SP-202 Agentic Software Debugger & Documenter: Software Design Document

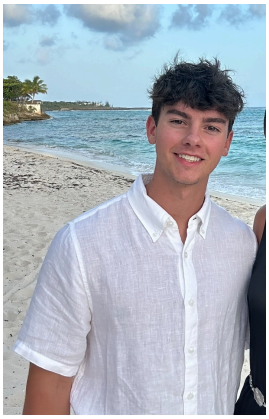
CS 4850 — Sec 02 — Spring 2026

January 27, 2026

Sharon Perry

Jade Le

Team Lead
Documentation & Testing



Preston Dietz

Development & Testing

Omodemi Olaoluwa

Development & Testing

Table of Contents

1. Introduction and Overview

The Agentic Software Debugger & Documenter is a modular system based on the concept of agents, which automates the debugging and documentation processes using a multi-stage approach. This means that the debugging and documentation processes are not performed in a single step, unlike in previous approaches where the entire code was run in a single pass for debugging and documentation. Instead, the agentic approach divides the debugging and documentation processes into separate agents coordinated using the `SequentialAgent` and `LoopAgent` patterns from Google's Agent Development Kit (ADK). The code snippet provided by the developer progresses in a pipeline where the `Bug Detection` agent, the `Fixer` agent, and the `Documentation` agent perform specific roles in the software architecture.

The architecture also allows for iteration and deterministic validation to improve the reliability of the results when using the LLMs (Large Language Models). After the attempt to fix the code, the generated code is validated using a language-specific linter to check for errors. If the validation fails, the results are refined in a loop to improve the results within certain limits. The architecture is based on the concept of agents, where each agent is responsible for a specific role in the software architecture, ensuring the results are obtained in a cost-effective manner.

2. Design Considerations

2.1. Assumptions and Dependencies

2.1.1. Dependencies

- **Programming Language and Runtime**
    - Python 3.10+
    - Local runtime environment capable of executing and validating code snippets.
- **Agent Framework**
    - Google Agent Development Kit (ADK) for implementing SequentialAgent and LoopAgent patterns.
    - Support for tool invocation and structured (JSON) agent outputs.
- **Large Language Models (LLMs)**

    - Google Gemini models (`gemini-1.5-flash` or `gemini-2.0-flash`) accessed through Google AI Studio.

- **Linting and Validation Tools**
    - Language-specific linters selected dynamically based on the detected programming language.
        - **Python**: `pylint` / `flake8`

2.1.2. Assumptions

- **Input Constraints**
    - Submitted code snippets are relatively small and self-contained.
    - Code does not require external services, databases, or network access.
- **Agent Reliability**
    - LLM agents may produce imperfect outputs; therefore, validation and refinement loops are required.
    - Structured outputs (e.g., JSON) are assumed to be consistently parseable.
- **Controlled Iteration**
    - A maximum number of refinement iterations (e.g. 3) is enforced to prevent infinite loops and API overuse.
    - Failure after the maximum iteration count results in a graceful fallback response.

2.2. General Constraints

2.2.1. Hardware and Software Environment

The system must run on commodity hardware and standard operating systems (Windows, macOS, Linux) without requiring specialized processors or proprietary tools.

- **Impact on Design:**
    - Execution and analysis components must be lightweight and modular.

- Language analysis tools (e.g. linters, formatters) are containerized or sandboxed to avoid OS-specific dependencies.
- The system avoids heavy runtime dependencies and relies on widely supported languages (e.g. Python) for orchestration.

2.2.2. End-User Environment

End users may have varying levels of technical expertise and may access the system via a command-line tool.

- **Impact on Design:**
  - The user interface must be intuitive and abstract away internal complexity.
  - Error messages and documentation output must be human-readable and explanatory rather than purely technical.
  - Sensible defaults are provided to minimize required configuration.

2.2.3. Standards Compliance

The system must comply with established programming and documentation standards (e.g., PEP 8 for Python, Javadoc conventions for Java).

- **Impact on Design:**
  - Linting and formatting rules are based on widely accepted standards.
  - The system supports extensibility to allow future standards to be added without major refactoring.
  - Documentation generation follows conventional structures to ensure familiarity and acceptance.

2.2.4. Interface and Protocol Requirements

Communication between system components must be reliable and secure.

- **Impact on Design:**

- Components communicate via well-defined APIs or message-passing interfaces.
- Inputs/outputs are validated to prevent malformed or malicious data from propagating through the system.
- Clear versioning is used to prevent compatibility issues.

2.2.5. Quality Attribute Trade-offs

Improving one quality attribute (e.g. security) may negatively impact others (e.g. performance).

- **Impact on Design:**
  - Security is prioritized over raw execution speed.
  - Modularity is favored to improve maintainability, even at the cost of slight overhead.
  - Design decisions are documented to justify trade-offs.

3. Architectural Strategies

This project adopts an agent-oriented architectural strategy rather than a monolithic or purely deterministic design.

3.1. Rationale for Agentic Architecture

Instead of implementing a single procedural script that detects bugs, applies rule-based fixes, runs a linter, and generates documentation, this system breaks up those responsibilities into specialized agents coordinated using Google ADK's `SequentialAgent` and `LoopAgent` patterns.

3.2. Key Architectural Patterns

Sequential Pipeline Pattern

- Bug Detection → Fixer → Linter → Documentation
- Each stage produces structured output consumed by the next

Controlled Iteration (`LoopAgent`)

- Fix / Validate cycle repeats
- Maximum of 3 iterations (required in SRS)
- Prevents infinite loops and API overuse

Tool Augmented Reasoning

- Agents may invoke:

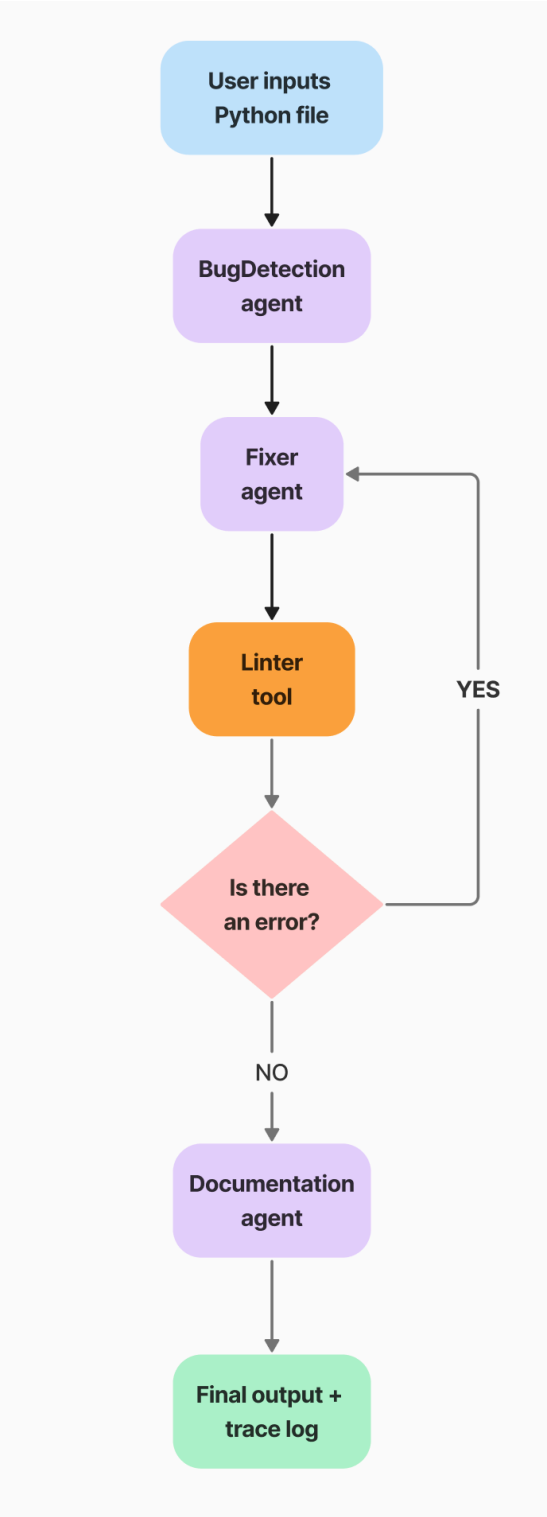
- Linter tool
- Patch formatter
- JSON schema validator

4. System Architecture

4.1. High-Level Architecture Overview

1. **Interface Layer**
    a. CLI interface
    b. Handles user input file path
    c. Displays run status
    d. Outputs artifacts (corrected code, documents, logs).
2. **Orchestration Layer**
    a. Implements `SequentialAgent`
    b. Implements `LoopAgent`
    c. Controls iteration count
    d. Manages state transitions
3. **Agent Layer**
    a. `BugDetection` agent
    b. `Fixer` agent
    c. `Documentation` agent
4. **Tool Layer**
    a. Linter runner (`pylint` / `flake8`)
    b. JSON schema validator
    c. Trace logger

4.2. Data Flow



`SequentialAgent` would ensure ordered execution and `LoopAgent` would wrap `Fixer` + linter until the linter passes or the 3 iterations have been reached.

5. Detailed System Design

5.1. Agent Overview

5.1.1. Bug Detection Agent

The Bug Detection Agent automatically identifies defects, vulnerabilities, and potential performance issues in code before deployment. Its goal is to reduce manual debugging effort and improve overall code reliability.

Inputs

- Python source code file (string or file)

Outputs

- List of detected bugs (structured JSON)
  - Issue's line number
  - Issue type (syntax, runtime, logic, etc.)
  - Issue description
  - Severity level
- Status flag: `issues_found` (true/false)

5.1.2. Fixer Agent

The Fixer Agent automatically generates patches or corrected code based on issues identified by the Bug Detection Agent.

Inputs

- Bug report (from Bug Detection Agent)
- Original Python source code
- Current iteration count (for loop control)

Outputs

- Corrected code snippet
- Summary of changes made (what was changed and what issues were addressed)
- Fix status: `fix_attempted` (true), `remaining_issues_possible` (true/false)

5.1.3. Documentation Agent

The Documentation Agent automatically generates clear, structured documentation for codebases, APIs, and system components. It ensures maintainability and developer onboarding efficiency.

Inputs

- Final validated Python source code
- Summary of fixes made (optional)

Outputs

- Markdown documentation:
  - High-level description
  - Function summaries
  - Parameter descriptions
  - Return values
- Documentation generation status

5.2. Inputs & Outputs

System Inputs

- CLI argument: Python file path

System Outputs

- Corrected Python code
- Linter result summary
- Markdown documentation
- Structured trace log (JSON)

5.3. Internal Logic & Loop Conditions

The refinement loop stops when:

1. Linter reports zero errors **OR**
2. Iteration count is 3

If iteration count is 3 or more:

- System returns best available version
- Linter errors included in output

5.4. Interfaces

5.4.1. CLI Interface

The Command Line Interface (CLI) serves as the primary entry point for interacting with the multi-agent system. It enables developers to configure execution parameters, monitor runtime progress, and access generated outputs.

Features

- **Help Text (`--help`)**

  The CLI provides a help command that displays usage instructions, available options, argument descriptions, and execution examples. This ensures usability and reduces configuration errors.

- **Iteration Status Display**

  During execution, the CLI displays structured progress updates, including:

  - Current iteration number
  - Active agent stage
  - Number of issues detected
  - Validation results
  - Completion status

  This feature improves transparency and supports debugging during development.

- **Output Directory Display**

  Upon completion, the CLI displays the location of generated artifacts. This includes:

  - Bug reports
  - Generated patches
  - Documentation files
  - Execution logs

  The output directory path is explicitly shown to ensure discoverability and reproducibility.

5.4.2. Tools Interfaces

The Linter Runner interface integrates external static analysis tools into the agent pipeline. It standardizes how linting tools are executed and how their results are returned to agents.

Responsibilities

- Detect programming language
- Execute appropriate linter
- Capture raw output
- Parse results into structured format
- Return standardized issue objects

Output Structure

The Linter Runner returns structured findings that include:

- File location
- Line number
- Rule identifier
- Description of issue
- Execution status

This abstraction ensures tool outputs remain consistent regardless of the underlying linter.

5.4.3. Trace Logger:

The Trace Logger records all agent actions and tool interactions during execution. It supports observability, debugging, and reproducibility.

Responsibilities

- Record agent invocations
- Log tool execution details
- Capture execution timestamps
- Record duration metrics
- Log errors and recovery actions

Trace logs are stored as structured artifacts to allow post-execution analysis.

5.5. Data & State Management

5.5.1. Context Management

The system maintains execution context throughout each run. The orchestrator manages global state, while individual agents operate with scoped contextual inputs.

Context includes:

- Current code snapshot
- Detected issues
- Patch history
- Iteration count

5.5.2. Memory Model

- **Short-Term Memory**

  Maintains temporary data during execution, including:
  - Active issue list
  - Intermediate tool outputs
  - Candidate patches

- **Long-Term Memory (Optional Extension)**

  May store recurring bug patterns, historical fixes, or system-level insights to improve future performance.

- **Artifacts**

  Artifacts are persistent outputs generated during execution. These include:

  - Bug report files
  - Patch files
  - Documentation updates
  - Execution trace logs

  Artifacts are versioned per iteration to ensure traceability and rollback capability.

5.6. Failure Modes & Recovery

5.6.1. Timeouts

If a tool or agent exceeds the predefined execution time:

- The process is terminated
- The failure is logged
- The system proceeds with partial results where possible
- Retry logic may be applied within defined limits

5.6.2. Invalid JSON

If an agent produces malformed structured output:

- Automatic repair attempts are performed
- A reformat request is issued
- If failure persists, the iteration is aborted and logged

This prevents pipeline corruption due to formatting errors.

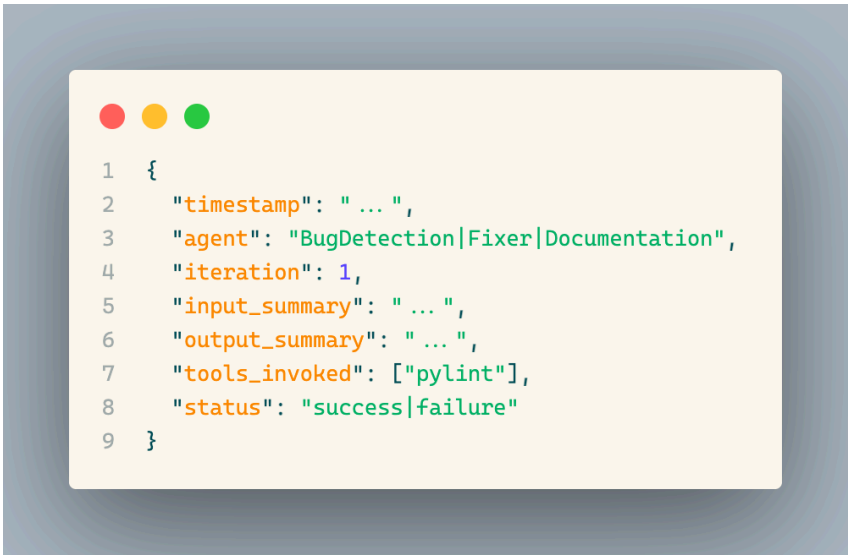
5.6.3. Linter Failure

If the linter execution fails due to configuration errors or missing dependencies:

- The error is logged
- The system falls back to available static analysis mechanisms
- Execution continues where feasible

6. Trace Logging and Observability

6.1. Trace Log Structure



6.2. Observability Strategy

Achieved through:

- Structured JSON trace logs
- Iteration counter visibility in CLI
- Explicit logging of tool invocations
- Recording linter results per iteration

7. Bibliography

https://google.github.io/adk-docs/agents/workflow-agents/sequential-agents/

https://google.github.io/adk-docs/agents/workflow-agents/loop-agents/

https://google.github.io/adk-docs/agents/multi-agents/