**SOFTWARE DEVELOPMENT LIFE CYCLE: An Agentic approach**

**Abstract**  
The Software-Development Life Cycle (SDLC) is the time-tested framework for converting stakeholder needs into dependable, maintainable software. Yet the hand-offs between SDLC phases remain labor-intensive, slow, and vulnerable to misinterpretation — especially on projects that must iterate rapidly. We introduce a **six-agent SDLC-automation pipeline** that aligns lightweight, specialised AI agents with every canonical phase: Requirements Curator, Design Synthesiser, Offline Developer, Test-Scenario Generator, Test-Case Generator, Report Generator, and a planned Maintenance/Refinement agent. Each runs offline on locally hosted large-language models (LLMs), exchanges artefacts through a shared vector store, and outputs phase-specific deliverables that become the structured inputs of the next agent. Mapping autonomous agents onto the traditional SDLC preserves the model’s conceptual clarity while injecting the adaptability and parallelism of multi-agent systems. Early qualitative observations indicate smoother phase transitions, faster turnaround on change requests, and stronger traceability from stakeholder intent to executable tests. This paper details the architecture, behaviours, and orchestration strategy that underpin the pipeline, laying a reproducible foundation for fully automated software-production workflows. ​

**1 Introduction**

The Software Development Life Cycle (SDLC) has long served as the foundational model for organizing complex software projects into a series of structured, traceable stages. It segments software engineering into discrete phases requirements gathering, system design, implementation, testing, deployment, and maintenance allowing teams to manage complexity through modular thinking and accountability. This structure has proven its value across decades of industrial practice, under various process models including waterfall, spiral, V-model, Agile, and DevOps-enabled hybrids.

Yet even the most sophisticated SDLC implementations encounter a persistent friction: **the human hand-off**. Between every stage lies a gap where information must be reformulated, reinterpreted, or recontextualized—often manually. Stakeholder intent gets translated into user stories, which must then be converted into designs, then code, then tests. Each transition introduces potential for error, misalignment, and delay. The overhead from these transitions becomes especially problematic in modern settings where rapid iteration, continuous delivery, and stakeholder responsiveness are paramount.

Recent advances in artificial intelligence particularly in **large language models (LLMs)** offer a compelling opportunity to reimagine how these transitions are handled. LLMs have demonstrated impressive capabilities in structured reasoning, code generation, summarization, and multi-turn dialogue. However, their true potential is unlocked when used not as monolithic assistants, but as **collaborative agents**, each focused on a specific goal and stage within a larger pipeline.

This paper proposes a novel **agentic architecture** for SDLC automation, where each traditional phase is supported by a specialised, autonomous agent that transforms its inputs into structured outputs. Rather than replace the SDLC, our goal is to **preserve its conceptual clarity** while enhancing its execution with the adaptability and speed of LLM-based systems. The approach enables improved traceability, faster response to changing requirements, and reduced manual effort—all while keeping models locally hosted and artefacts version-controlled.

**1.1 The Six-Agent Pipeline**

**Rather than rely on a single, general-purpose AI model, we delegate discrete responsibilities to specialised agents aligned with each SDLC phase. This division of labour mirrors the roles found in traditional software teams, but with the added benefit of determinism, repeatability, and auditability. Table 1 outlines the six core agents in our current implementation.**

| **#** | **Agent** | **SDLC Phase** | **Core Responsibilities** |
| --- | --- | --- | --- |
| **1** | **Requirements Curator** | **Requirements** | **Harvests and clusters stakeholder needs; produces structured user stories with acceptance criteria.** |
| **2** | **Design Synthesiser** | **System Design** | **Translates stories into architecture sketches, data-flow diagrams, and design-pattern recommendations.** |
| **3** | **Developer** | **Implementation** | **Generates code stubs, configures scaffolding, integrates modules.** |
| **4** | **Test-Scenario Generator** | **Test Planning** | **Creates high-level “Given-When-Then” scenarios linked to user stories.** |
| **5** | **Test-Case Generator** | **Verification** | **Converts scenarios into executable unit and integration tests.** |
| **6** | **Report Generator** | **Quality Reporting** | **Analyzes test results and produces coverage and traceability metrics.** |

**A seventh Maintenance Agent is under development, tasked with monitoring changes in business requirements or codebases and initiating appropriate updates across the pipeline.**

**1.2 Research Objectives**

This study explores three questions:

* **Phase Integrity:** Can specialised agents preserve requirements-to-test traceability better than ad-hoc human hand-offs?
* **Effort Reduction:** What proportion of traditionally manual SDLC labour (story writing, boilerplate coding, test authoring) can the pipeline automate without loss of quality?
* **Adaptation Latency:** How quickly can the pipeline converge on a stable artefact set after mid-stream requirement changes?

**2 Methodology**

To operationalize the agentic SDLC pipeline, we structured each phase around a **core team of three complementary agents**: a **Developer Agent**, a **Validator Agent**, and a **Corrector Agent**. This triad ensures that generated artefacts are not only created autonomously but also undergo systematic validation and refinement before being passed downstream. The result is a more resilient, traceable, and self-correcting system.

**2.1 Agent Roles and Responsibilities**

Each agent within a core team plays a distinct role:

* **Developer Agent**: Responsible for generating the initial artefact for the phase. This agent performs the primary transformation (e.g., from requirements to user stories, or from designs to code).
* **Validator Agent**: Assesses the artefact produced by the Developer Agent. It checks for logical consistency, completeness, alignment with inputs from previous phases, and conformance to best practices.
* **Corrector Agent**: Takes feedback from the Validator Agent and iteratively improves the artefact until it meets quality criteria or reaches a confidence threshold.

This three-agent structure balances creativity, scrutiny, and refinement—mimicking the roles of creators, reviewers, and maintainers in traditional software teams.

**2.2 Orchestration Strategy**

Agents are orchestrated using **LangGraph**, a multi-agent coordination framework that allows for structured, event-driven workflows. Each phase of the SDLC is treated as a micrograph with its own flow logic, branching rules, and shared context. Inter-agent communication is managed through a combination of:

* **Phase-specific memory**: Used to store artefacts, intermediate validations, and feedback loops.
* **Global vector store**: Acts as the persistent memory across all agents, ensuring context continuity and enabling traceability.
* **Standardized input/output schemas**: Each agent adheres to strict I/O contracts to guarantee compatibility across phases.

The entire pipeline is hosted locally, enabling secure and reproducible runs without relying on third-party APIs.

**2.3 Phase-wise Agent Descriptions**

Each SDLC phase in our agentic pipeline is managed by a dedicated agent that assumes end-to-end responsibility for the artefacts and decisions of that phase. Below is a detailed breakdown of the agents, their target SDLC phases, and their core functionalities:

**1. Requirements Curator (Requirements Phase)**

The Requirements Curator is tasked with capturing the voice of the stakeholder. It ingests raw requirements whether from documents, transcripts, or issue trackers and clusters them by thematic relevance. Using domain-specific templates and prompt chaining, the agent generates structured user stories, each with embedded acceptance criteria. These artefacts are stored in a vector database for traceable retrieval in subsequent phases.

* **Developer Agent:** Generates structured user stories from stakeholder inputs.
* **Validator Agent:** Ensures all inputs are captured and stories follow acceptance criteria patterns.
* **Corrector Agent:** Refines poorly formed or ambiguous stories.

**2. Design Synthesiser (System Design Phase)**

Once the user stories are established, the Design Synthesiser interprets them into technical blueprints. It creates architecture diagrams, module boundaries, and data-flow charts using LLM-guided reasoning and design-pattern libraries. It recommends patterns like MVC, Observer, or Factory where applicable, ensuring that each design decision is tied back to the original user story. These outputs are validated for completeness and feasibility before downstream processing.

* **Developer Agent:** Produces high-level design diagrams and component layouts such as Functional and Technical Specifiactions.
* **Validator Agent:** Checks for architectural soundness, separation of concerns, and design-pattern compliance.
* **Corrector Agent:** Adjusts inconsistencies or suboptimal patterns.

**3. Developer (Implementation Phase)**

The Developer generates skeletal codebases aligned with the approved design. It scaffolds the project repository (e.g., directory structure, config files, build scripts), implements core modules in the designated programming language, and inserts documentation where necessary. It also integrates standard libraries, dependencies, and APIs, preparing the codebase for testing. The generated code is syntax-checked and modular, facilitating unit and integration testing in later stages.

* **Developer Agent:** Generates code and integrates modules.
* **Validator Agent:** Reviews for syntax correctness, logical integrity, and dependency consistency.
* **Corrector Agent:** Refactors and aligns code to project conventions.

**4. Test-Scenario Generator (Test Planning Phase)**

This agent bridges functional requirements and executable tests. The Test-Scenario Generator extracts user stories and reformulates them into high-level test scenarios using the Given-When-Then (GWT) structure. These scenarios emphasize user behaviour and system responses, creating a testable abstraction of the software’s expected behaviour. Scenarios are indexed for traceability and aligned with both the requirements and code modules.

* **Developer Agent:** Extracts structured user stories and their acceptance criteria from the shared vector store and transforms them into high-level test scenarios using the “Given-When-Then” (GWT) format. This abstraction captures the expected system behavior from a user perspective and ensures coverage of both functional and edge-case flows.
* **Validator Agent:** Reviews the GWT scenarios for logical completeness and alignment with the originating user stories. It verifies that each acceptance criterion is reflected in at least one scenario and that the language used is unambiguous, traceable, and aligned with domain standards.
* **Corrector Agent:** Refines scenarios that contain redundant or vague steps, normalizes naming conventions for reusability, and splits overly complex scenarios into modular, testable components. This agent also flags inconsistent user behavior flows for manual review if the correction confidence is low.

Top of Form

Bottom of Form

**5. Test-Case Generator (Verification Phase)**

Building on the test scenarios, the Test-Case Generator produces executable test scripts—both unit and integration level. It selects appropriate testing frameworks (e.g., JUnit, PyTest, GTest) based on project metadata and synthesizes test functions, mock objects, and assertions. Test coverage reports are produced in tandem, indicating which portions of the codebase are validated. The tests are verified for correctness, independence, and minimal duplication.

* **Developer Agent**: Consumes validated GWT scenarios and transforms them into executable unit and integration test scripts using suitable testing frameworks (e.g., PyTest, JUnit, Mocha). It constructs mocks, test data, and assertions that align with the scenario logic, and injects metadata tags for traceability.
* **Validator Agent**: Ensures each generated test aligns with the original scenario intent, executes without errors in a sandbox environment, and covers the defined positive and negative paths. It cross-references the test results against previously defined acceptance criteria to identify gaps.
* **Corrector Agent**: Edits or regenerates test cases that produce false positives/negatives, contain syntactic issues, or fail to execute. It also optimizes test structure by removing redundancy, minimizing external dependencies, and enhancing logging and diagnostics for easier debugging.

**6. Report Generator — Continuous Quality Reporting (After Every Phase)**

Rather than operating only at the end of the SDLC, the Report Generator is invoked **immediately after the completion of each phase**, producing lightweight, phase-specific reports that promote early error detection, traceability, and progress monitoring.

* **Developer Agent**: Aggregates artefacts and metrics from the just-completed phase (e.g., number of user stories curated, design completeness, test generation success rate). It compiles structured reports that include status summaries, traceability links, and milestone validation.
* **Validator Agent**: Verifies the integrity and accuracy of reported data, checks that key metrics (like story coverage, scenario/test-case density, code quality scores, etc.) fall within predefined thresholds, and flags any inconsistencies or missing artefacts for review.
* **Corrector Agent**: Rectifies reporting errors such as incorrect mappings, missing traceability links, or improperly formatted visualizations. It also enhances report clarity through improved phrasing, standardized tables/graphs, and relevant summaries tailored to stakeholders (e.g., QA leads, PMs, or devs).

This continuous reporting strategy ensures that each phase transition is **transparent, traceable, and measurable**, reducing debugging overhead and promoting smoother hand-offs in the automation pipeline.

**2.4 Feedback Loops and Auto-Correction**

At each phase, a feedback loop between the Validator and Corrector agents ensures that artefacts are continuously improved until they meet the defined success criteria. If a phase cannot converge within a fixed number of iterations, it flags the artefact for human review, ensuring that quality is never sacrificed.