SP-202 Agentic Software Debugger & Documenter: Final Report

CS 4850 — Sec 02 — Spring 2026

Preston Dietz, Jade Le, Olaoluwa Omodemi

Sharon Perry

April 17, 2026

Website: https://agentic-debugger.github.io/

Github: https://github.com/Agentic-debugger/Debugger

Stats:

- ~168,148 lines of code
- 12 required tools/components
    - google-adk
    - python-dotenv
    - pytest
    - pydantic
    - rich
    - uvicorn
    - fastapi
    - python-multipart
    - sse-starlette
    - markdown
    - weasyprint
    - httpx
- Total man hours: ~ 145 hours

Status: 87% complete

Table of Contents

1. Introduction

The Agentic Software Debugger & Documenter is a multi-agent system built with Google's Agent Development Kit (ADK) that autonomously analyzes Python source code to identify bugs, apply iterative fixes, validate corrections, and generate documentation. The system was developed as the capstone project for CS 4850, Spring 2026.

The system employs a sequential pipeline architecture: a Bug Detection Agent identifies defects in the submitted code, a Fixer Agent applies targeted corrections, a LoopAgent orchestrates iterative fix-and-validate cycles up to a maximum of three attempts, and a Documentation Agent produces structured Markdown output describing the final corrected code. A linter serves as the objective quality gate between each fix iteration, and a trace logger records all agent actions and tool calls to ensure full observability of the autonomous decision-making process.

This report summarizes the project's requirements, design decisions, development process, and test outcomes across all phases of the software development lifecycle.

2. Requirements

2.1. Functional Requirements

- The multi-agent system shall accept a Python source code file as input via CLI.
- The multi-agent system shall autonomously identify potential bugs in the provided code.
- The multi-agent system shall attempt code corrections using rule-based and LLM-driven tools.
- The multi-agent system shall validate corrected code using a linter after each fix attempt.
- Based on linter results, the system shall either trigger a refinement iteration (on errors) or proceed to documentation generation (on success).
- The multi-agent system shall limit refinement iterations to a maximum of three attempts per input.
- The multi-agent system shall terminate the refinement loop and return the best available output if the maximum iteration limit is reached.
- The multi-agent system shall generate structured Markdown documentation from the final corrected output.
- The multi-agent system shall generate structured trace logs for each execution session, capturing all agent actions and tool calls.

2.2. Nonfunctional Requirements

- The system shall provide clear CLI prompts and help text indicating required inputs, run status (iteration count), and output file locations.
- Trace logs shall be produced in a structured format (JSON) that can be parsed programmatically for evaluation.
- The system shall always terminate after at most 3 refinement iterations and shall not run indefinitely.
- The system shall not execute code with network access and shall not access external databases or modify files outside the provided input.
- The system's workflow shall be modular — separate agents for detection, fixing, and documentation — to support future replacement or extension of individual agents without redesigning the full pipeline.
- The system shall run on commodity hardware and standard operating systems (Windows, macOS, Linux) without requiring specialized processors or proprietary tools.
- Linting and formatting rules shall comply with established programming standards (e.g., PEP 8 for Python).

3. Analysis & Design

3.1. Agentic Architecture Rationale

- Traditional debuggers and static analyzers reveal program state or style violations but lack semantic reasoning about programmer intent and cannot coordinate multi-step workflows.
- A deterministic script can apply rules but cannot propose reasonable candidates when information is ambiguous or missing.
- The agentic approach (specialized agents iteratively proposing and validating fixes) improves debugging workflow quality compared to a single-pass deterministic tool, while trace logs and iteration limits keep autonomy observable and bounded.

3.2. Key Architectural Patterns

3.2.1. Sequential Pipeline

Bug Detection → Fixer → Linter → Documentation. Each stage produces structured output consumed by the next, ensuring separation of concerns and independent testability.

3.2.2. Controlled Iteration (LoopAgent)

The fix/validate cycle repeats up to a maximum of 3 iterations (as required by the SRS). The loop terminates on CLEAN (zero linter

errors), NO_IMPROVEMENT (error count unchanged between iterations), or MAX_ITERATIONS_REACHED. This prevents infinite loops and bounds API usage.

3.2.3. Tool-Augmented Reasoning

Agents may invoke the linter runner, a patch formatter, and a JSON schema validator. The FixerEngine applies fast rule-based fixes deterministically (mutable defaults, bare excepts, None comparisons) before invoking the LLM for any issues requiring contextual reasoning.

3.3. System Layers

- **Interface Layer** — CLI handles user input (file path), displays run status and iteration count, and outputs artifacts (corrected code, documentation, logs).
- **Orchestration Layer** — SequentialAgent and LoopAgent control execution order, iteration count, and state transitions.
- **Agent Layer** — BugDetectionAgent, FixerAgent, and DocumentationAgent, each responsible for a discrete role.
- **Tool Layer** — Linter runner (pylint / flake8), JSON schema validator, and trace logger.

3.4. System Layers

The SequentialAgent ensures ordered execution across the pipeline. The LoopAgent wraps the Fixer and linter until the linter passes or three iterations have been reached. All intermediate state (bug reports, fix iterations, linter output) is held in memory during a single pipeline run and is not persisted between sessions.

4. Development

- Baseagent.py serves as the foundation class all other agents inherit from. It centralizes shared initialization: loading the API key from the .env file, instantiating the Gemini model via Google ADK, creating the ADK runner, and establishing a session context. This eliminates boilerplate duplication across specialized agents.
- Detector.py analyzes a Python source file and returns a structured JSON object listing each detected issue, including line number, category label (e.g., MUTABLE_DEFAULT, BARE_EXCEPT, NONE_COMPARISON), and a plain-language description.
- Fixer.py and Loop.py implement the fix-and-retry pipeline. FixerEngine first applies fast rule-based fixes with no API call, then the LLM handles remaining issues

requiring reasoning. The pipeline passes an iteration counter into fix_file() each cycle; MAX_ITERATIONS = 3 is enforced at the top of fix_file() so that exceeding the cap immediately returns MAX_ITERATIONS_REACHED with the best available code.

- Documentation.py converts the final corrected code into structured Markdown documentation covering code summaries, detected issues, applied fixes, and explanations of changes. Markdown was chosen for its readability and compatibility with tools such as GitHub.
- A key development challenge was API rate limiting on the free tier of the Google Gemini API. The multi-agent pipeline triggers several API calls per run, and the iterative fix loop can cause rapid successive requests that are throttled. This was addressed through iteration caps and deterministic pre-processing to minimize unnecessary LLM calls.

5. Tests and Results

5.1. Integration Testing

Integration testing validated the full end-to-end pipeline, including agent-to-agent handoffs, linter integration, LoopAgent termination conditions, and CLI behavior. Integration tests required a live GOOGLE_API_KEY and were executed against the Gemini model in a local Python 3.10 environment.

Test execution summary:

Category	Count
Unit Tests (no API dependency)	10
Integration / E2E Tests (require live Gemini API key)	4
Grand Total	**14**

All 10 unit tests passed in an offline environment. All 4 integration tests passed with a valid GOOGLE_API_KEY present in .env.

BaseAgent Tests (test_baseagent.py)

Test Case	Type	Scope	API Required	Result
test_env_var_missing	Negative / Unit	Unit	No	PASS
test_agent_creation	Smoke / Unit	Unit	No	PASS
test_agent_run_async	Functional / Integration	Integration	Yes	PASS
test_agent_run_sync	Functional / Integration	Integration	Yes	PASS

BugDetectionAgent (test_Detector.py)

Test Case	Type	Scope	API Required	Result
test_forensic_workflow	Integration / E2E	Integration	Yes	PASS*

* Input/output validation — the detector was provided intentionally malformed Python code and output was reviewed to confirm all expected issue categories were identified.

Fixer Agent / Fixer Engine (test_fixer.py)

Test Case	Type	Scope	API Required	Result
test_fixer_engine_on_sample	Unit	Unit	No	PASS
test_extract_bug_report	Unit	Unit	No	PASS
test_fix_file_no_issues	Unit / Edge Case	Unit	No	PASS
test_iteration_cap	Unit / Boundary	Unit	No	PASS
test_full_fix_on_sample	Integration	Integration	Yes	PASS

Documentation Agent (test_documentation.py)

Test Case	Type	Scope	API Required	Result
test_documentation_build_and_save	Unit / Functional	Unit	No	PASS

Validator / Linting Engine (test_validator.py)

Test Case	Type	Scope	API Required	Result
test_validator_findings	Unit	Unit	No	PASS

5.2. Test Case Results

ID	Test	Description	Expected Result	Actual Result
TC-01	Bug Detection: Basic syntax error	Ensure Detector catches syntax errors	DetectorEngine.analyze() catches SyntaxError via try/except	DetectorEngine.analyze() wraps ast.parse() in try/except and explicitly handles SyntaxError

ID	Test	Description	Expected Result	Actual Result
TC-02	Bug Detection: AST only mode	Run with --no-llm-detection	AST findings returned; no Gemini merge	AST findings returned; no Gemini merge
TC-03	Fixer: Deterministic fixes	Mutable default argument	Fixer rewrites default to None and adds guard clause	Fix is logged; guard clause (if items is None: items = []) is not inserted
TC-04	Fixer: LLM fix application	Complex logic bug requiring LLM	LLM returns fenced code block; Fixer extracts and writes file	LLM returns fenced code block; Fixer extracts and writes file
TC-05	LoopAgent: CLEAN termination	Linter returns zero errors after iteration 1	Loop stops; documentation begins	Loop stops; documentation begins
TC-06	LoopAgent: MAX_ITERATIONS_REACHED	Code cannot be fully fixed	Loop stops at 3 iterations; status = PARTIAL	Loop stops at 3 iterations
TC-07	LoopAgent: NO_IMPROVEMENT	Error count unchanged between iterations	Loop stops early	Loop stops early
TC-08	Linter integration	Validator returns structured error list	Errors passed to Fixer as new bug report	Errors passed to Fixer as new bug report
TC-09	Documentation Agent	Generate Markdown report	Report includes detector findings, fix logs, loop summary, and final code	Report includes detector findings, fix logs, loop summary, and final code
TC-10	CLI: Missing file	Run with nonexistent path	Exit code 1; error message printed	Exit code 1; error message printed
TC-11	Trace logging	Validate trace log structure	Log includes timestamps, agent actions, iteration count, and errors	All information captured; structured logging module not yet implemented

5.3. Detection Rule Coverage

Rule Code	Severity	Covered By
MUTABLE_DEFAULT	Warning	test_validator_findings

Rule Code	Severity	Covered By
NON_SNAKE_CASE_FUNCTION	Warning	test_validator_findings
EVAL_EXEC_USAGE	Error	test_validator_findings
BARE_EXCEPT	Error	test_validator_findings, test_fixer_engine_on_sample
NONE_COMPARISON	Warning	test_validator_findings
HARDCODED_SECRET	Error	test_extract_bug_report (mock)
OPEN_NOT_IN_WITH	Warning	Defined in rules.py — no unit-level test coverage
GLOBAL_VARIABLE_USAGE	Warning	Defined in rules.py — no unit-level test coverage
UNUSED_IMPORT	Info (backlog)	Not yet emitted by detector
MISSING_TYPE_HINTS	Info (backlog)	Not yet emitted by detector
SHADOWED_BUILTIN	Info (backlog)	Not yet emitted by detector

6. Version Control

Version control for this project was managed using Git, with the repository hosted on GitHub under the organization account Agentic-debugger. GitHub served as the central coordination point for collaboration, code reviews, issue tracking, and release management. This approach aligns with the Version Control Plan defined early in the project, which specified Git/GitHub as the standard due to its industry adoption, flexibility, and support for team workflows.

6.1. Repository Structure and Organization

The repository was structured to mirror the modular architecture of the system. Each subsystem—Agents, Linting, Orchestration, Documentation, and Reports—was placed in its own directory to support separation of concerns and parallel development. The final directory layout is:

```
SP202/
├── Agents/
│   ├── Baseagent.py      # ADK + Gemini wiring
│   ├── Detector.py       # AST + optional Gemini merge
│   ├── Documentation.py  # Markdown report
│   ├── Fixer.py          # Deterministic + LLM fixes
│   ├── Loop.py           # Fix → validate → retry
│   └── test_*.py
├── Documentation/        # Course design PDFs (if present)
```

```
├── Linting/
│   ├── rules.py
│   ├── validator.py
│   └── test_validator.py
├── Orchestration/
│   ├── controller.py     # CLI
│   ├── pipeline.py       # run_pipeline()
│   └── state.py          # PipelineState
├── reports/              # Generated reports (example / local)
├── bad_code_sample.py
├── requirements.txt
└── README.md
```

This structure ensured that each agent, validator, and orchestration component could be developed and tested independently while maintaining a clear, navigable project layout. It also supported traceability between requirements, implementation, and testing.

7. Summary

This project delivers a multi-agent system capable of autonomously detecting bugs in Python code, proposing and validating fixes through an iterative loop, and generating structured Markdown documentation. Built using Google's Agent Development Kit (ADK) and Gemini models, the system integrates a Bug Detection Agent, Fixer Agent, LoopAgent, and Documentation Agent into a coordinated workflow that improves code quality while maintaining strict iteration limits and producing transparent trace logs. The pipeline accepts a Python file via CLI, performs static and LLM-augmented analysis, applies deterministic and LLM-based corrections, validates results with a custom linting engine, and outputs corrected code, validation summaries, and a complete debug report.

Throughout development, the team designed modular components, implemented robust error-handling and JSON-structured outputs, and addressed challenges such as prompt formatting, ADK integration, and iterative loop behavior. Testing with diverse code samples demonstrated both the strengths and limitations of agentic refinement, reinforcing the value of controlled autonomy and observability. The final system showcases how multi-agent workflows can enhance debugging beyond traditional deterministic tools, offering a foundation for future extensions such as expanded language support, improved fix-generation strategies, and richer documentation capabilities.