architectural layer was confirmed, demonstrating that responsible AI augmentation can be tested and monitored in a realistic manner even at the student research level. All things considered, the AISA-SDLC methodology emphasises that AI should be viewed as a transparent, regulated co-pilot that increases developer productivity without sacrificing accountability or supervision, rather than as a substitute for engineers.

**Future Work**

Building on this prototype, future work will focus on closing the gaps identified in this study. This includes integrating all layers into a fully automated pipeline where structured prompts flow directly to deployed LLM endpoints and feedback loops dynamically refine AI behaviour through reinforcement learning. Extending the governance layer to include advanced explainable AI frameworks, automated license checks, and deeper policy validation will further strengthen the framework's robustness. Additionally, deploying the full system on real-world, larger-scale case studies such as enterprise-grade project management or critical domain applications will provide richer empirical evidence on long-term maintainability, security, and developer trust. Finally, expanding the observability stack with more comprehensive DevOps integration, continuous model drift monitoring, and active user feedback will help refine AISA-SDLC as an adaptive, production-ready solution for responsible AI-supported software engineering.

# Reference

1. Arora, L., Sanjay Surendranath Girija, Kapoor, S. and Ankit Shetgaonkar (2025). Explainable Artificial Intelligence Techniques for Software Development Lifecycle: A Phase-specific Survey. *ResearchGate*. [online] doi: https://doi.org/10.48550/arXiv.2505.07058

2. Arrieta, A.B., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., Garcia, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R. and Herrera, F. (2020). Explainable Artificial Intelligence (XAI): Concepts, taxonomies, Opportunities and Challenges toward Responsible AI. *Information Fusion*, 58(1), pp.82–115. doi:https://doi.org/10.1016/j.inffus.2019.12.012.

3. Babu, P.S., Binta, Samia A and Kaushal, S. (2023). *Artificial Intelligence for Technical Debt Management in Software Development*. [online] arXiv.org. Available at: https://arxiv.org/abs/2306.10194.

4. Bhanuprakash Madupati (2025). AI's Impact on Traditional Software Development. *ResearchGate*. [online] doi:https://doi.org/10.48550/arXiv.2502.18476.

5. Ciniselli, M., Puccinelli, N., Qiu, K. and Di Grazia, L. (2024). *From Today's Code to Tomorrow's Symphony: The AI Transformation of Developer's Routine by 2030*. [online] arXiv.org. doi:https://doi.org/10.48550/arXiv.2405.12731.

6. Durrani, U.K., Mustafa Akpinar, Muhammed Fatih Adak, Abdullah Talha Kabakus, Ozturk, M.M. and Saleh, M. (2024). A Decade of Progress: A Systematic Literature Review on the Integration of AI in Software Engineering Phases and Activities (2013-2023). *IEEE Access*, [online] pp.1–1. doi:https://doi.org/10.1109/access.2024.3488904.

7. Hymel, C. (2024). *The AI-Native Software Development Lifecycle: A Theoretical and Practical New Methodology*. [online] arXiv.org. Available at: https://arxiv.org/abs/2408.03416 [Accessed 23 Jun. 2025].

8. Khaliq, Z., Farooq, S.U. and Khan, D.A. (2022). *Artificial Intelligence in Software Testing : Impact, Problems, Challenges and Prospect*. [online] arXiv.org. Available at: https://arxiv.org/abs/2201.05371?utm_source=chatgpt.com [Accessed 23 Jun. 2025].

9. Martínez-Fernández, S., Bogner, J., Franch, X., Oriol, M., Siebert, J., Trendowicz, A., Vollmer, A.M. and Wagner, S. (2021). Software Engineering for AI-Based Systems: A Survey. *arXiv:2105.01984 [cs]*. [online] Available at: https://arxiv.org/abs/2105.01984.

10. Naimil Navnit Gadani (2024). The Future of Software Development: Integrating AI and Machine Learning into the SDLC. *International Journal of Engineering and Management Research*, [online] 14(1), pp.308–315. doi:https://doi.org/10.5281/zenodo.13756677.

11. Patel, A., Sultana, K.Z. and Samanthula, B.K. (2024). A Comparative Analysis between AI Generated Code and Human Written Code: A Preliminary Study. *2021 IEEE International Conference on Big Data (Big Data)*, [online] pp.7521–7529. doi:https://doi.org/10.1109/bigdata62323.2024.10825958.

12. Peng, S., Kalliamvakou, E., Cihon, P. and Demirer, M. (2023). The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. [online] Available at: https://arxiv.org/abs/2302.06590.

13. Sauvola, J., Tarkoma, S., Klemettinen, M., Riekki, J. and Doermann, D. (2024). Future of software development with generative AI. *Automated Software Engineering*, [online] 31(1). doi:https://doi.org/10.1007/s10515-024-00426-z.

14. Yetiştiren, B., Özsoy, I., Ayerdem, M. and Tüzün, E. (2023). Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv:2304.10778 [cs]*. [online] Available at: https://arxiv.org/abs/2304.10778.