AI-Powered Code Reviewer and Quality- by Dhikshita

1.TITLE:

AI-Powered Code Reviewer and Quality assistant

2.PROJECT STATEMENT:

In modern software development, codebases grow rapidly and are often maintained by multiple developers with varying levels of experience. Ensuring consistent code quality through manual code reviews has become increasingly challenging. Traditional code reviews are time-consuming, depend heavily on individual reviewer expertise, and may overlook important issues such as code complexity, unused components, missing documentation, and poor coding practices.

There is a need for an **automated, intelligent, and developer-friendly code review system** that not only detects issues in Python code but also explains them clearly and suggests improvements. Such a system should integrate easily into development workflows, support command-line usage, and optionally provide a graphical interface for better visualization of issues.

This project aims to address these challenges by developing an **AI-Powered Code Reviewer and Quality Assistant** that combines static code analysis with AI-based feedback generation. The tool helps developers maintain high code quality standards, reduces manual review effort, and supports learning through clear, actionable suggestions.

3. PROJECT OBJECTIVES

The main objectives of this project are:

- To design and implement an automated code review system for Python projects.
- To analyze Python source code using static analysis techniques.
- To detect common code quality issues such as unused imports, high complexity, missing type hints, and code smells.
- To generate human-readable review comments with severity levels.
- To provide a command-line interface (CLI) for easy usage.
- To lay the foundation for future integration with AI models, Git workflows, and web-based dashboards.

4. SCOPE OF THE PROJECT

The scope of the project includes:

- Static analysis of Python source code using the ast module.
- Rule-based AI-like review generation (with scope for future LLM integration).
- Generation of code quality metrics and reports.
- Command-line interaction for scanning and reviewing code.
- Report generation in structured formats.

STRUCTURE OF SOURCE CODE:

```
AI-Powered-Code-Reviewer/
│
├── cli/
│   ├── __init__.py
│   └── commands.py
│
├── core/
│   ├── docstring_engine/
│   │   ├── __init__.py
│   │   └── generator.py
│   │
│   ├── parser/
│   │   ├── __init__.py
│   │   └── python_parser.py
│   │
│   ├── reporter/
│   │   ├── __init__.py
│   │   └── coverage_reporter.py
│   │
│   ├── review_engine/
│   │   ├── __init__.py
│   │   └── ai_review.py
│   │
│   └── validator/
│       ├── __init__.py
│       └── validator.py
│
├── examples/
│   ├── sample_a.py
│   └── sample_b.py
│
├── experiments/
│   ├── llm_groq.py
│   └── llm_local.py
```

```
|
├── storage/
|   └── reports/
|       └── review_logs.json
|
├── tests/
|   ├── __init__.py
|   └── test_parser.py
|
├── env/
|
├── main_app.py
├── pyproject.toml
├── requirements.txt
└── README.md
```

5. MILESTONE 1: Core Code Analysis and Review Engine

5.1Milestone Description

Milestone 1 focuses on building the **foundation of the AI-Powered Code Reviewer**. This phase establishes the core architecture, implements static code analysis, and produces AI-like review feedback using heuristic rules. The goal is to create a fully functional prototype that can analyze Python files and generate meaningful review suggestions via a CLI.

This milestone ensures that the system works end-to-end without external AI dependencies, making it stable, testable, and easy to extend in future phases.

5.2 Functional Components Implemented in Milestone 1

1. Code Parsing & Analysis Module

- Uses Python’s built-in ast (Abstract Syntax Tree) module.
- Parses Python source files to extract:
    - Imports and unused imports
    - Functions and methods
    - Classes and their structure
    - Line numbers and code boundaries
- Estimates cyclomatic complexity using branching constructs.

2. AI Review Engine (Heuristic-Based)

- Generates human-readable review comments.
- Assigns severity levels:
    - **Info** – suggestions and improvements
    - **Warning** – potential maintainability issues
    - **Critical** – severe complexity or design problems

3. Validation & Quality Metrics Module

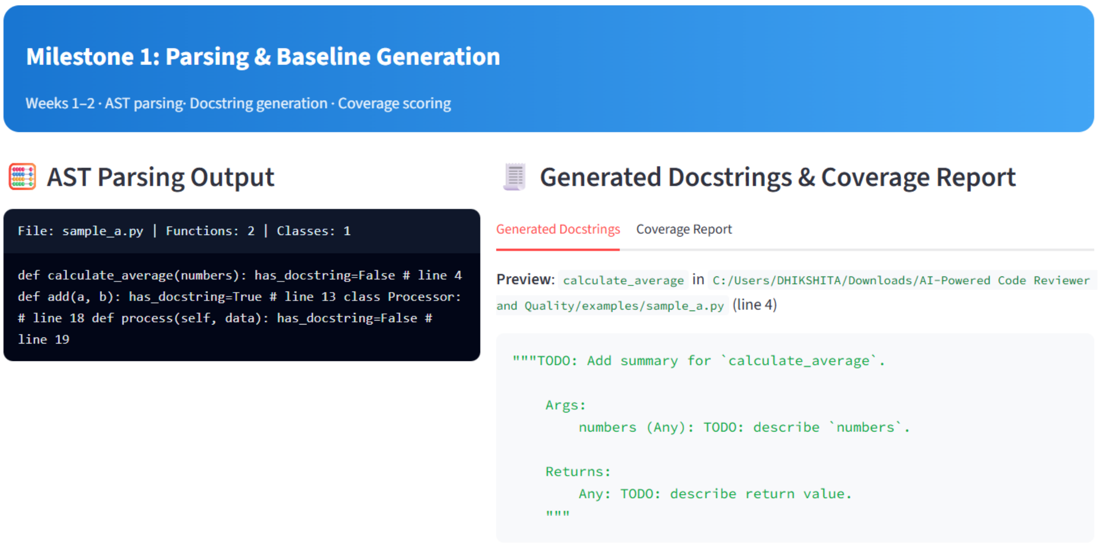
- Computes a **code quality score (0–100)** for each file.
- Penalizes:
    - Unused imports
    - High complexity
    - Missing type hints
    - Excessive review comments
- Produces per-file quality metrics for maintainability assessment.

4. CLI Interface

- Provides a simple and intuitive command-line interface.

- Supported commands:
  - scan – Analyze code structure and metrics
  - review – Display AI-generated review comments
  - report – Generate structured review reports
- Allows severity filtering during reviews.

OUTPUT:



Milestone 1: Parsing & Baseline Generation

Weeks 1–2 · AST parsing· Docstring generation · Coverage scoring

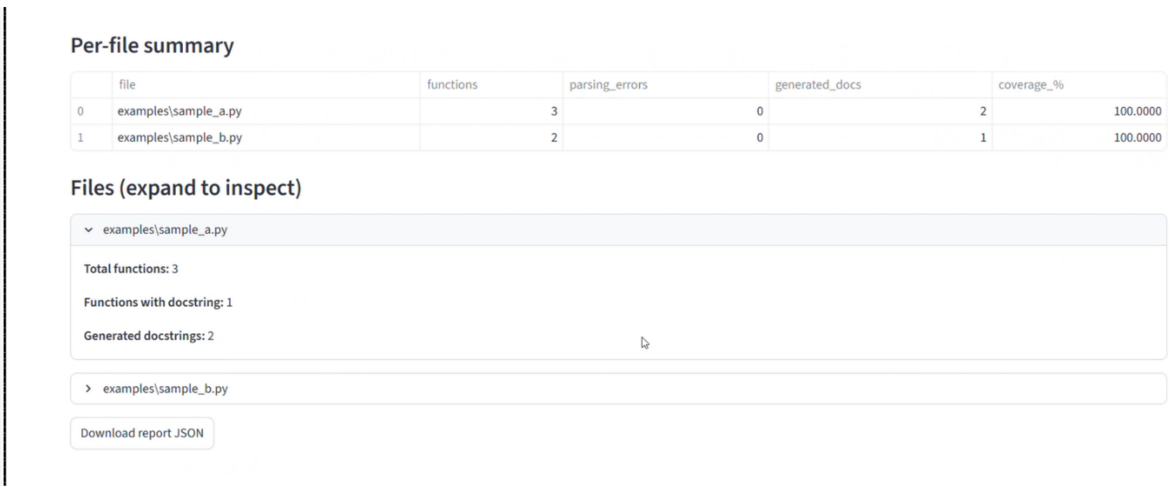
AST Parsing Output

```
File: sample_a.py | Functions: 2 | Classes: 1

def calculate_average(numbers): has_docstring=False # line 4
def add(a, b): has_docstring=True # line 13 class Processor:
# line 18 def process(self, data): has_docstring=False #
line 19
```

Generated Docstrings & Coverage Report

Preview: `calculate_average` in `C:/Users/DHIKSHITA/Downloads/AI-Powered Code Reviewer and Quality/examples/sample_a.py` (line 4)

```
"""TODO: Add summary for `calculate_average`.

    Args:
        numbers (Any): TODO: describe `numbers`.

    Returns:
        Any: TODO: describe return value.
    """
```



Per-file summary

	file	functions	parsing_errors	generated_docs	coverage_%
0	examples\sample_a.py	3	0	2	100.0000
1	examples\sample_b.py	2	0	1	100.0000

Files (expand to inspect)

examples\sample_a.py

Total functions: 3

Functions with docstring: 1

Generated docstrings: 2

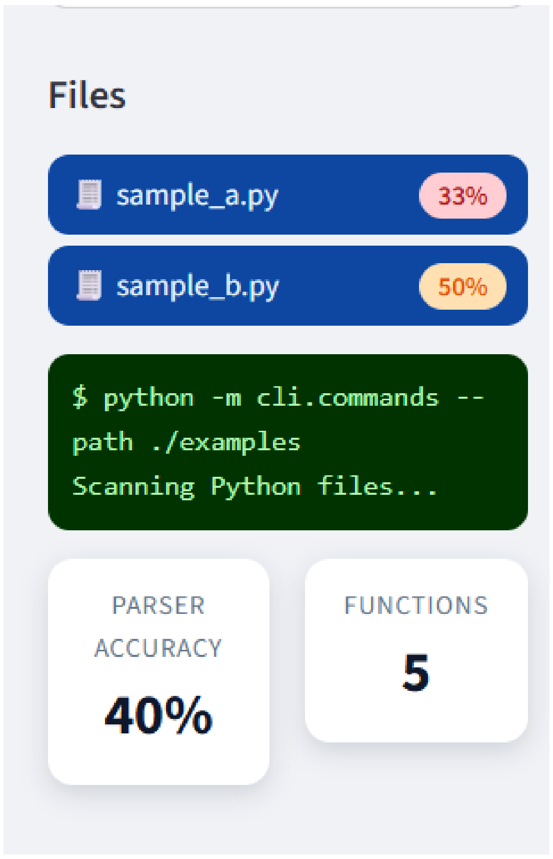
examples\sample_b.py

Per-file Coverage

File	Total	Documented	Percent %
sample_a.py	3	1	33.3
sample_b.py	2	1	50.0

Overall Coverage

40.0%



Files

sample_a.py 33%

sample_b.py 50%

```
$ python -m cli.commands --
path ./examples
Scanning Python files...
```

PARSER ACCURACY

40%

FUNCTIONS

5

5. MILESTONE 2: Interactive Review Interface and Quality Evaluation

5.1 Milestone Description

Milestone 2 focuses on transforming the core code analysis engine developed in Milestone 1 into an **interactive, user-friendly application** using **Streamlit**. While the previous milestone established static analysis and heuristic-based review capabilities via a CLI, this phase emphasizes **visualization, usability, and real-time interaction**.

The primary objective of this milestone is to allow users to **scan Python projects, inspect review findings, preview docstring improvements, validate code quality, and analyze maintainability metrics** through a web-based dashboard. This milestone bridges the gap between backend analysis and practical developer usage, enabling safer review workflows without directly modifying source files.

The system now operates as a **fully functional prototype**, capable of end-to-end code review through a graphical interface, making it suitable for demonstrations, mentor evaluations, and further AI enhancements in future milestones.

5.2 Functional Components Implemented in Milestone 2

1. Streamlit-Based User Interface Module

- Developed using the **Streamlit framework** for rapid UI development.
- Provides a clean and intuitive dashboard layout with sidebar-based navigation.
- Supports multiple views:
  - Home
  - Scan
  - Docstrings

    - Validation
    - Metrics
- Allows users to specify and load a Python project folder dynamically.

Purpose:
To enable non-CLI users to interact with the code reviewer easily and visually.

2. Project Scanning & File Management Module

- Automatically scans all .py files within the selected project directory.
- Displays total number of Python files detected.
- Identifies files with missing or incomplete docstrings.
- Provides instant feedback upon scan completion.

Key Features:

- Folder-based recursive scanning
- Safe read-only analysis
- Summary metrics displayed using Streamlit indicators

3. Docstring Review & Diff Preview Module

- Displays source code **before** and **after** docstring generation.
- Generates preview-only docstring suggestions without modifying files.
- Uses difflib to present a **unified diff view** highlighting changes.
- Ensures transparency and safety in code review.

Advantages:

- Helps developers understand suggested improvements
- Prevents accidental overwrites
- Encourages manual review and learning

4. Validation & Rule-Based Feedback Module

- Validates Python files against predefined quality and documentation rules.
- Categorizes feedback into:
    - Success
    - Warning
    - Error
- Displays validation results in a readable and color-coded format.

Checks Include:

- Docstring presence
- Structural correctness
- Basic style and quality rules

5. Code Quality Metrics & Maintainability Analysis

- Computes **Cyclomatic Complexity** for logical assessment.
- Calculates **Maintainability Index** for long-term sustainability evaluation.
- Displays metrics using numeric indicators and explanatory notes.

Interpretation Support:

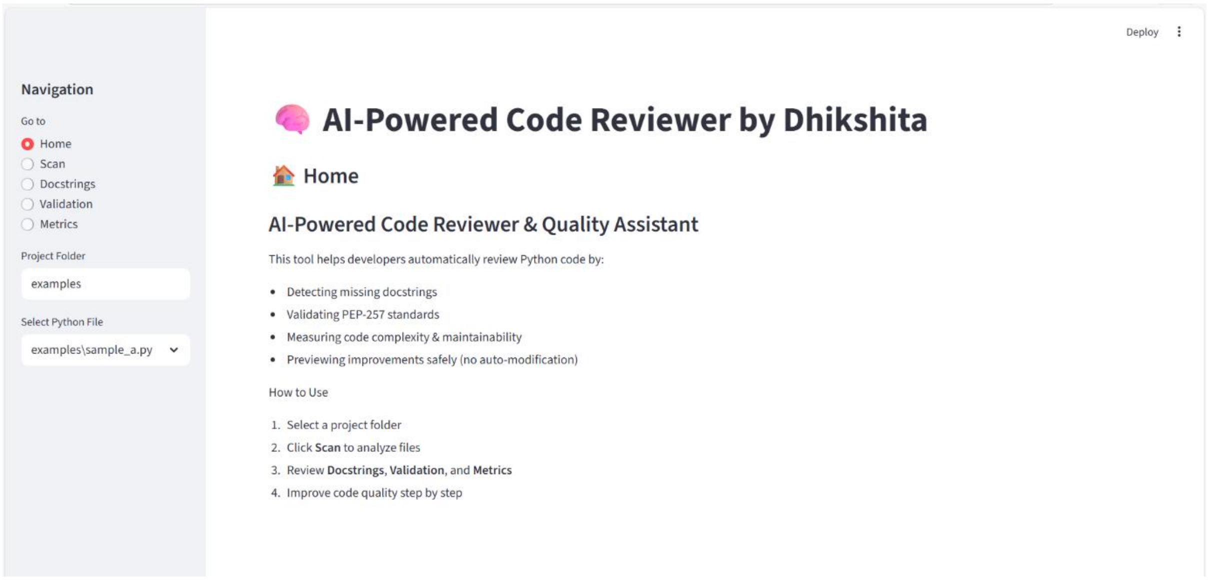
- Cyclomatic Complexity $> 10 \rightarrow$ Indicates complex logic
- Maintainability Index $< 65 \rightarrow$ Indicates difficult maintenance

This module helps developers make informed decisions about refactoring and optimization.

5.3 OUTCOMES OF MILESTONE 2

- Fully functional **Streamlit-based interactive dashboard**
- Seamless integration of review engine, validator, and metrics modules
- Safe preview of AI-assisted docstring improvements
- Improved usability compared to CLI-only interaction
- Strong foundation for AI-driven enhancements and CI/CD integration in future milestones

OUTPUTS:



AI-Powered Code Reviewer by Dhikshita

Home

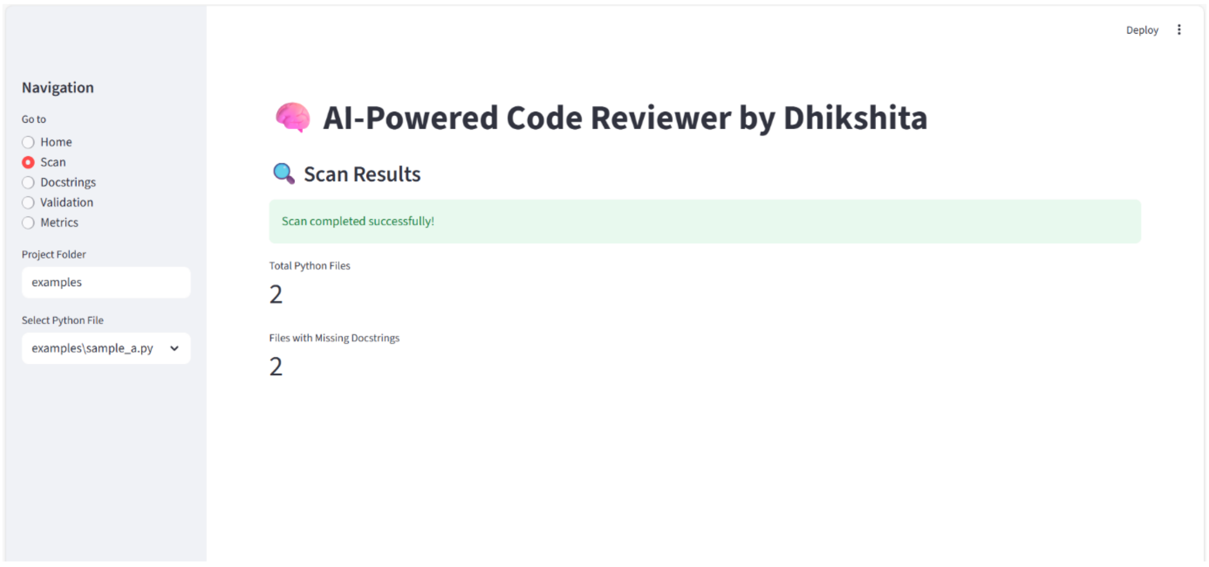
AI-Powered Code Reviewer & Quality Assistant

This tool helps developers automatically review Python code by:

- Detecting missing docstrings
- Validating PEP-257 standards
- Measuring code complexity & maintainability
- Previewing improvements safely (no auto-modification)

How to Use

1. Select a project folder
2. Click **Scan** to analyze files
3. Review **Docstrings**, **Validation**, and **Metrics**
4. Improve code quality step by step



Deploy

Navigation

Go to

- Home
- [✓ Scan]
- Docstrings
- Validation
- Metrics

Project Folder

examples

Select Python File

examples\sample_a.py

AI-Powered Code Reviewer by Dhikshita

Scan Results

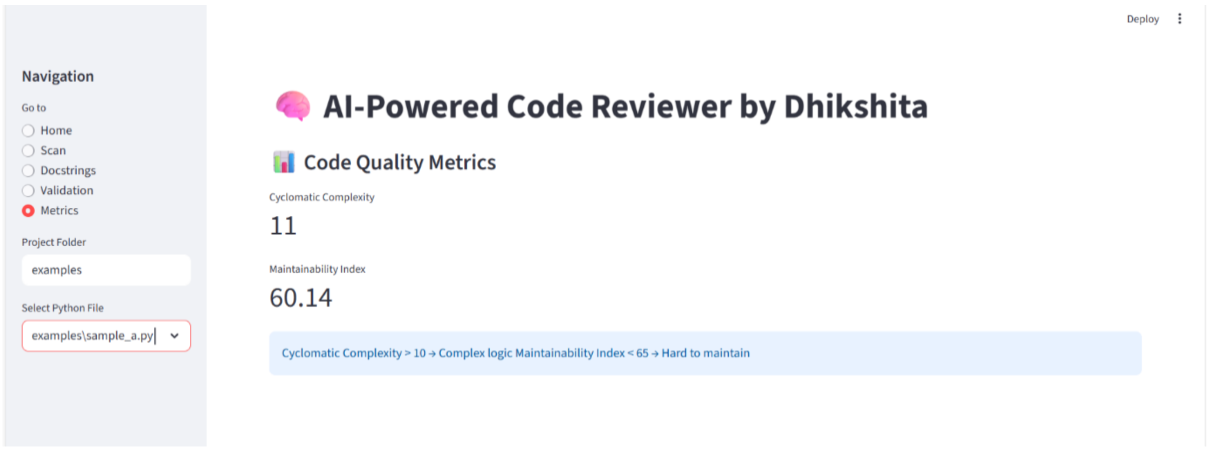
Scan completed successfully!

Total Python Files

2

Files with Missing Docstrings

2



Deploy

Navigation

Go to

- Home
- Scan
- Docstrings
- Validation
- [✓ Metrics]

Project Folder

examples

Select Python File

examples\sample_a.py

AI-Powered Code Reviewer by Dhikshita

Code Quality Metrics

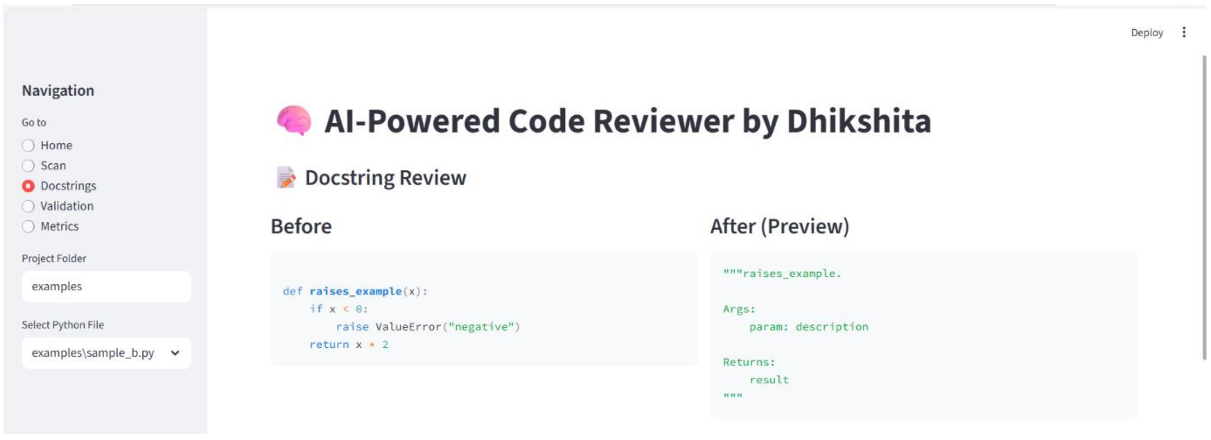
Cyclomatic Complexity

11

Maintainability Index

60.14

Cyclomatic Complexity > 10 → Complex logic Maintainability Index < 65 → Hard to maintain



Deploy

Navigation

Go to

- Home
- Scan
- [✓ Docstrings]
- Validation
- Metrics

Project Folder

examples

Select Python File

examples\sample_b.py

AI-Powered Code Reviewer by Dhikshita

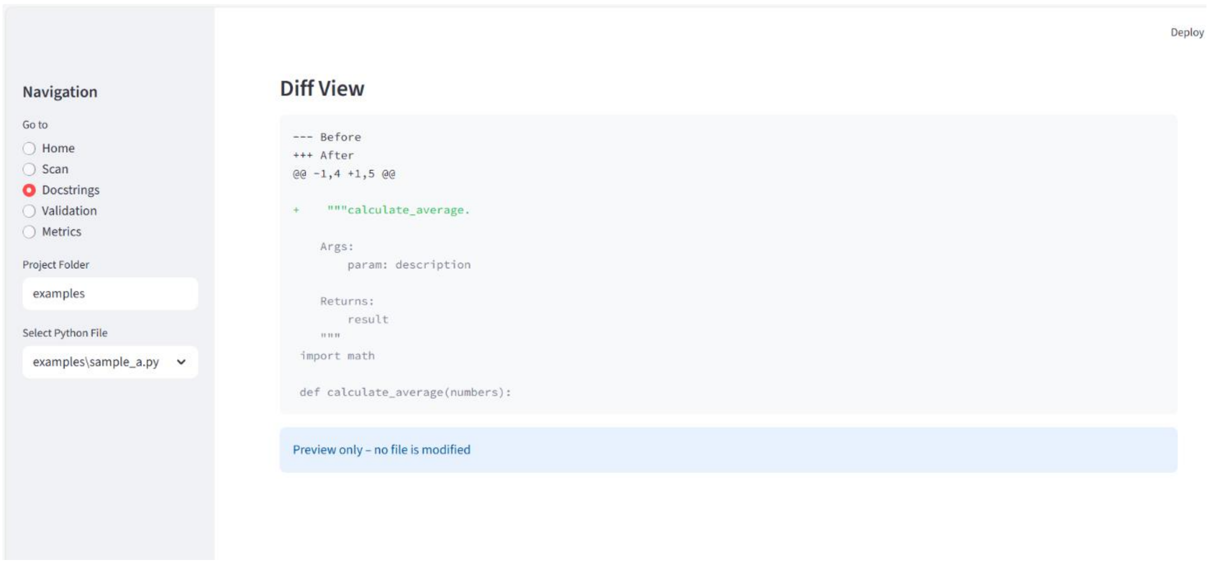
Docstring Review

Before

```
def raises_example(x):
    if x < 0:
        raise ValueError("negative")
    return x * 2
```

After (Preview)

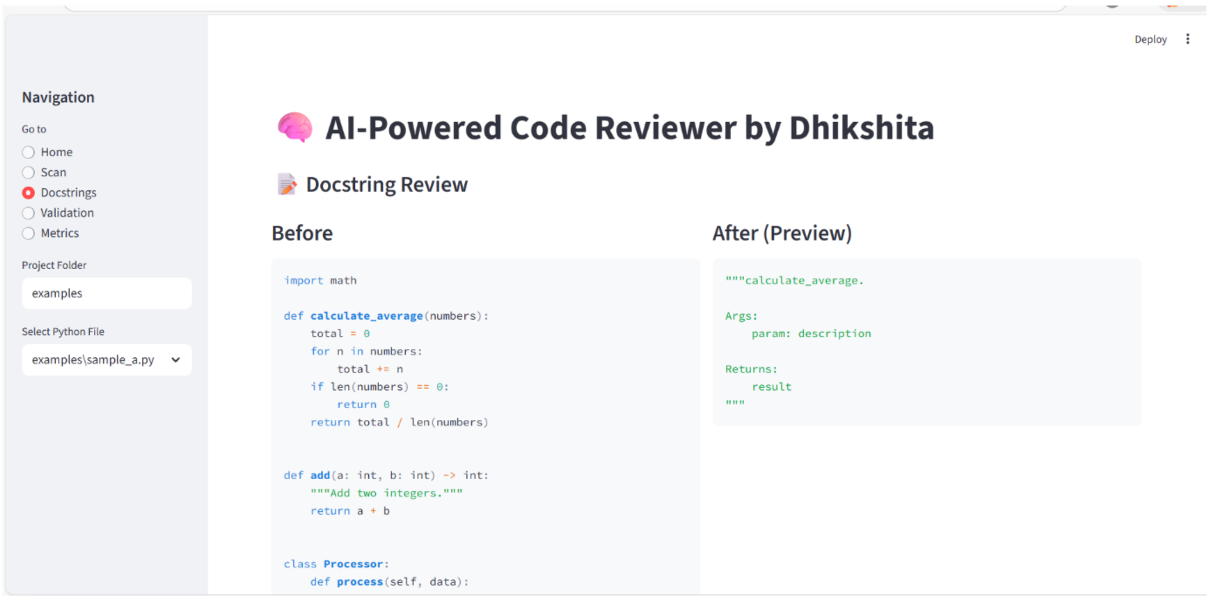
```
"""raises_example.

Args:
    param: description

Returns:
    result
"""
```



Diff View

```
--- Before
+++ After
@@ -1,4 +1,5 @@

+    """calculate_average.

    Args:
        param: description

    Returns:
        result
    """
 import math

 def calculate_average(numbers):
```

Preview only – no file is modified



AI-Powered Code Reviewer by Dhikshita

Docstring Review

Before

```
import math

def calculate_average(numbers):
    total = 0
    for n in numbers:
        total += n
    if len(numbers) == 0:
        return 0
    return total / len(numbers)


def add(a: int, b: int) -> int:
    """Add two integers."""
    return a + b


class Processor:
    def process(self, data):
```

After (Preview)

```
"""calculate_average.

Args:
    param: description

Returns:
    result
"""
```

6. Milestone 3: LLM-Driven Docstring Generation & Review

6.1 Milestone Description

The AI-Powered Code Reviewer is a static analysis and documentation system designed to improve Python code quality.
It analyzes source code using **AST (Abstract Syntax Tree)**, detects missing or incomplete docstrings, and assists developers by **intelligently generating professional docstrings using an LLM**, while maintaining full user control.

Milestone 3 focuses on **AI-based docstring generation**, preview, and safe application into source code.

6.2. Purpose of Milestone 3

The goal of Milestone 3 is to:

- Automate **semantic docstring generation** using an LLM
- Support **multiple docstring styles**
- Ensure **no automatic code modification**
- Provide a **human-in-the-loop review workflow**
- Maintain **AST safety and reversibility**

This milestone bridges **static analysis** with **generative AI** in a controlled manner.

↓

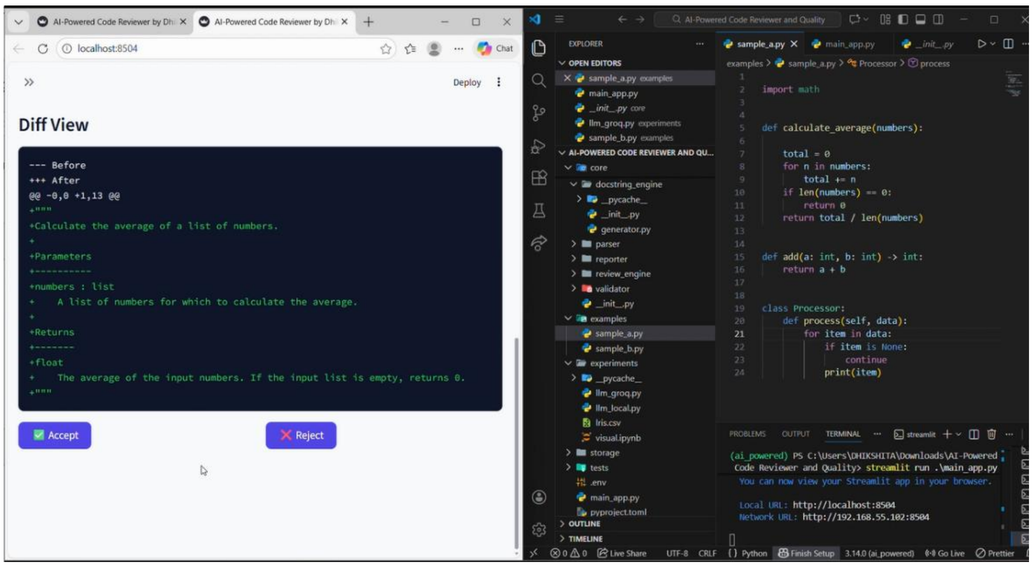
6.3 Why LLM Integration is Required?

Traditional rule-based docstring generators:

- Produce static and repetitive templates
- Cannot understand function intent
- Fail to describe real logic and behavior

Using an LLM enables:

- Semantic understanding of function purpose
- Human-like documentation quality
- Reduced manual documentation effort

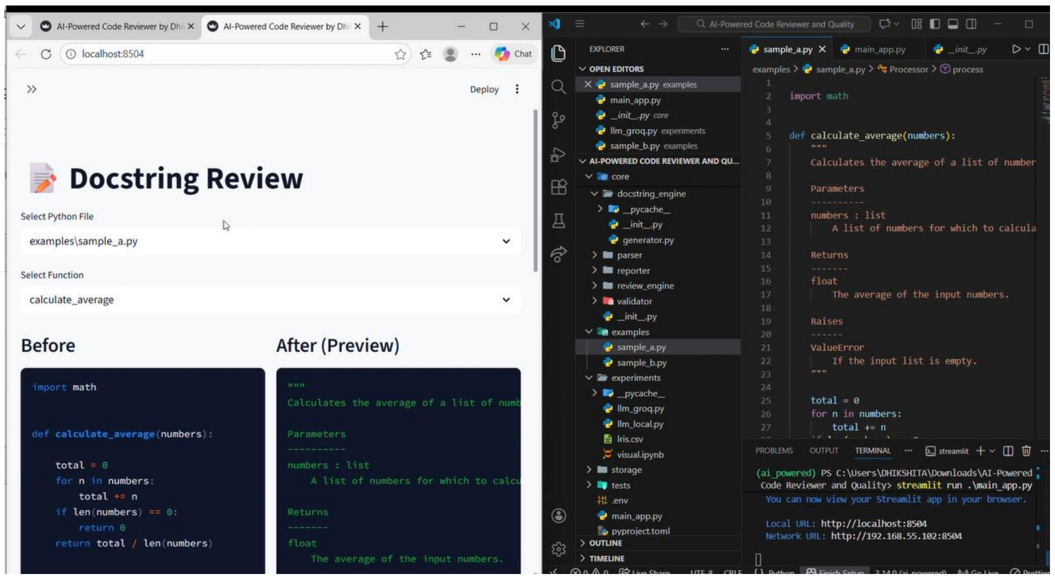

AI-Powered Code Reviewer by Dh
localhost:8504
Deploy
Chat
Accept
Reject
AI-Powered Code Reviewer and Quality
sample_a.py
main_app.py
init.py
examples > sample_a.py > Processor > process
PROBLEMS
OUTPUT
TERMINAL
streamlit

Diff View

```
--- Before
+++ After
@@ -0,0 +1,13 @@
+"""
+Calculate the average of a list of numbers.
+
+Parameters
+----------
+numbers : list
+    A list of numbers for which to calculate the average.
+
+Returns
+-------
+float
+    The average of the input numbers. If the input list is empty, returns 0.
+"""
```

```python

import math


def calculate_average(numbers):

    total = 0
    for n in numbers:
        total += n
    if len(numbers) == 0:
        return 0
    return total / len(numbers)


def add(a: int, b: int) -> int:
    return a + b


class Processor:
    def process(self, data):
        for item in data:
            if item is None:
                continue
            print(item)
```

```
(ai_powered) PS C:\Users\DHIKSHITA\Downloads\AI-Powered
 Code Reviewer and Quality> streamlit run .\main_app.py
  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8504
  Network URL: http://192.168.55.102:8504
```


AI-Powered Code Reviewer by Dh
localhost:8504
Deploy
Chat
AI-Powered Code Reviewer and Quality
sample_a.py
main_app.py
init.py
examples > sample_a.py > Processor > process
PROBLEMS
OUTPUT
TERMINAL
streamlit

Docstring Review

Select Python File: examples\sample_a.py

Select Function: calculate_average

Before

```python
import math


def calculate_average(numbers):

    total = 0
    for n in numbers:
        total += n
    if len(numbers) == 0:
        return 0
    return total / len(numbers)
```

After (Preview)

```
"""
Calculates the average of a list of numb

Parameters
----------
numbers : list
    A list of numbers for which to calcu

Returns
-------
float
    The average of the input numbers.
```

```python

import math


def calculate_average(numbers):
    """
    Calculates the average of a list of number

    Parameters
    ----------
    numbers : list
        A list of numbers for which to calcula

    Returns
    -------
    float
        The average of the input numbers.

    Raises
    ------
    ValueError
        If the input list is empty.
    """

    total = 0
    for n in numbers:
        total += n
```

```
(ai_powered) PS C:\Users\DHIKSHITA\Downloads\AI-Powered
 Code Reviewer and Quality> streamlit run .\main_app.py
  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8504
  Network URL: http://192.168.55.102:8504
```

MILESTONE 4: AI INTEGRATION, USER INTERFACE & FINAL DEPLOYMENT

5. MILESTONE 4: AI-Driven Review, UI Enhancement & System Integration

5.1 Milestone Description

Milestone 4 represents the **final and most critical phase** of the AI-Powered Code Reviewer project. This phase focuses on integrating **Large Language Models (LLMs)** to generate intelligent, human-like code review feedback, enhancing the **Streamlit-based user interface**, and ensuring **end-to-end system integration**.

The goal of this milestone is to transform the previously rule-based static analyzer into a **fully functional AI-assisted software quality tool** capable of:

- Understanding code semantics
- Providing contextual suggestions
- Offering best-practice recommendations
- Delivering an intuitive and user-friendly experience

This milestone bridges the gap between traditional static analysis and modern AI-assisted developer tools.

5.2 Objectives of Milestone 4

The key objectives achieved in this milestone are:

- Integrate **LLM-based AI review engine** (Groq / LLaMA / LangChain)
- Generate **context-aware review comments**
- Enhance **UI/UX using Streamlit**
- Display review results with **severity, metrics, and explanations**

- Enable **project-level scanning**
- Ensure **system stability and portability**

5.3 Functional Components Implemented

1. AI Review Engine Integration

- Integrated LLMs using **LangChain** framework.
- Code content is converted into structured prompts.
- AI generates:
  - Code quality feedback
  - Optimization suggestions
  - Readability improvements
  - Best practice recommendations
- Supports multi-file project analysis.

Technologies Used:

- LangChain
- Groq / LLaMA
- Prompt Engineering
- Project folder scanner
- Review result cards
- Severity-based color indicators
- Expandable feedback sections
- Sidebar navigation

UI Improvements:

- Dark/Light theme compatibility
- High-contrast button colors

5.4 Tools & Technologies Used

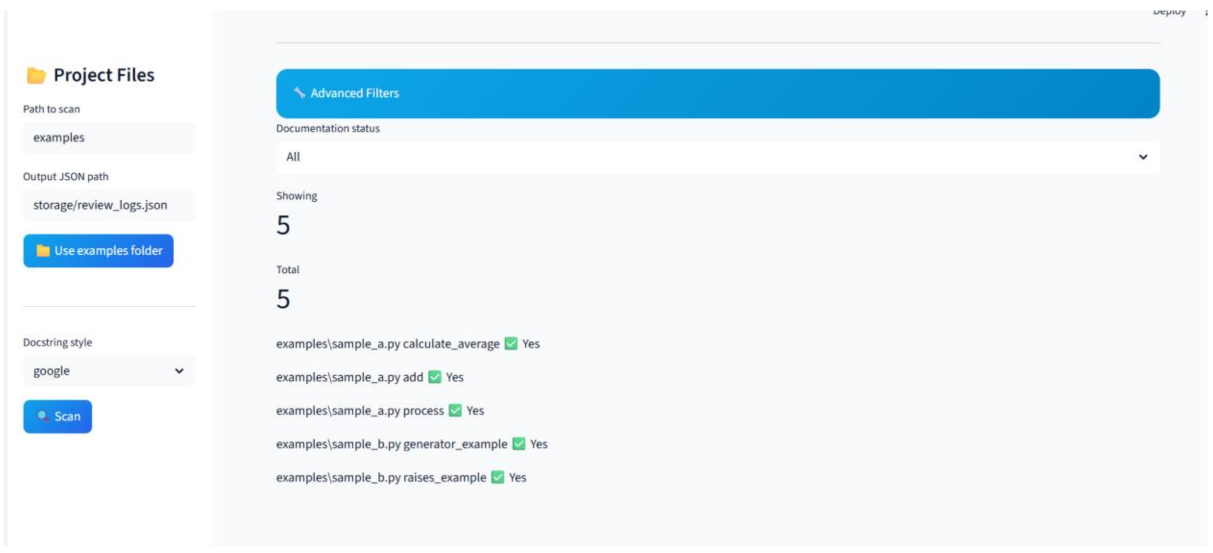
Category	**Tools**
Programming Language	Python
UI Framework	Streamlit
AI Framework	LangChain
LLM Provider	Groq / LLaMA
Static Analysis	AST, Radon, Pydocstyle
Testing	Pytest
Environment	Python Virtual Environment

5.5 Key Outcomes of Milestone 4

- Successfully integrated AI-driven code review
- Enhanced developer experience with intuitive UI
- Achieved real-time feedback generation
- Improved code understanding beyond rule-based checks
- Delivered a **production-ready student project**

5.6 Conclusion

Milestone 4 successfully completes the AI-Powered Code Reviewer by integrating artificial intelligence with static code analysis and a user-friendly interface. The project now stands as a **complete, deployable, and industry-aligned solution** that demonstrates both technical depth and practical relevance.



Project Files

Path to scan: examples

Output JSON path: storage/review_logs.json

Docstring style: google

Advanced Filters

Documentation status: All

Showing
5

Total
5

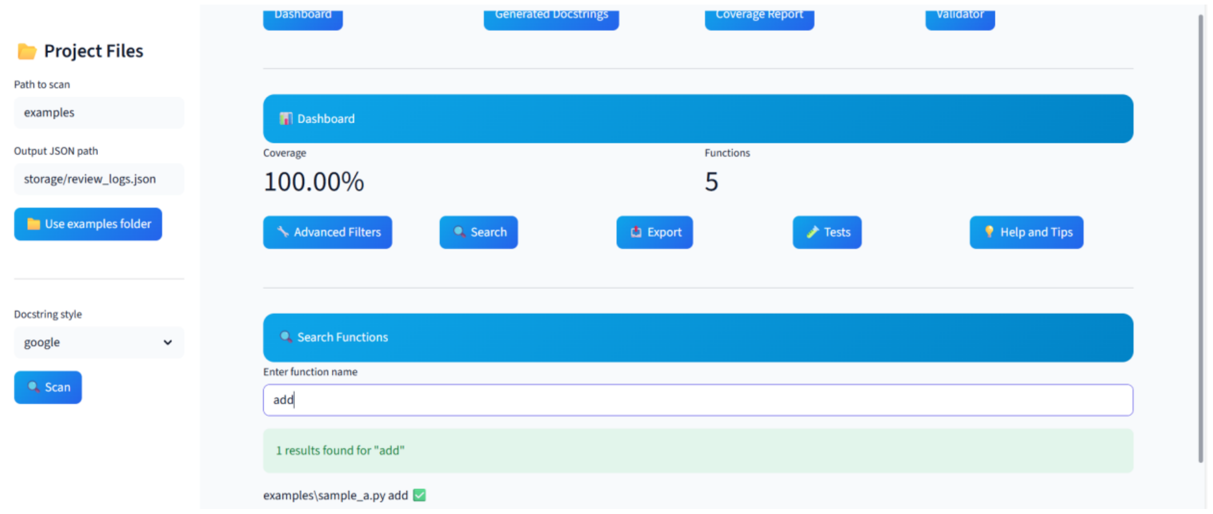
examples\sample_a.py calculate_average ✅ Yes

examples\sample_a.py add ✅ Yes

examples\sample_a.py process ✅ Yes

examples\sample_b.py generator_example ✅ Yes

examples\sample_b.py raises_example ✅ Yes



Project Files

Path to scan: examples

Output JSON path: storage/review_logs.json

Docstring style: google

Dashboard

Coverage
100.00%

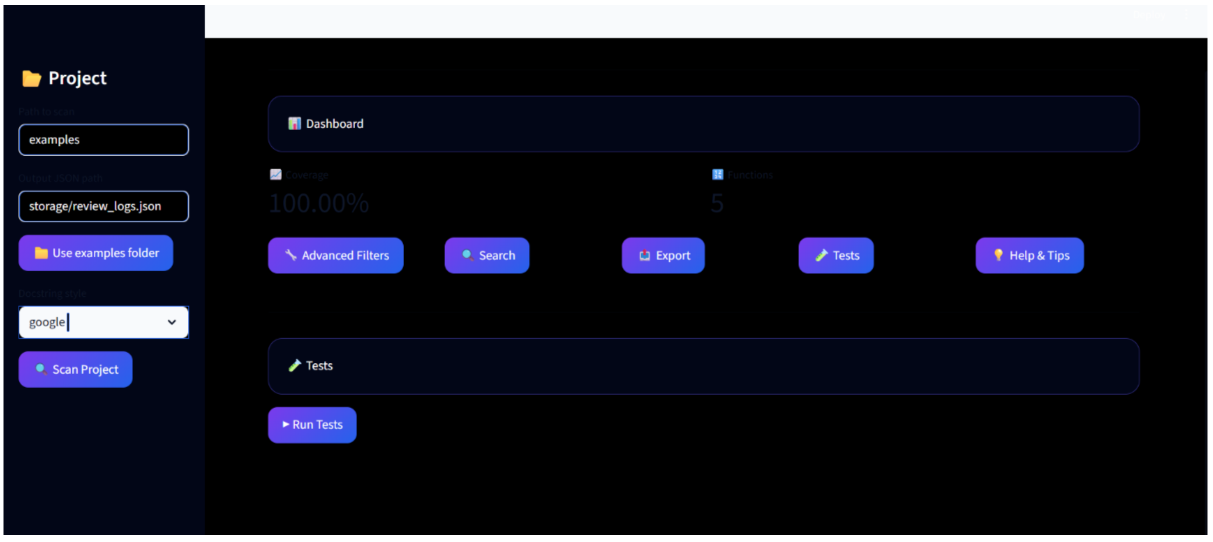
Functions
5

Search Functions

Enter function name: add

1 results found for "add"

examples\sample_a.py add ✅



Project

Path to scan: examples

Output JSON path: storage/review_logs.json

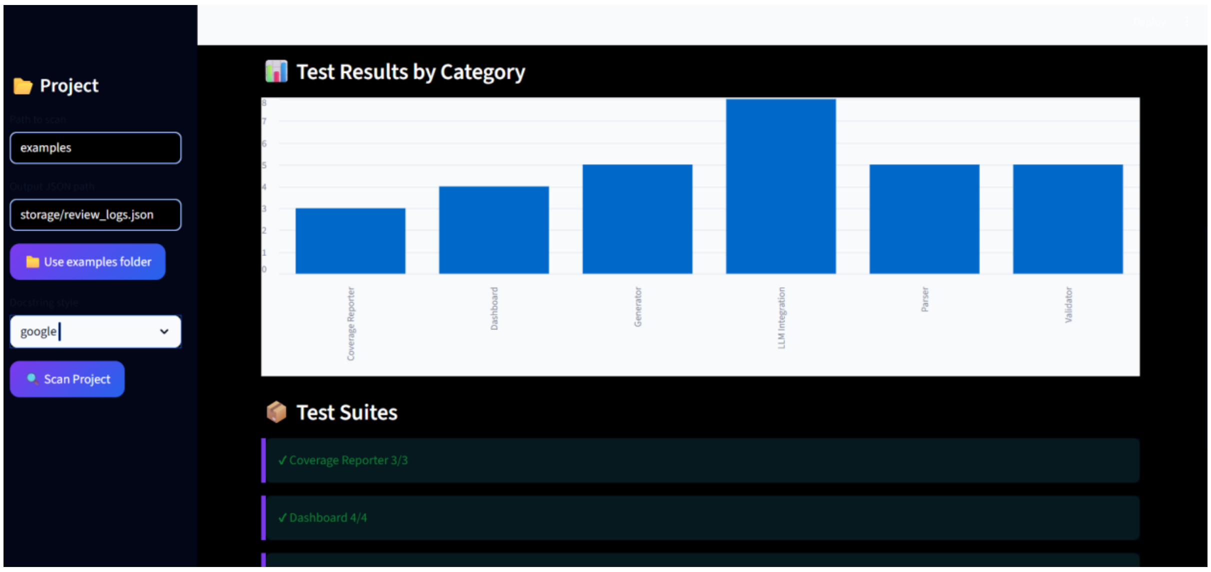
Docstring style: google

Dashboard

Coverage
100.00%

Functions
5

Tests



Project

Path to scan: examples

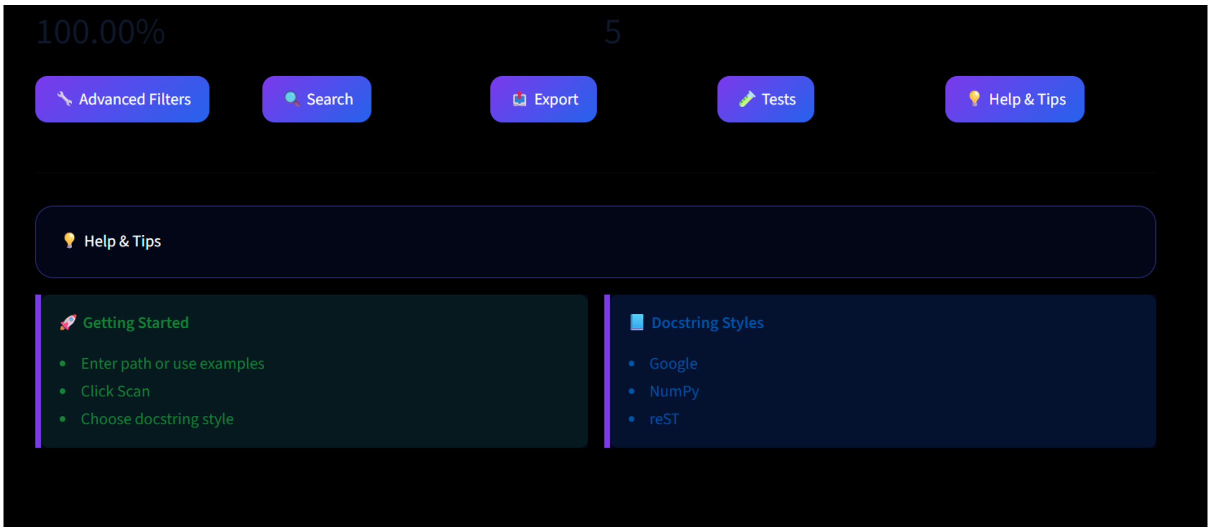
Output JSON path: storage/review_logs.json

Docstring style: google

Test Suites

✓ Coverage Reporter 3/3

✓ Dashboard 4/4



100.00%

5

Help & Tips

Getting Started

- Enter path or use examples
- Click Scan
- Choose docstring style

Docstring Styles

- Google
- NumPy
- reST