

CodeGuard – AI-Powered Code Reviewer and Quality Assistant

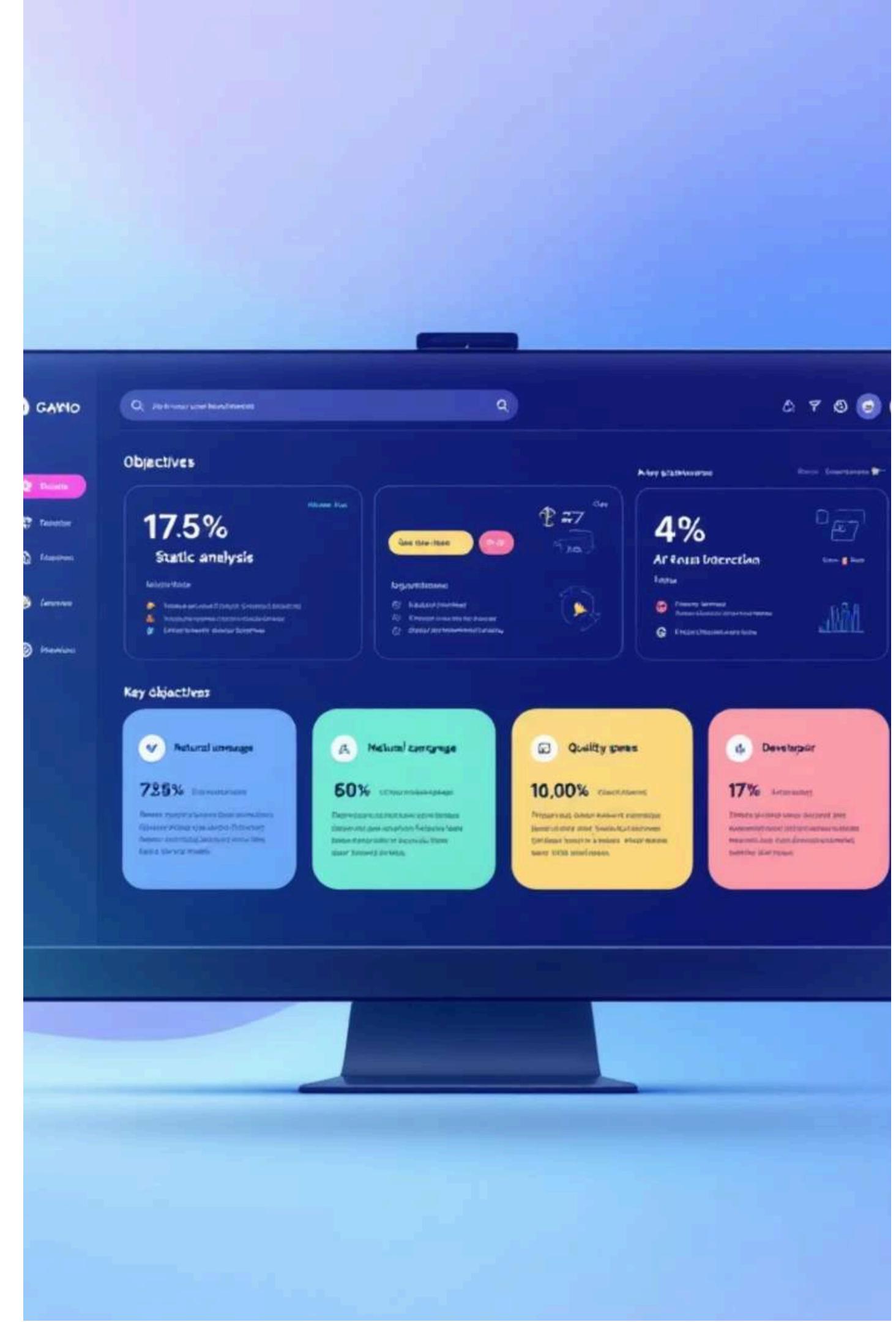


PROBLEM STATEMENT

Code reviews are essential for maintaining software quality, but traditional manual reviews are often time-consuming, inconsistent, and prone to human oversight. Developers struggle to keep documentation, readability, and coding standards intact across large projects, while existing static analysis tools provide only technical checks without contextual, human-like feedback. This leads to reduced maintainability, slower onboarding, and quality drift over time. To address these challenges, there is a need for an AI-assisted solution that can automatically detect issues, explain them in natural language, and suggest actionable fixes, thereby improving efficiency, consistency, and developer understanding.

OBJECTIVES

- 1 Automate code review using static analysis and AST parsing
- 2 Leverage Ollama Phi-3 model to provide natural-language feedback and auto-fixes
- 3 Ensure quality gates with Git pre-commit hooks and CI/CD integration
- 4 Provide developer interaction via CLI commands and Streamlit dashboard



PROJECT OVERVIEW



Deliverables

AI-assisted code review tool for Python projects combining static analysis and Ollama Phi-3 feedback to deliver actionable suggestions.



Developer interfaces

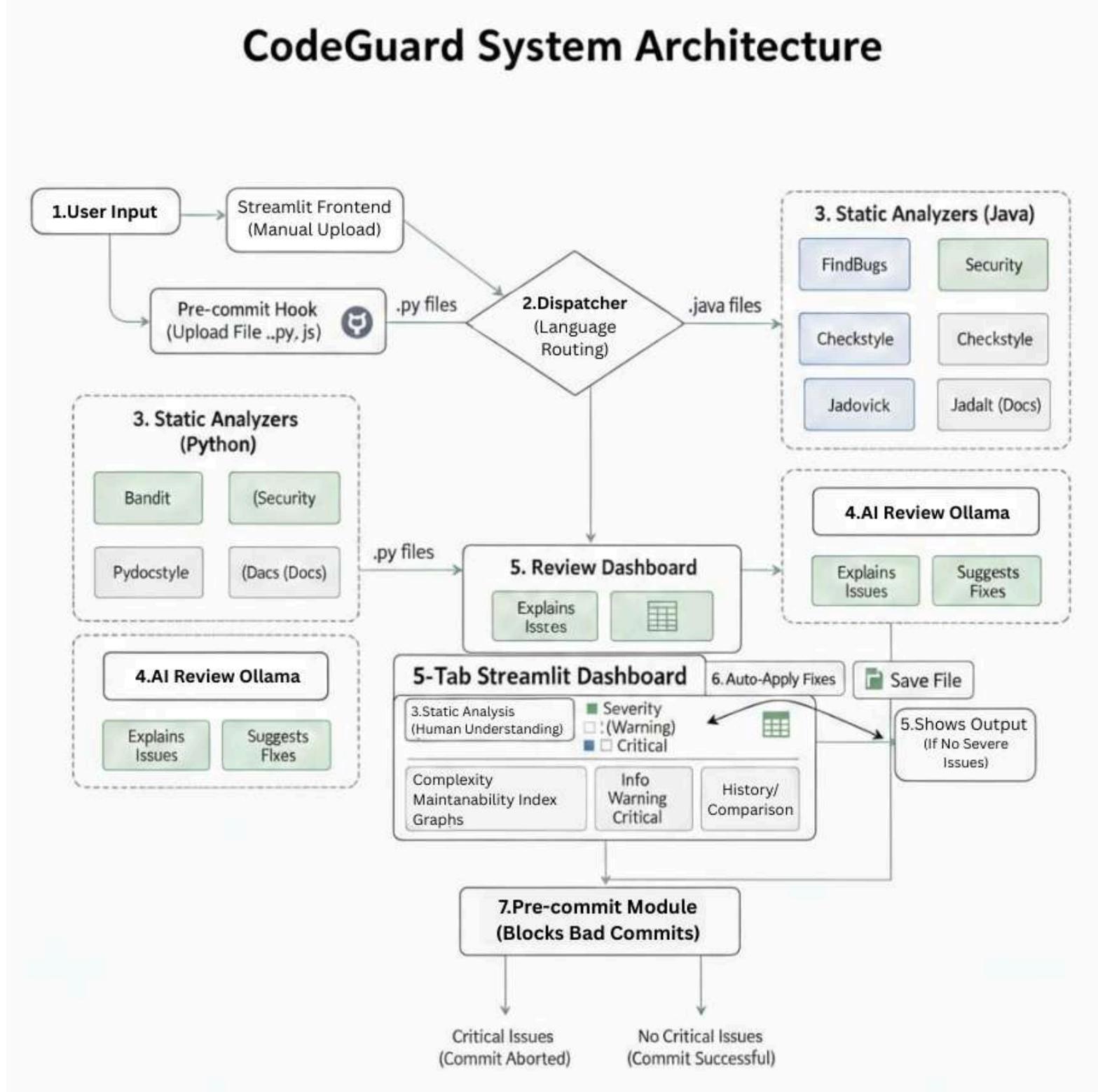
CLI commands for scanning, reviewing, applying fixes, reporting, plus a Streamlit dashboard for interactive visualization.



Configurability

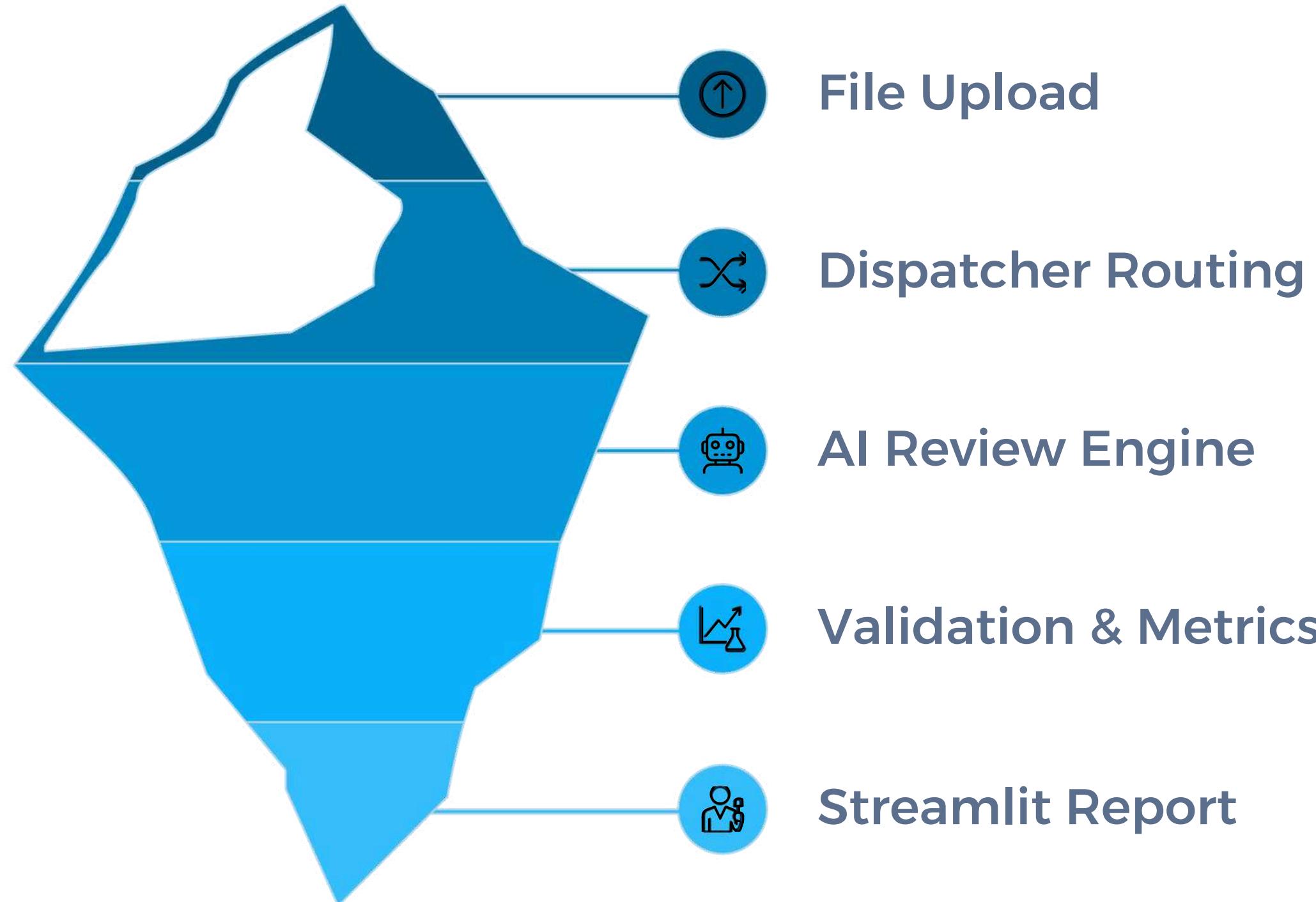
Seamless integration with Git workflows (pre-commit hooks, CI/CD pipelines), configurable rules (PEP8, custom checks, severity thresholds), and outputs including docstring coverage, maintainability index, and complexity metrics..

SYSTEM ARCHITECTURE



- Input Layer:** Code files uploaded via Streamlit frontend or Git pre-commit hooks.
- Processing Layer:** Dispatcher routes files to language-specific static analysers (Python: Bandit, Pydocstyle; Java: FindBugs, Checkstyle, etc.) and AI review engine for explanations and fix suggestions.
- Output Layer:** Streamlit dashboard visualizes issues (severity, complexity, maintainability), supports interactive review, auto-apply fixes, and enforces quality gates through pre-commit/CI pipelines.

WORKFLOW



Each run: parse AST, run deterministic checks, enrich findings with LLM explanations, propose patch, run safety validation and present an actionable report to the developer.

AST Parser

Module 1: Code Parsing & Analysis

AST Parser



```
f Functions: function();
1 Classes;
2 Chatterbacks;
3 Functions;
18 Imports;
46 Imports;
16 Long Functions;
16. Missing Type Hints();
```

Static Analysis

Code Smell

Code Smells

Missing Function

Static Analysis

{-

Extractive: Constitarted: >

<ix:

```
Functions functions;
Classes, Missing Functions;
Cart. Long Functions
Missin type Hints;
```

>

)

MODULE 1 : CODE PARSING & ANALYSIS

AST-driven analysis for Python and pluggable detectors for other languages.

Detects complexity, style, security, and documentation issues.

- High cyclomatic complexity
- Long functions, missing docstrings & type hints
- Naming violations, hardcoded secrets
- Unsafe eval/exec patterns

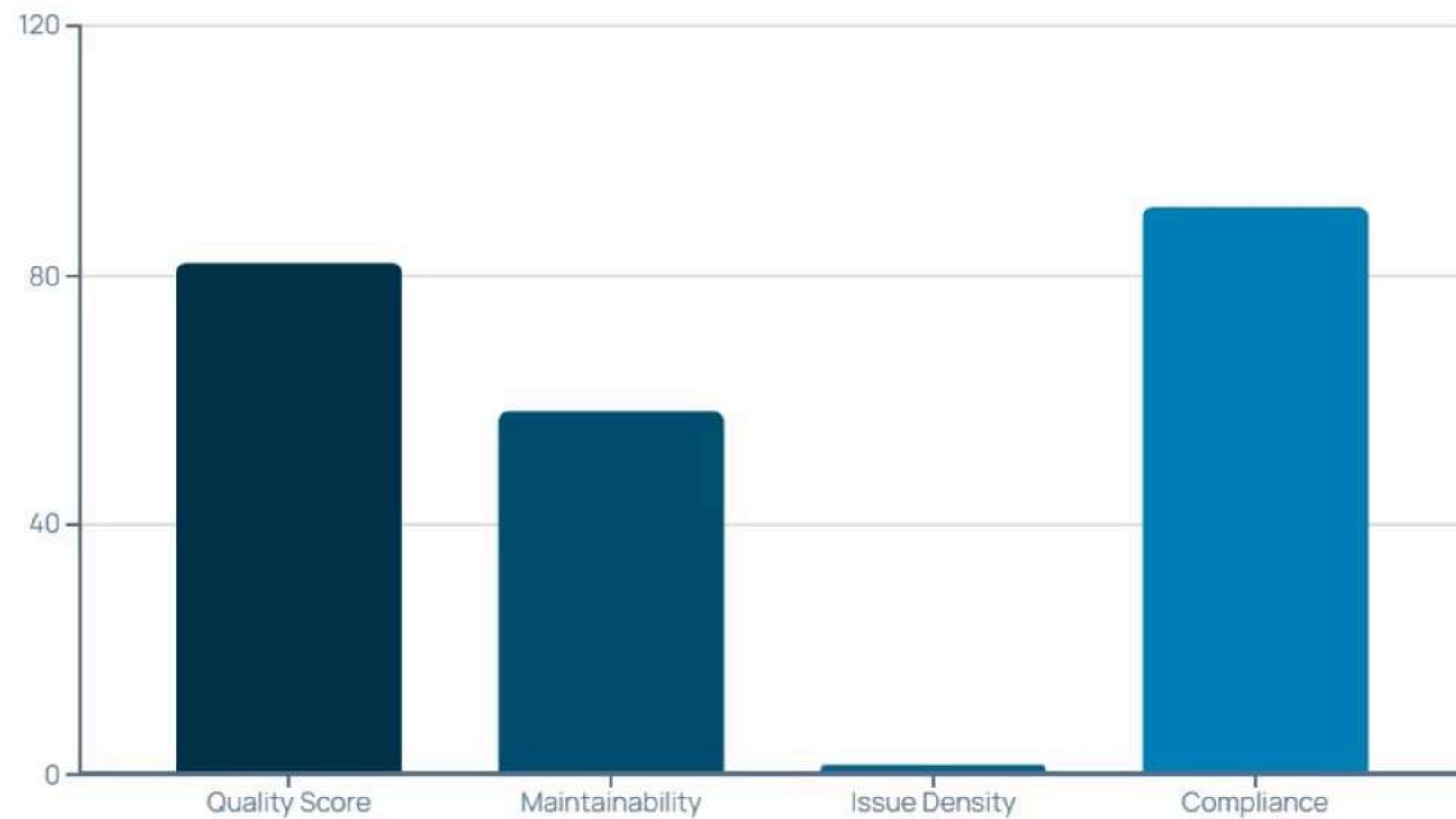
Dispatcher enables language-specific analyzers and unified issue format.

MODULE 2 : AI REVIEW ENGINE

- Uses Ollama Phi-3 model with prompt templates to generate human-like code reviews.
- Classifies findings by severity: Info, Warning, Critical.
- Explains issues in natural language for easy understanding.
- Automatically corrects simple issues like naming, docstrings, and spacing.
- Compatible with CLI, Streamlit UI, and Git workflow.



MODULE 3 : VALIDATION & METRICS



- Tracks key indicators: Quality Score, Maintainability Index, Issue Density, Severity breakdown.
- Enforces quality gates with thresholds (Score ≥ 70 , MI ≥ 50).
- Identifies best and worst files for targeted improvements.
- Uses severity-weighted scoring to prioritize issues.
- Generates exportable reports (CSV/HTML; planned JSON/ZIP) for easy sharing.

MODULE 4 – CLI & CONFIGURATION

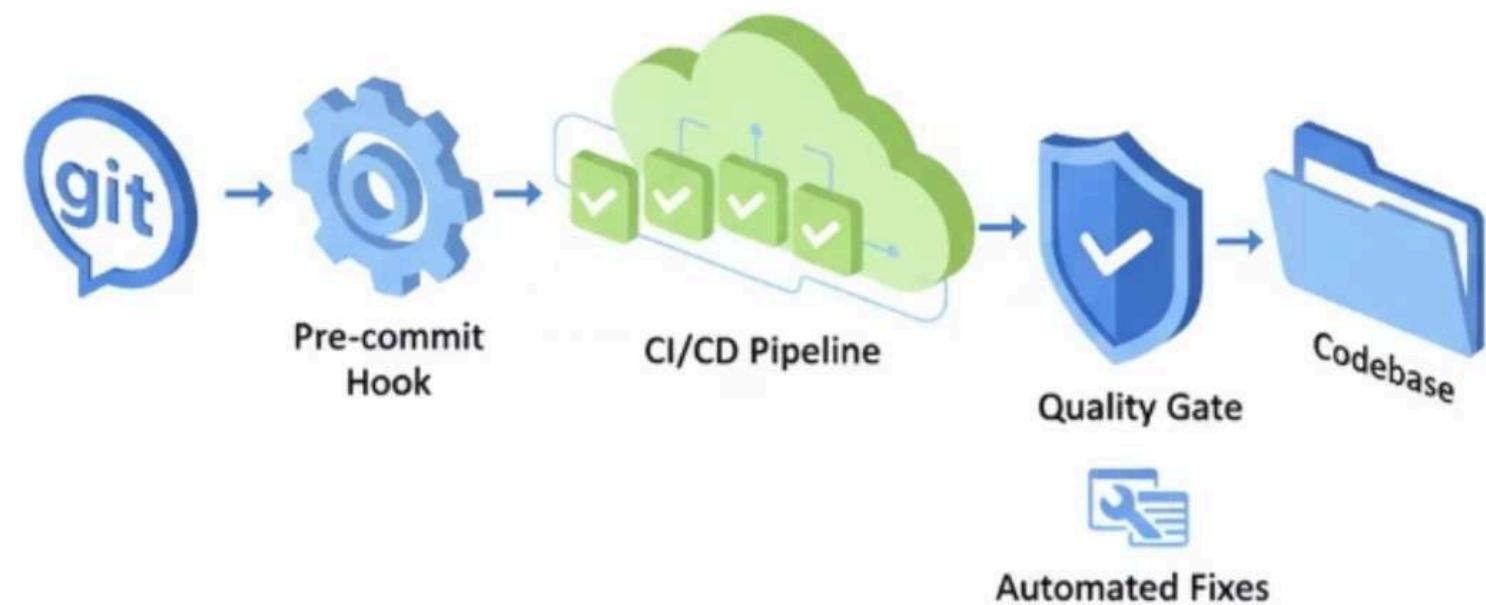
Command Set

- `scan` — initiates static analysis of code.
- `review` — triggers AI-powered explanations and feedback.
- `apply` — automatically applies suggested code fixes.
- `report` — generates comprehensive review artifacts.
- `diff` — displays proposed code changes in detail.

Capabilities

- Configurable rules via `pyproject.toml` (PEP8, custom checks, severity thresholds, excluded paths)
- Flexible severity control: block commits on critical issues, warn on lower-priority findings
- Supports exclusions for specific files or directories
- Seamless Git integration for smooth developer workflows

Module 5 : VCS & CI Integration



- Pre-commit hook reviews staged files before commit; CI/CD templates enforce quality gates in pipelines
- Fast execution (<5s on staged changes) ensures smooth developer workflow without friction
- Coverage thresholds (e.g., $\geq 90\%$) block builds if standards are not met
- Seamless integration with GitHub/GitLab workflows for consistent code quality.

MODULE 6 : STREAMLIT WEB UI



- Interactive dashboard to visualize issues and fixes
- Side-by-side diff view with AI suggestions for easy comparison
- Preview, accept, or reject fixes directly in the interface
- Enhanced usability with filters, search, and tooltips (planned features)
- Fast performance (<5s review speed on staged changes)

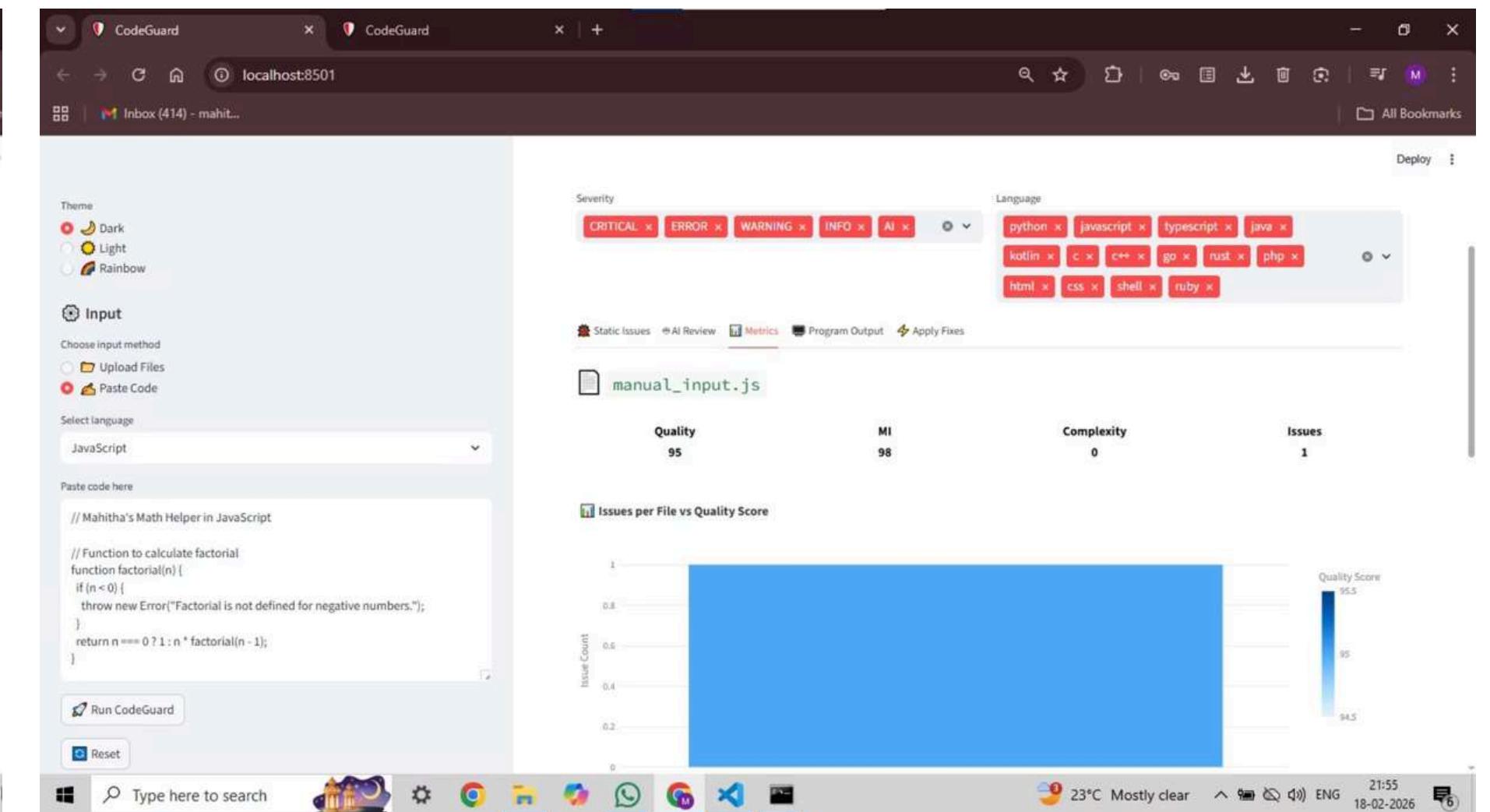
PERFORMANCE & RESULTS

The screenshot shows the CodeGuard web application interface. At the top, there are two browser tabs both titled "CodeGuard" and "localhost:8502". The left tab displays a green banner stating "Congratulations! No issues found." and a message "Your code is clean and beautiful!". Below this are sections for "Filters" (Severity: CRITICAL, ERROR, WARNING, INFO, AI) and "Language" (Python, JavaScript, TypeScript, Java, Kotlin, C, C++, Go, Rust, PHP, HTML, CSS, Shell, Ruby). A code editor window shows "manual_input.py" with the following code:

```
// Mahitha's Math Helper in Python
def add(a, b):
    return a + b
print(add(2, 3))
```

Below the code editor is a "Program Output" section showing the result: "2 + 3 = 5". At the bottom, there are buttons for "Run CodeGuard" and "Reset". The taskbar at the bottom of the screen shows various application icons.

CodeGuard confirms no issues — clean, documented, and standards-compliant code.



Metrics dashboard highlights Quality Score, Maintainability Index, Complexity, and issue count with clear visualizations for per-file analysis.

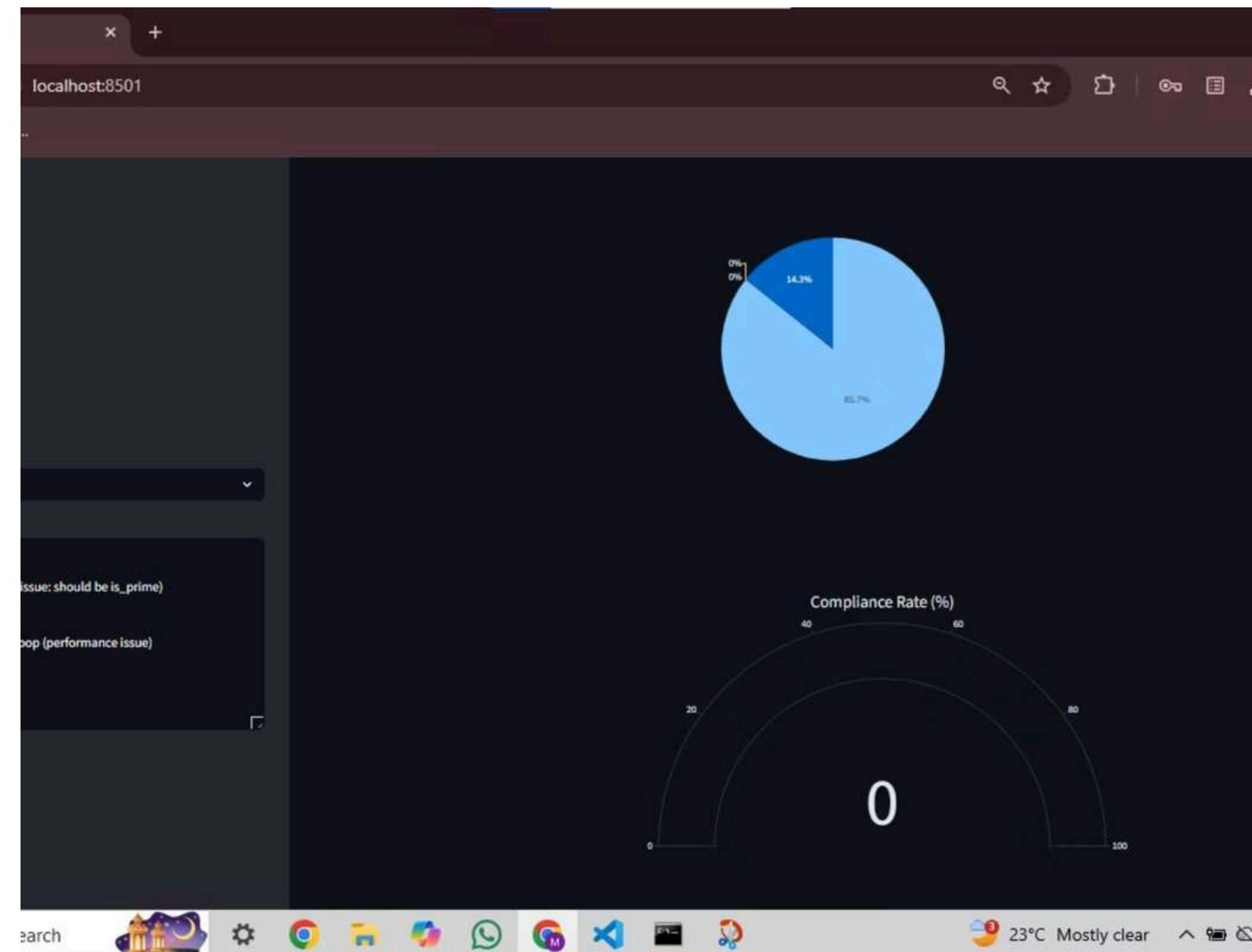
PERFORMANCE & RESULTS

The screenshot shows a browser-based static analysis tool. At the top, there are severity filters for CRITICAL, ERROR, WARNING, INFO, and AI. Below that is a language filter for Python, JavaScript, TypeScript, Java, Kotlin, C, C++, Go, Rust, PHP, HTML, CSS, Shell, and Ruby. The main area displays a file named `manual_input.py` with Python code. The code includes comments like "issue: should be is_prime" and "# Inefficient loop (performance issue)". A sidebar on the left shows a list of issues, including:

- > INFO - Missing docstring in function 'factorial'
- > INFO - Missing return type hint in function 'factorial'
- > INFO - Missing docstring in function 'isPrime'
- > INFO - Missing return type hint in function 'isPrime'
- > INFO - Missing docstring in function 'main'
- > INFO - Missing return type hint in function 'main'
- > CRITICAL - Possible hardcoded secret detected

At the bottom, there's a search bar and a taskbar with various icons.

Static analysis highlights missing docstrings, type hints, and critical security risks, enabling developers to address style and documentation gaps alongside severe issues.



Compliance dashboard visualizes issue severity distribution with pie chart and gauge, highlighting adherence gaps in coding standards.

PERFORMANCE & RESULTS

The image displays two side-by-side screenshots of the CodeGuard web application interface, both titled "Inbox (414) - mahit...".

Screenshot 1 (Left): This screenshot shows the "manual_input.py" file. A "CRITICAL - batched_fix" section is highlighted. The "What's wrong?" section provides a detailed explanation of various security, style/naming, and documentation issues found in the code. Below this, a list of specific problems is presented, such as hardcoded secrets, poor naming conventions, and inefficient loops. The "AI Fix" button is visible at the bottom.

```
def isPrime(n):
    # Poor naming convention (style issue: should be is_prime)
    if n <= 1:
        return False
    for i in range(2, n): # Inefficient loop (performance issue)
        if n % i == 0:
            return False
    return True
```

Screenshot 2 (Right): This screenshot shows the same "manual_input.py" file after AI fixes have been applied. The "Occurrences: 7" section indicates the number of changes made. The code has been modified to remove hardcoded secrets, improve naming conventions, and enhance documentation. The "AI Fix" button is also present here.

```
--- Original
+++ AI Fix
@@ -1,39 +1,40 @@
# Bad Demo File for CodeGuard
# This file intentionally contains issues for testing analyzers.
# Improved Version of CodeGuard Demo File after Security/Style/Docs Reviews. For testing analyzer capabilities while
import os # No longer importing sys, irrelevant to current use cases and not needed herein; additionally introduces
import sys

# Hardcoded secret (security issue)
-API_KEY = "12345-SECRET-KEY"
-
def factorial(n):
    # Missing docstring (docstring issue)
    if n <= 0:
        return -1 # Wrong handling of negative numbers (logic issue)
    def calculate_factorial(n):
        """Returns factorial of given non-negative integer n."""
        # Security/Style Correction: Removed secret key from code to prevent accidental exposure during tests or reviews
        # This function is now more clearly documented and uses a proper variable name, following Python's snake_case convention
        #
        # Ensuring non-negative inputs for maintainable and logical execution flow
        assert n >= 0, f"n must be a non-negative integer."
        result = 1
        for i in range(1, n+1):
            result *= i
        for i in range(2, (n+1)):
            result *= i
        return result

def isPrime(n):
```

AI-assisted fixes remove hardcoded secrets, improve naming conventions, and enhance documentation, producing secure and maintainable code."

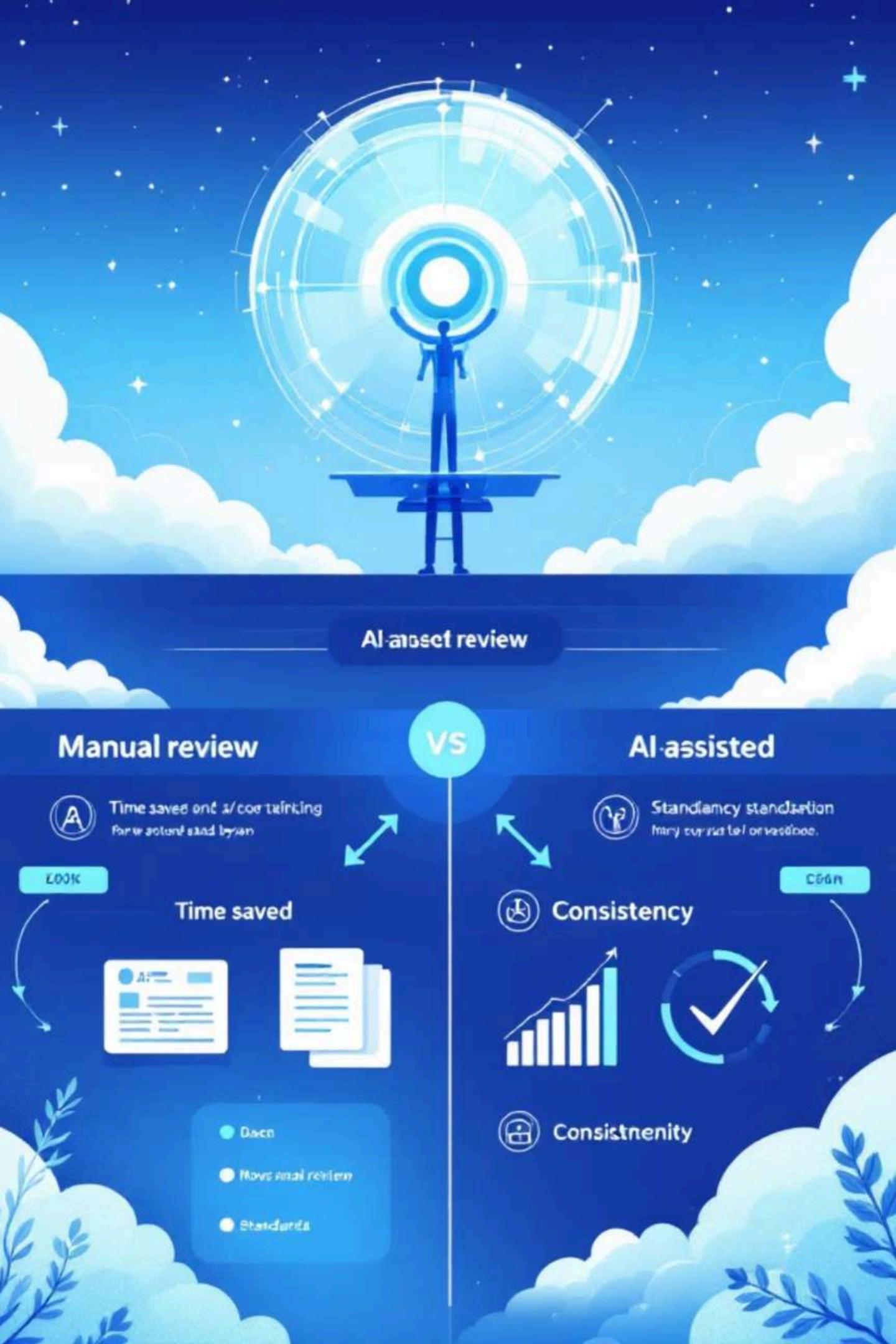
PERFORMANCE & RESULTS

The screenshot shows the CodeGuard web application interface. On the left, there's a sidebar with themes (Dark, Light, Rainbow), input methods (Upload Files, Paste Code), and language selection (Python). Below these are buttons for 'Run CodeGuard' and 'Reset'. At the bottom are 'Export Reports' and 'Download Static Issues JSON'. The main area displays a Python script with several annotations. One annotation on the first line of the `isPrime` function indicates poor naming convention. Another annotation on the `check_prime` function suggests improved naming. A third annotation on the loop iteration indicates an inefficient loop. A fourth annotation on the print statement suggests missing punctuation. A fifth annotation on the API key usage indicates insecure usage. A sixth annotation on the module name suggests maintaining function names. A seventh annotation on the loop iteration indicates an optimized loop. A eighth annotation on the return statement suggests utilizing boolean return statements. A ninth annotation on the documentation indicates appropriate comments. A tenth annotation on the final return statement ensures maintainability and readability. The code itself is a prime number checker with various security and performance issues highlighted.

AI-assisted fixes remove hardcoded secrets, improve naming conventions, and enhance documentation, producing secure and maintainable code."

The screenshot shows the CodeGuard interface with a focus on execution output and static analysis filters. On the left, there's a sidebar with themes, input methods, and language selection (Python). Below these are buttons for 'Run CodeGuard' and 'Reset'. At the bottom are 'Export Reports' and 'Download Static Issues JSON'. The main area has a 'Filters' section with severity levels (CRITICAL, ERROR, WARNING, INFO, AI) and language selection (python, javascript, typescript, java, c++, go, rust, php, html, css). It also includes tabs for 'Static Issues', 'AI Review', 'Metrics', and 'Program Output'. The 'Program Output' tab is active, showing the execution of a file named 'manual_input.py'. The output shows the program running, printing "Hello Mahitha let's do some math.", calculating the factorial of 7 (5040), checking if 7 is prime (True), and using an API key (12345-SECRET-KEY). Below the output are buttons for 'Apply Fixes' and 'Apply AI Fixes'.

Execution output reveals program results alongside static analysis filters, demonstrating detection of style, performance, and security issues across multiple languages.



COMPARISON: MANUAL VS AI-ASSISTED REVIEW



Turnaround Time

Manual: hours–days. CodeGuard: minutes with immediate feedback and gating.



Consistency

Manual results vary by reviewer. AI enforces deterministic rules and documented rationale.



Quality Assurance

Automated severity ranking reduces human oversight errors and prevents regressions via CI gates.



Scalability

AI-assisted scans scale across repositories and languages with plugin analyzers and LLM prompting.

FUTURE ENHANCEMENTS

Multi-language support

Extending language-specific environments for fixes across Java, C++, and other multi-languages.

Advanced Streamlit UI

Filters, search, tooltips, audit trails and collaborative review modes.

Packaging & Exports

Publish as pip package, add CSV/HTML report exports and CI integrations for enterprise pipelines.



CONCLUSION

CodeGuard automates code quality by accelerating reviews, ensuring consistency, and detecting security and maintainability issues at an early stage. With seamless integration into Git workflows, it enforces quality gates without disrupting developer productivity. The system is production-ready, robust, and designed with extensibility in mind, making it well-suited for future enhancements and broader language support.

Thank You

CodeGuard — Reliable, Secure, Future-Ready.

Submitted By: TEAM 1

Mahitha Chippada

Sahasra Doddi

Lakshmi Priya Vuyyuru

Rishitha Rallapalli

Jyotsnamathi J

Thottempudi Venkat Sainadh

Disha R Gowda