

1. Title

AI-Powered Code Reviewer and Quality Assistant

AI-Powered Code Reviewer and Quality Assistant

2. Project Statement

Modern codebases evolve rapidly, often without consistent review quality or standardized practices. Manual code reviews are time-intensive and depend heavily on reviewer expertise. This project builds an **AI-assisted code review tool** that automatically analyzes Python code for style, performance, and potential bugs. It leverages static analysis, large language models (LLMs), and code embeddings to provide **actionable feedback** in pull requests or CI pipelines. Developers can interact through a **CLI** or an optional **Streamlit UI**, reviewing, accepting, or ignoring suggestions.

By integrating into Git workflows, the tool helps maintain high-quality code standards while reducing reviewer workload.

3. Outcomes

- A CLI tool for AI-assisted code reviews on Python projects.
- Identifies potential issues: unused imports, complexity spikes, missing tests, poor variable naming, etc.
- Suggests improvements with natural-language explanations.
- Configurable linting and rule sets (PEP8, custom checks).
- Optional Streamlit dashboard for reviewing AI suggestions.
- Integration with Git pre-commit and CI/CD to enforce quality gates.

4. Modules to be Implemented

1. Code Parsing & Analysis

- Parse Python files via ast and static analysis.
- Extract structure: imports, classes, functions, complexity, dependencies.
- Detect smells: long functions, deeply nested loops, missing type hints.

2. AI Review Engine

- Use LLM-based prompt templates to generate human-like code feedback.

- Rank findings by severity (info, warning, critical).
- Optionally auto-fix simple issues (naming, docstrings, spacing).

3. Validation & Metrics

- Evaluate code quality scores per file and overall.
- Track maintainability index, complexity metrics, and coverage hints.
- Export reports (CSV/HTML).

4. CLI & Configuration

- Commands: scan, review, apply, report, diff.
- Configurable rules in pyproject.toml.
- Supports severity thresholds and excluded paths.

5. VCS & CI Integration

- Git pre-commit hook to auto-review staged files.
- CI templates for GitHub/GitLab enforcing quality gates.

6. Review Web UI (Optional)

- Streamlit interface for visualizing issues and fixes.
- Side-by-side diff view with AI suggestions.

5. Week-wise Module Implementation & High-Level Requirements Milestone 1 (Weeks 1–2) – Parsing & Baseline Generation

- Implement AST-based extractor for functions, classes, and modules.
- Generate baseline docstrings in Google style.
- Produce initial docstring coverage report.

Milestone 1: Parsing & Baseline Generation

Weeks 1-2 Implementation Output

The screenshot displays the user interface for Milestone 1. It includes three main sections: 'Project Files' on the left, 'AST Parsing Output' in the center, and 'Generated Docstrings & Coverage Report' on the right.

- Project Files:** Shows a list of Python files with their parsing progress:
 - data_processor.py: 57%
 - utils.py: 73%
 - models.py: 50%
 - api.py: 92%
 - main.py: 70%A terminal window shows the command: `$ docstring-generator scan --path ./src`. Below it, a summary indicates 95% Parser Accuracy and 42 Functions.
- AST Parsing Output:** Displays an example of Python code parsed with AST, along with extracted metadata about the `calculate_average` function.
- Generated Docstrings & Coverage Report:** Provides a breakdown of generated docstrings and coverage reporting for two methods:
 - `DataProcessor.__init__`: Initialize the DataProcessor with data. Args: data (list): The data to be processed.
 - `DataProcessor.process_item`: Process a single data item. Args: item (str): The data item to process. index (int, optional): Position index. Defaults to 0. Returns: str: The processed data item.Coverage details include Generated Docstrings and Coverage Report tabs.

Milestone 2 (Weeks 3–4) – Synthesis & Validation

- Add support for **NumPy** and **reST styles**.
- Improve generation with Raises/Yields/Attributes sections.
- Integrate **pydocstyle checks** and coverage reporting.

Milestone 2: Synthesis & Validation

Weeks 3-4 Implementation Output

The screenshot shows a Streamlit application interface for Milestone 2. It includes three main sections: 'Docstring Styles' (listing Google, NumPy, and reStructuredText styles), 'Generated Docstrings' (displaying a sample docstring for a 'process_item' function), and 'Validation Results' (showing a bar chart of compliance, warnings, and errors, and a list of validation findings for 'data_processor.py', 'utils.py', and 'api.py').

Docstring Styles

- Google
- NumPy
- re

Generated Docstrings

```
Google Style
...
Process a single data item.
Args:
item (str): The data item to process.
index (int, optional): Position index. Defaults to 0.
strict (bool, optional): Whether to enforce strict validation.
Defaults to False.
Returns:
Optional[str]: The processed item in uppercase, or None if empty.
Raises:
ValueError: If strict is True and item is empty.
...
```

Validation Results

Compliant: 45, Warnings: 5, Errors: 2

File	Details
data_processor.py	All docstrings follow PEP 257 standards
utils.py:23	D400: First line should end with a period
api.py:45	D205: Blank line missing after one-line docstring

Milestone 3 (Weeks 5–6) – Workflow & CI

- Add **pre-commit hook** and **CI workflow** with coverage enforcement.
- Support configuration via `pyproject.toml`.
- Build Streamlit **review UI prototype**.

Milestone 3: Workflow & CI

Weeks 5-6 Implementation Output

The screenshot shows a Streamlit application interface for Milestone 3. It includes three main sections: 'Configuration' (displaying a `.pre-commit-config.yaml` file), 'CI Pipeline' (listing five steps: Checkout Code, Setup Environment, Scan & Generate Docstrings, Validate Coverage Threshold, and Generate Report), and 'Streamlit Review UI' (showing a dashboard with files and a diff viewer for the 'process_item' function).

Configuration

```
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/docstringgen/pre-commit
    rev: v1.0.0
    hooks:
      - id: docstring-generator
        args: [-style=google, --threshold=90]
        files: "src/*.py"
        tool:
          docstring-generator:
            style = "google"
            include = ["src/**/*.py"]
            exclude = ["**/_init_.py"]
            coverage_threshold = 90
            autofix = true
```

CI Pipeline

- 1 Checkout Code
- 2 Setup Environment
- 3 Scan & Generate Docstrings
- 4 Validate Coverage Threshold
- 5 Generate Report

Streamlit Review UI

Docstring Review Dashboard

- data_processor.py (3)
- utils.py (2)
- api.py (1)

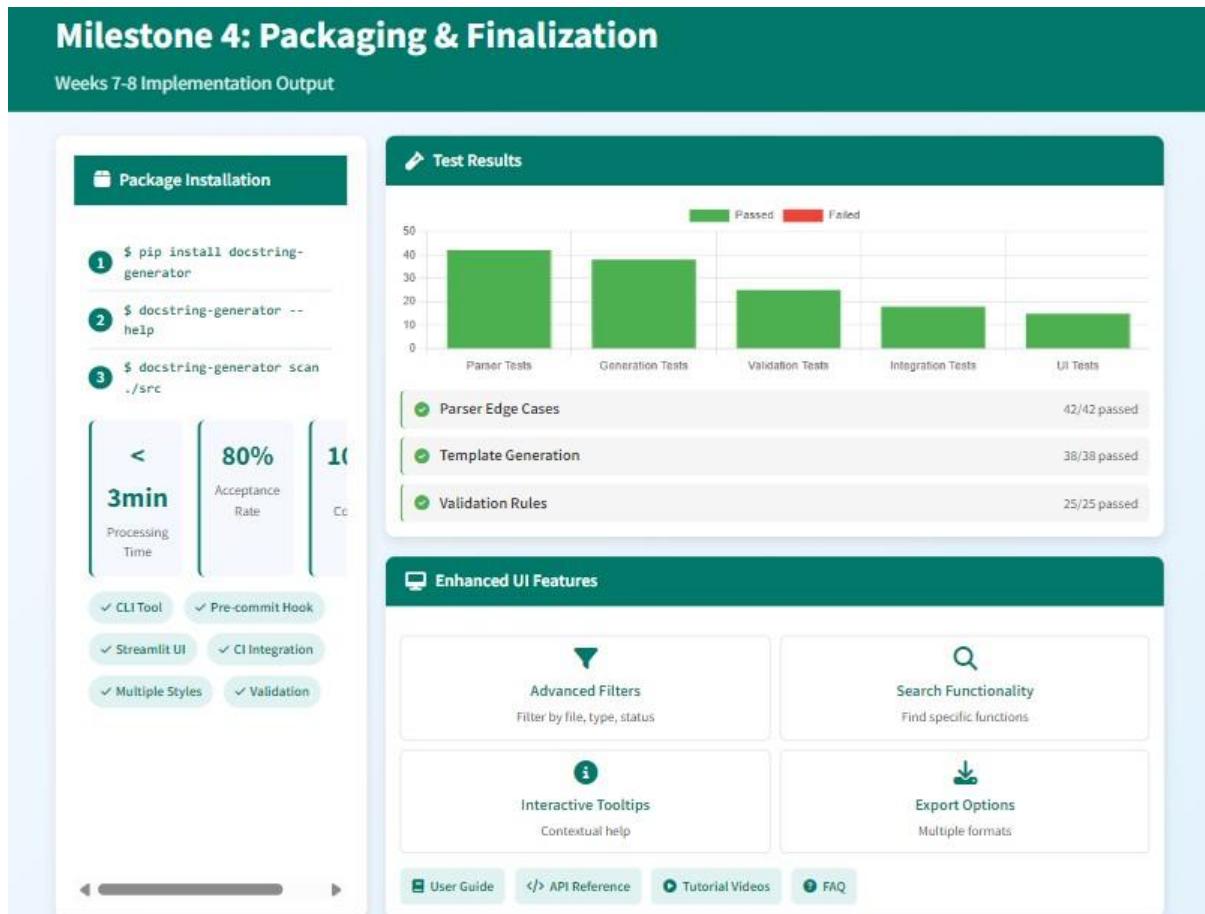
process_item function

```
25 def process_item(self, item, index=0):
26     """Process a single data item"""
27     if strict and not item:
28         Args:
```

Reject Accept

Milestone 4 (Weeks 7–8) – Packaging & Finalization

- Package tool as a **pip-installable library**.
- Add robust tests for edge cases.
- Improve Streamlit UI (filters, search, tooltips).
- Publish documentation and usage guides.



6. Evaluation Criteria

Milestone 1 (Week 2)

- Parser extracts ≥95% of functions/classes without errors.
- Baseline docstrings generated for all missing cases.
- Coverage report generated successfully.

Milestone 2 (Week 4)

- Generated docstrings conform to **PEP 257**.
- Support for at least 3 docstring styles.
- Coverage ≥90% for target sample repositories.

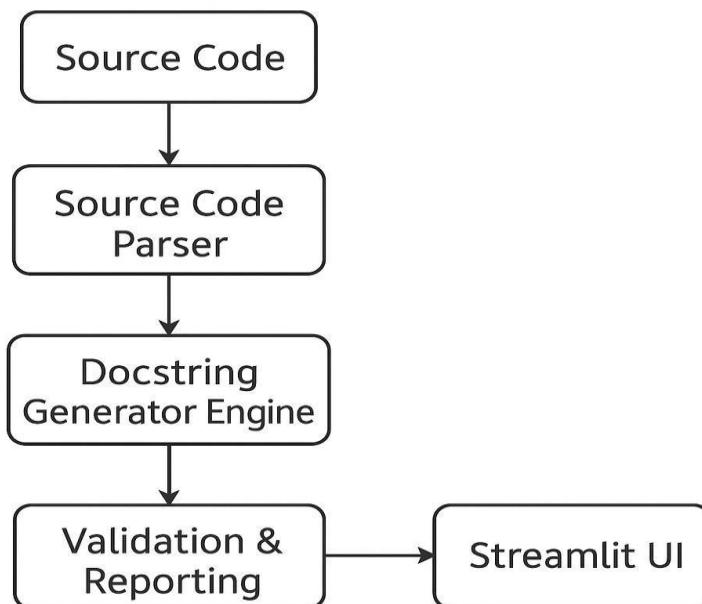
Milestone 3 (Week 6)

- CI pipeline blocks builds under configured coverage threshold.
- Pre-commit runs in <5s on staged changes.
- Streamlit UI supports previewing and accepting/rejecting docstrings.

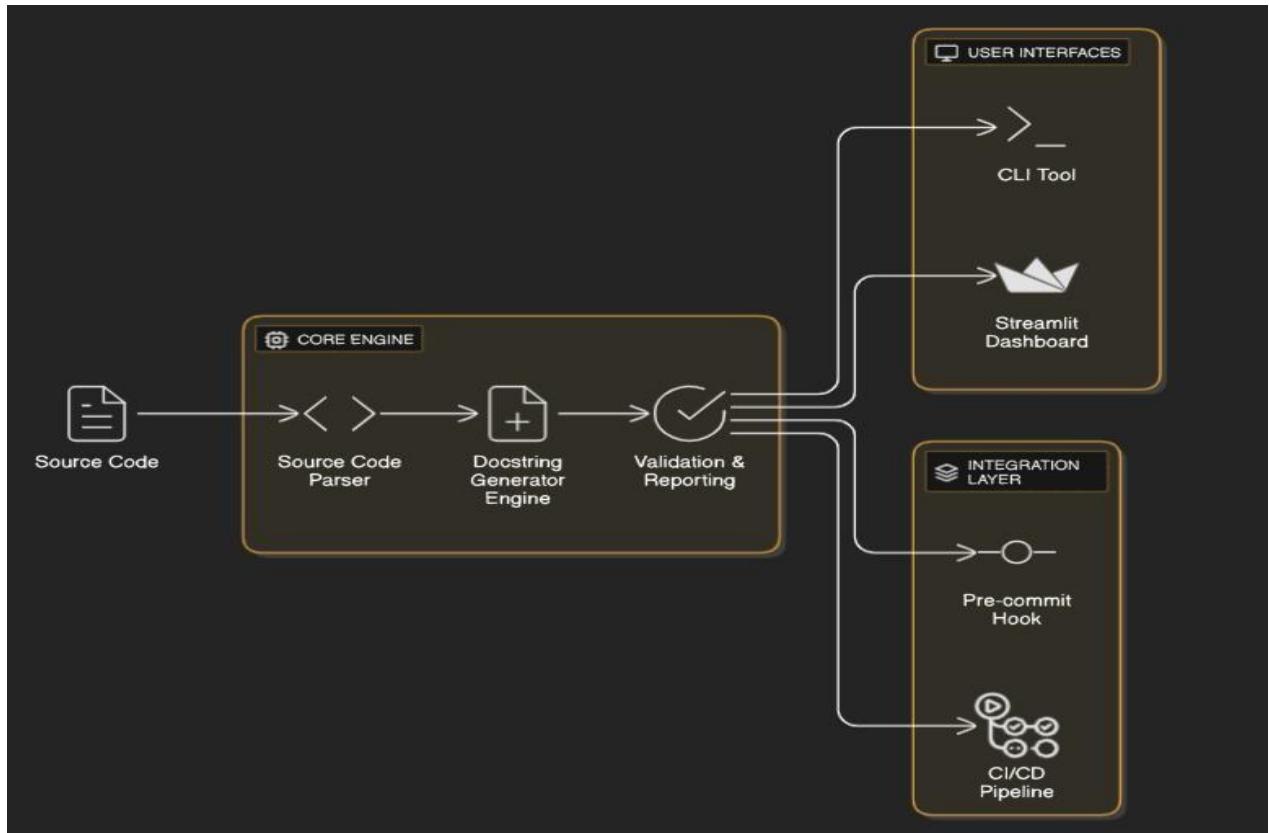
Milestone 4 (Week 8)

- Tool installable via pip with documentation.
- Works end-to-end on medium projects (<2k functions) in <3 minutes.
- Developers report ≥80% acceptance rate of generated docstrings.

7. Workflow Diagram



8. Architecture Diagram



9. Database Schema

