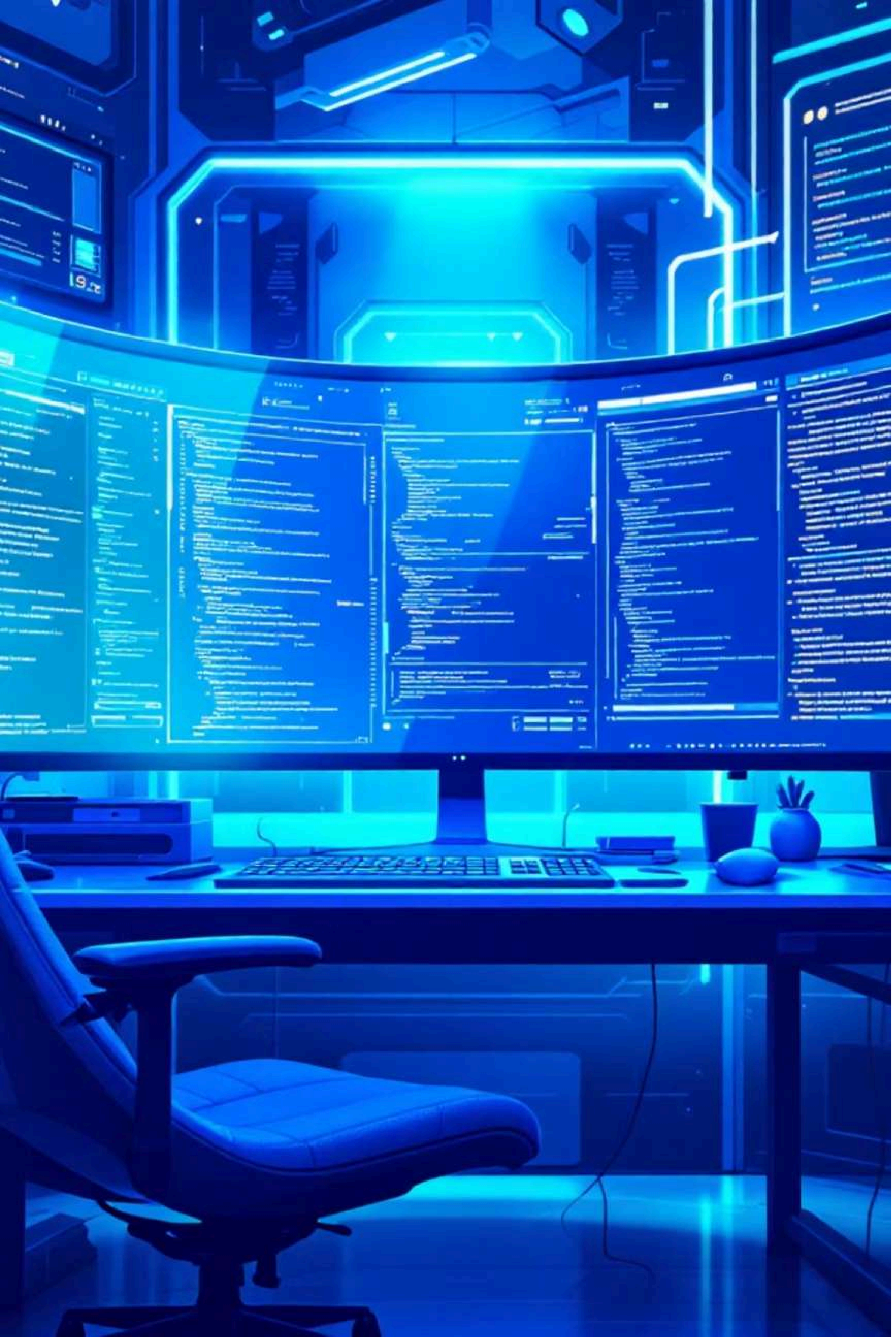


CodeGuard – AI-Powered Code Reviewer and Quality Assistant

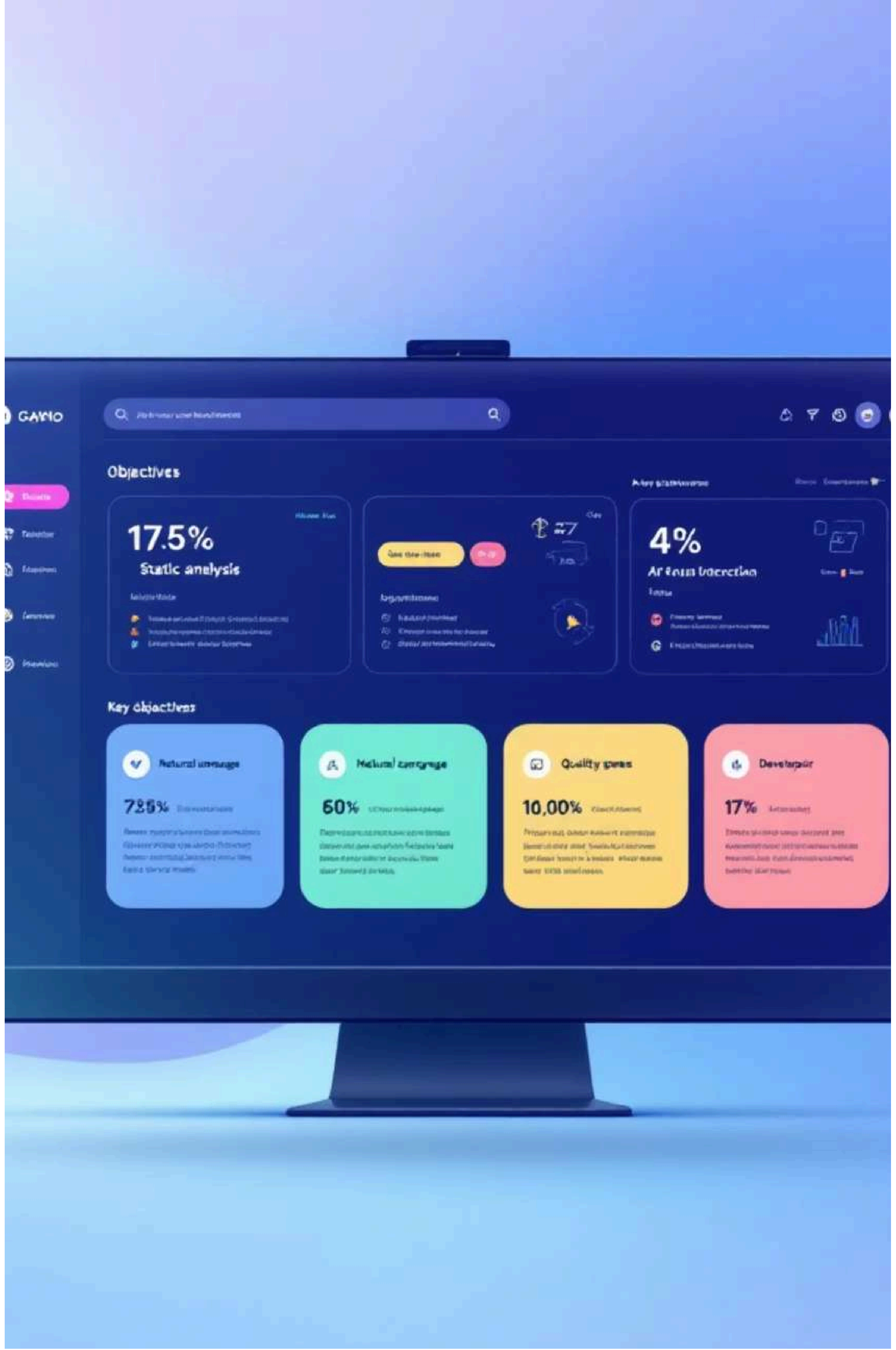
Presented By: Ch Mahitha

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.

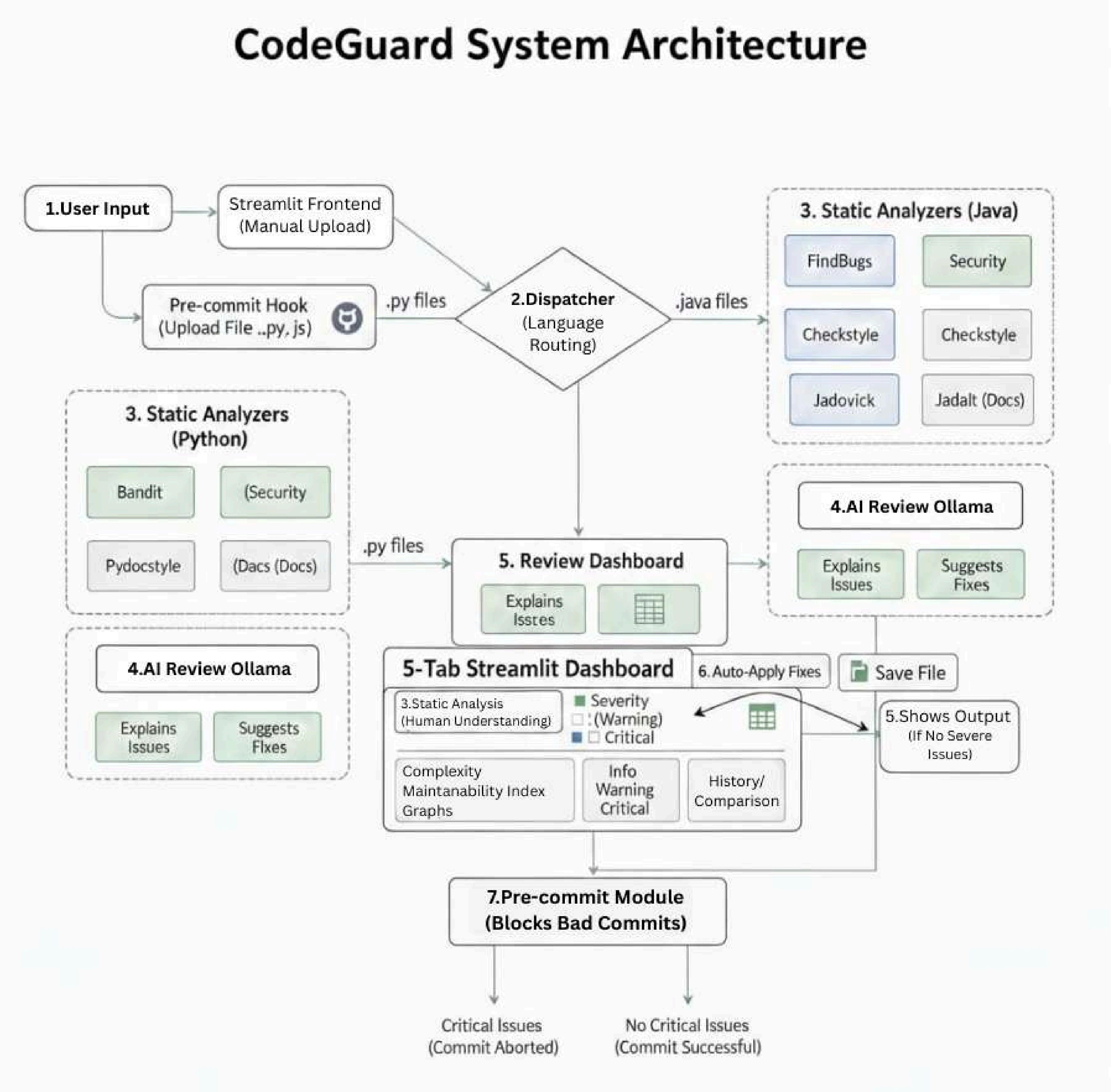
Developer interfaces

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

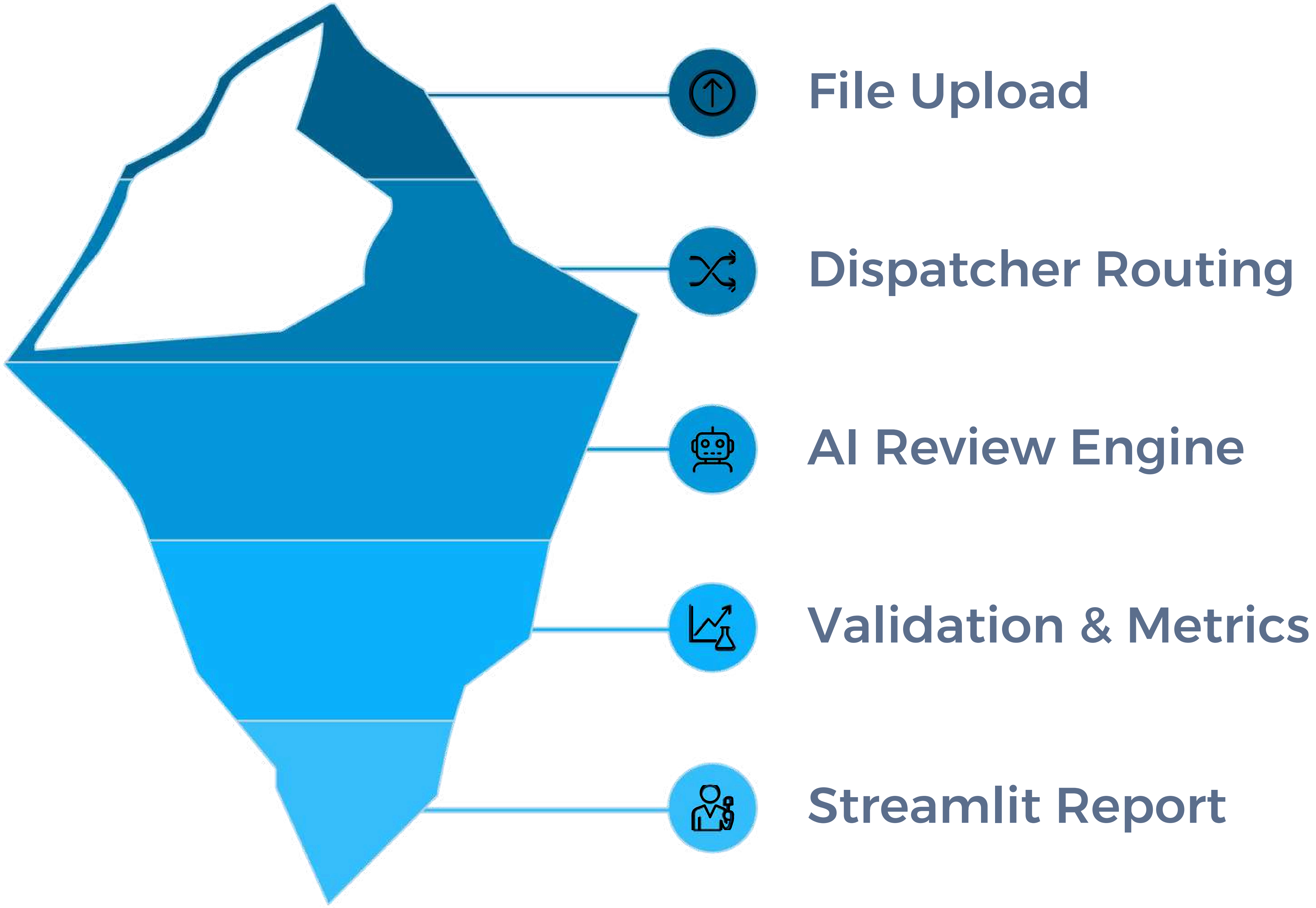

CodeGuard System Architecture
1.User Input
Streamlit Frontend (Manual Upload)
Pre-commit Hook (Upload File ..py, js)
.py files
2.Dispatcher (Language Routing)
.java files
3. Static Analyzers (Java)
FindBugs
Security
Checkstyle
Checkstyle
Jadovick
Jadalt (Docs)
3. Static Analyzers (Python)
Bandit
(Security
Pydocstyle
(Dacs (Docs)
.py files
5. Review Dashboard
Explains Isstes
4.AI Review Ollama
Explains Issues
Suggests Fixes
4.AI Review Ollama
Explains Issues
Suggests Fixes
5-Tab Streamlit Dashboard
6. Auto-Apply Fixes
Save File
3.Static Analysis (Human Understanding)
Severity
:(Warning)
Critical
5.Shows Output (If No Severe Issues)
Complexity Maintanability Index Graphs
Info Warning Critical
History/ Comparison
7.Pre-commit Module (Blocks Bad Commits)
Critical Issues (Commit Aborted)
No Critical Issues (Commit Successful)

- **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



File Upload

Dispatcher Routing

AI Review Engine

Validation & Metrics

Streamlit Report

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.

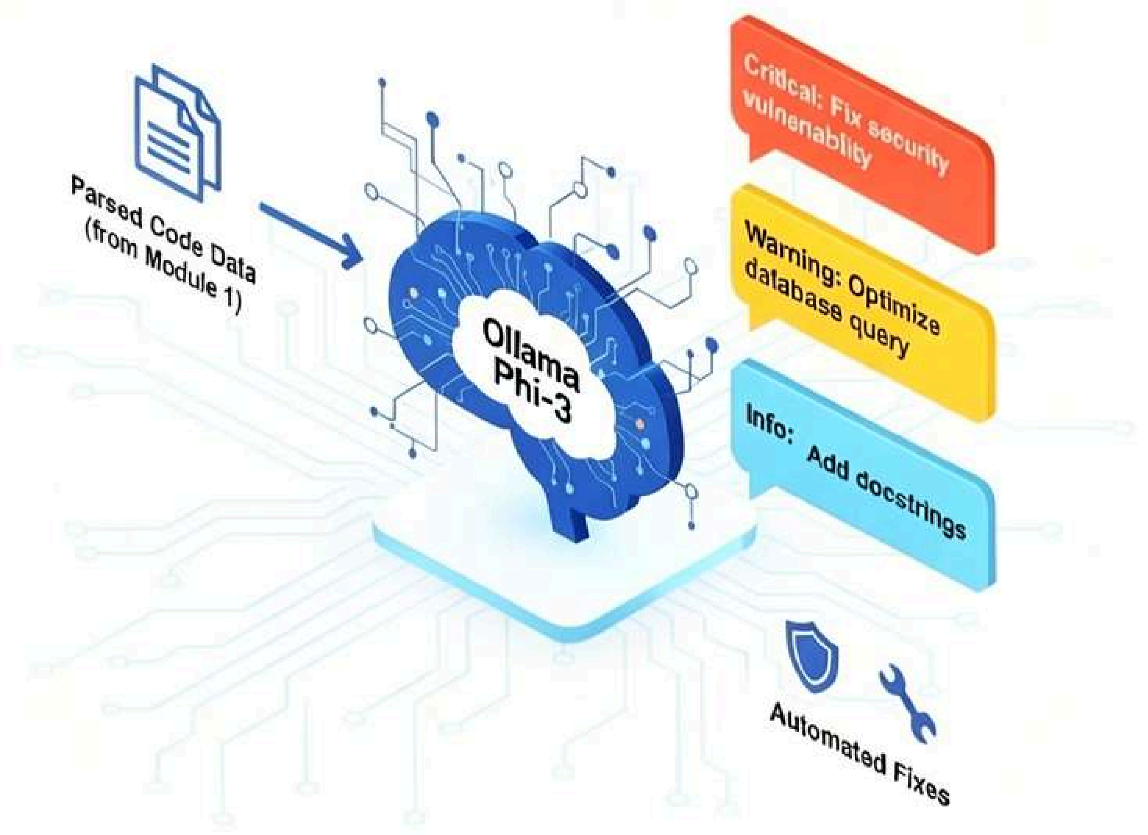
MODULE 1 : CODE PARSING & ANALYSIS

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.

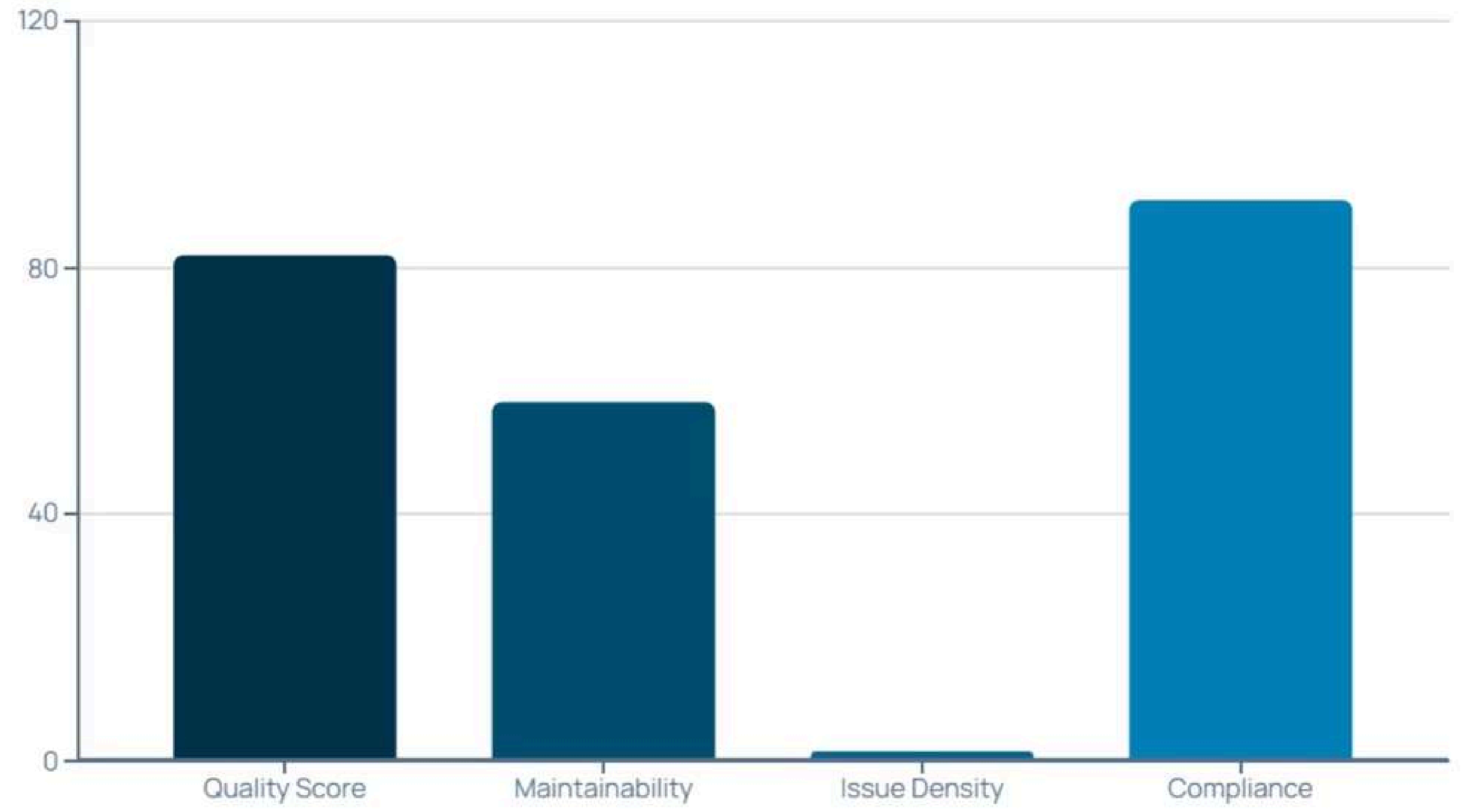


Parsed Code Data (from Module 1)
Ollama Phi-3
Critical: Fix security vulnenablity
Warning: Optimize database query
Info: Add docstrings
Automated Fixes

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

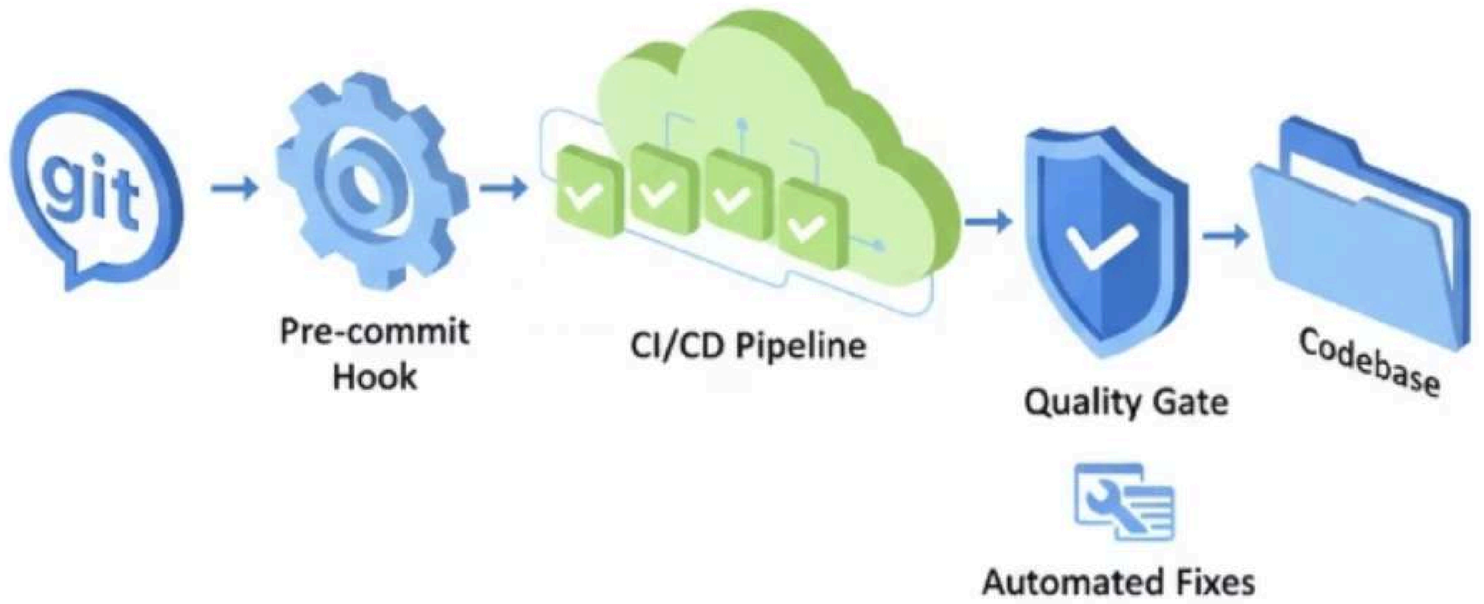
Command Set

- 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.

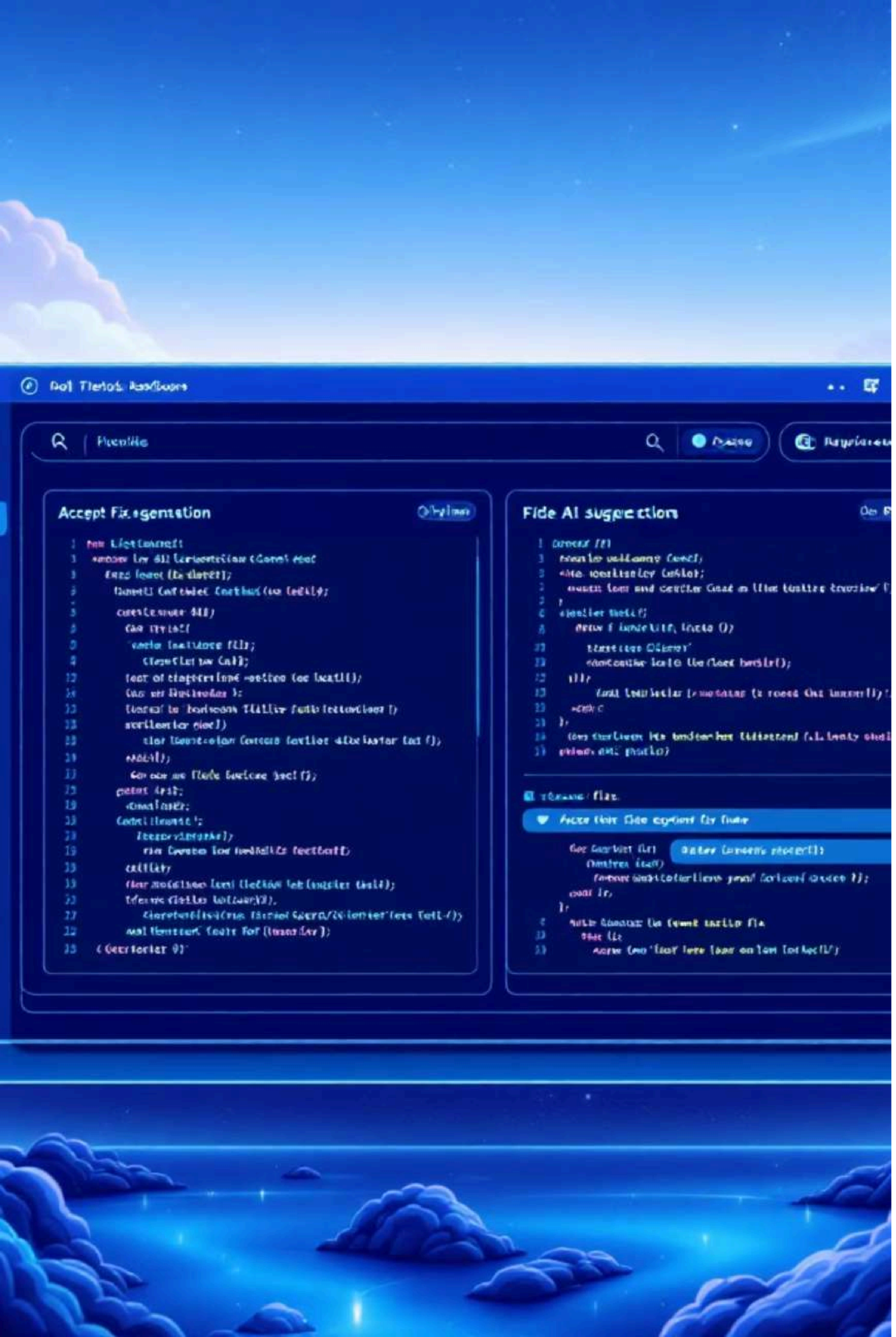
Capabilities

- 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



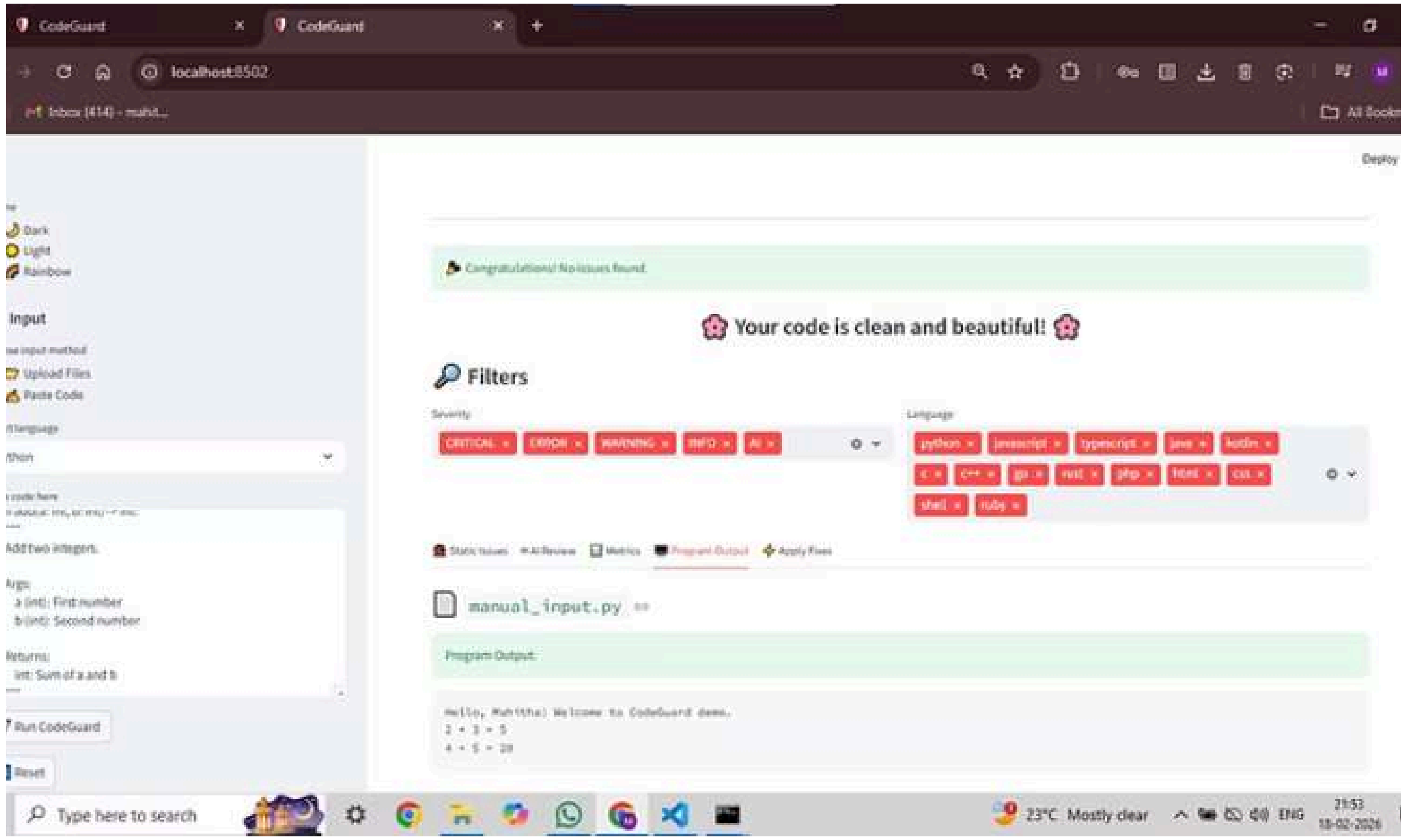
- Pre-commit hook reviews staged files before commit; CI/CD templates enforce quality gates in pipelines
- Fast execution (<5s on staged changes) ensures smooth developer workflow without friction
- Coverage thresholds (e.g., ≥90%) block builds if standards are not met
- Seamless integration with GitHub/GitLab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



Congratulations! No issues found.

Your code is clean and beautiful!

Filters

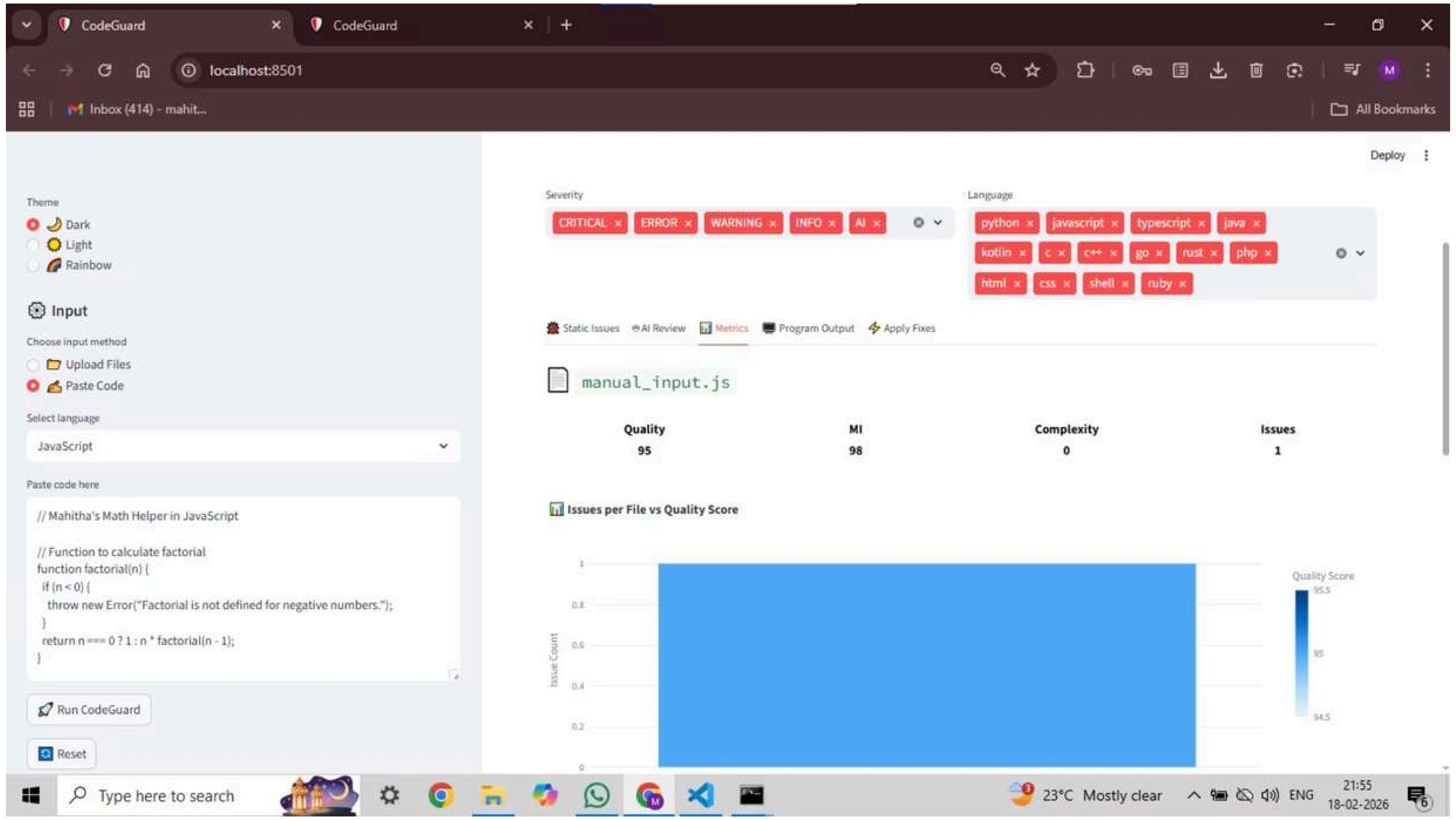
manual_input.py

Program Output

```
Hello, Mahitha! Welcome to CodeGuard demo.
2 + 3 = 5
4 * 5 = 20
```

```
Add two integers.

Args:
    a (int): First number
    b (int): Second number

Returns:
    int: Sum of a and b
```

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



Theme: ● Dark ○ Light ○ Rainbow

Choose input method: ○ Upload Files ● Paste Code

Select language: JavaScript

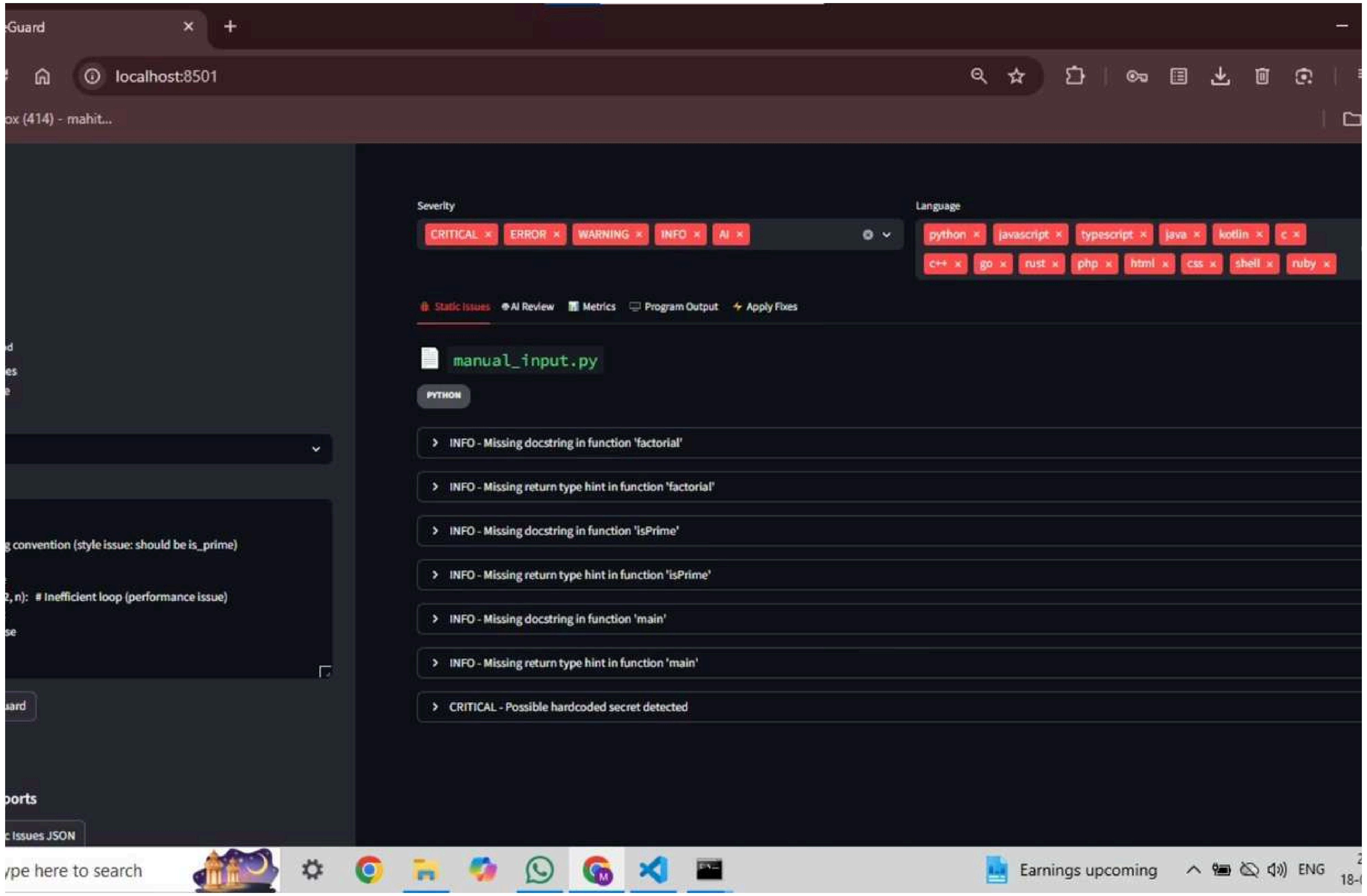
```
// Mahitha's Math Helper in JavaScript

// Function to calculate factorial
function factorial(n) {
  if (n < 0) {
    throw new Error("Factorial is not defined for negative numbers.");
  }
  return n === 0 ? 1 : n * factorial(n - 1);
}
```

manual_input.js

Quality	MI	Complexity	Issues
95	98	0	1

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

PERFORMANCE & RESULTS

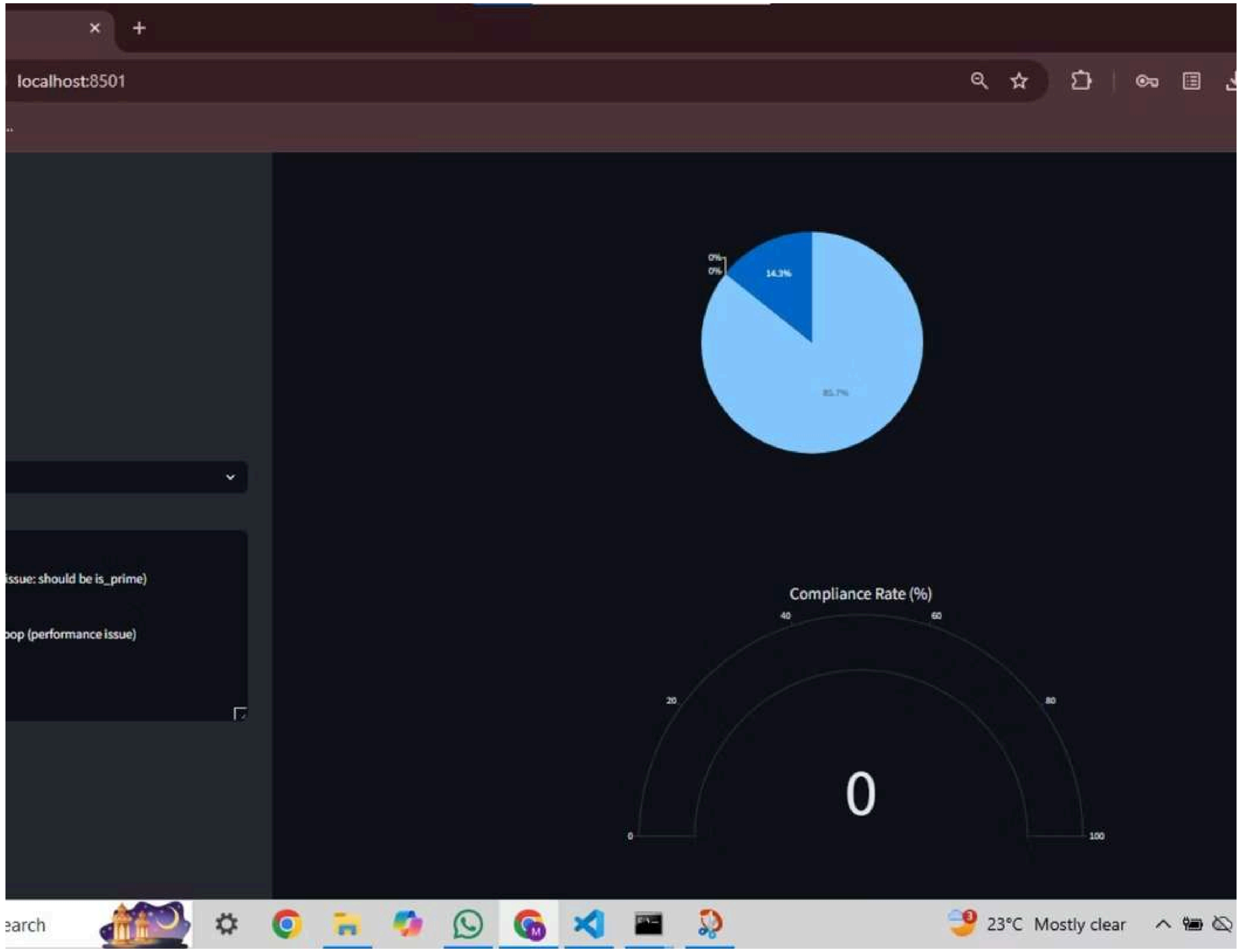

Guard
localhost:8501
Severity
CRITICAL × ERROR × WARNING × INFO × AI ×
Language
python × javascript × typescript × java × kotlin × c ×
c++ × go × rust × php × html × css × shell × ruby ×
Static Issues
AI Review
Metrics
Program Output
Apply Fixes
PYTHON
Type here to search
Earnings upcoming
ENG

manual_input.py

```
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
```

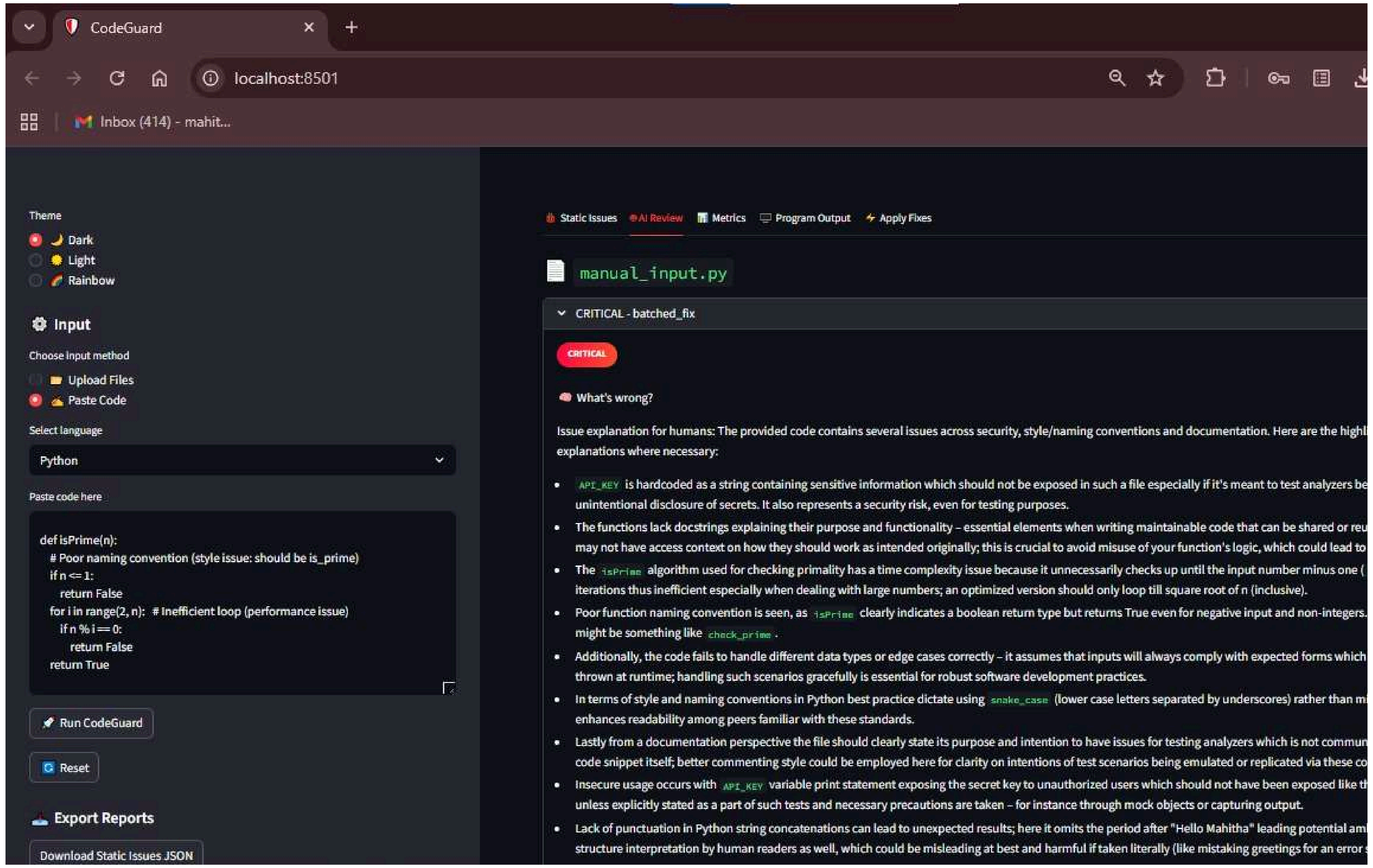
```
convention (style issue: should be is_prime)
2, n):  # Inefficient loop (performance issue)
```

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



Theme

- ● 🌙 Dark
- ○ 🌞 Light
- ○ 🌈 Rainbow

⚙ Input

Choose input method

- ○ 📁 Upload Files
- ● 🍝 Paste Code

Select language: Python

Paste code here

```
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
```

manual_input.py

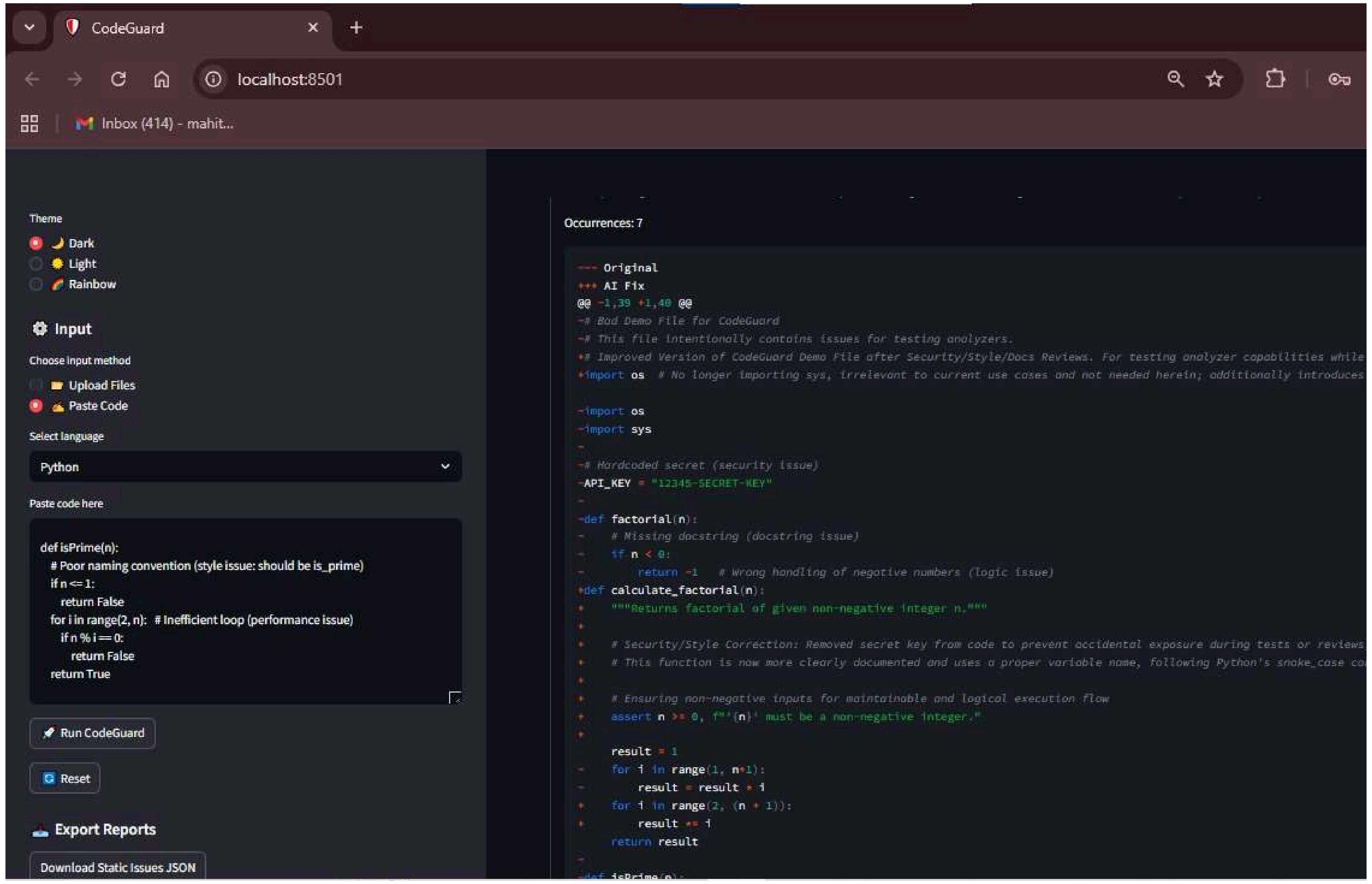
CRITICAL - batched_fix

CRITICAL

What's wrong?

Issue explanation for humans: The provided code contains several issues across security, style/naming conventions and documentation. Here are the highl
explanations where necessary:

- `API_KEY` is hardcoded as a string containing sensitive information which should not be exposed in such a file especially if it's meant to test analyzers be
unintentional disclosure of secrets. It also represents a security risk, even for testing purposes.
- The functions lack docstrings explaining their purpose and functionality – essential elements when writing maintainable code that can be shared or reu
may not have access context on how they should work as intended originally; this is crucial to avoid misuse of your function's logic, which could lead to
- The `isPrime` algorithm used for checking primality has a time complexity issue because it unnecessarily checks up until the input number minus one (
iterations thus inefficient especially when dealing with large numbers; an optimized version should only loop till square root of n (inclusive).
- Poor function naming convention is seen, as `isPrime` clearly indicates a boolean return type but returns True even for negative input and non-integers.
might be something like `check_prime` .
- Additionally, the code fails to handle different data types or edge cases correctly – it assumes that inputs will always comply with expected forms which
thrown at runtime; handling such scenarios gracefully is essential for robust software development practices.
- In terms of style and naming conventions in Python best practice dictate using `snake_case` (lower case letters separated by underscores) rather than mi
enhances readability among peers familiar with these standards.
- Lastly from a documentation perspective the file should clearly state its purpose and intention to have issues for testing analyzers which is not commun
code snippet itself; better commenting style could be employed here for clarity on intentions of test scenarios being emulated or replicated via these co
- Insecure usage occurs with `API_KEY` variable print statement exposing the secret key to unauthorized users which should not have been exposed like th
unless explicitly stated as a part of such tests and necessary precautions are taken – for instance through mock objects or capturing output.
- Lack of punctuation in Python string concatenations can lead to unexpected results; here it omits the period after "Hello Mahitha" leading potential am
structure interpretation by human readers as well, which could be misleading at best and harmful if taken literally (like mistaking greetings for an error



Theme

- ● 🌙 Dark
- ○ 🌞 Light
- ○ 🌈 Rainbow

⚙ Input

Choose input method

- ○ 📁 Upload Files
- ● 🍝 Paste Code

Select language: Python

Paste code here

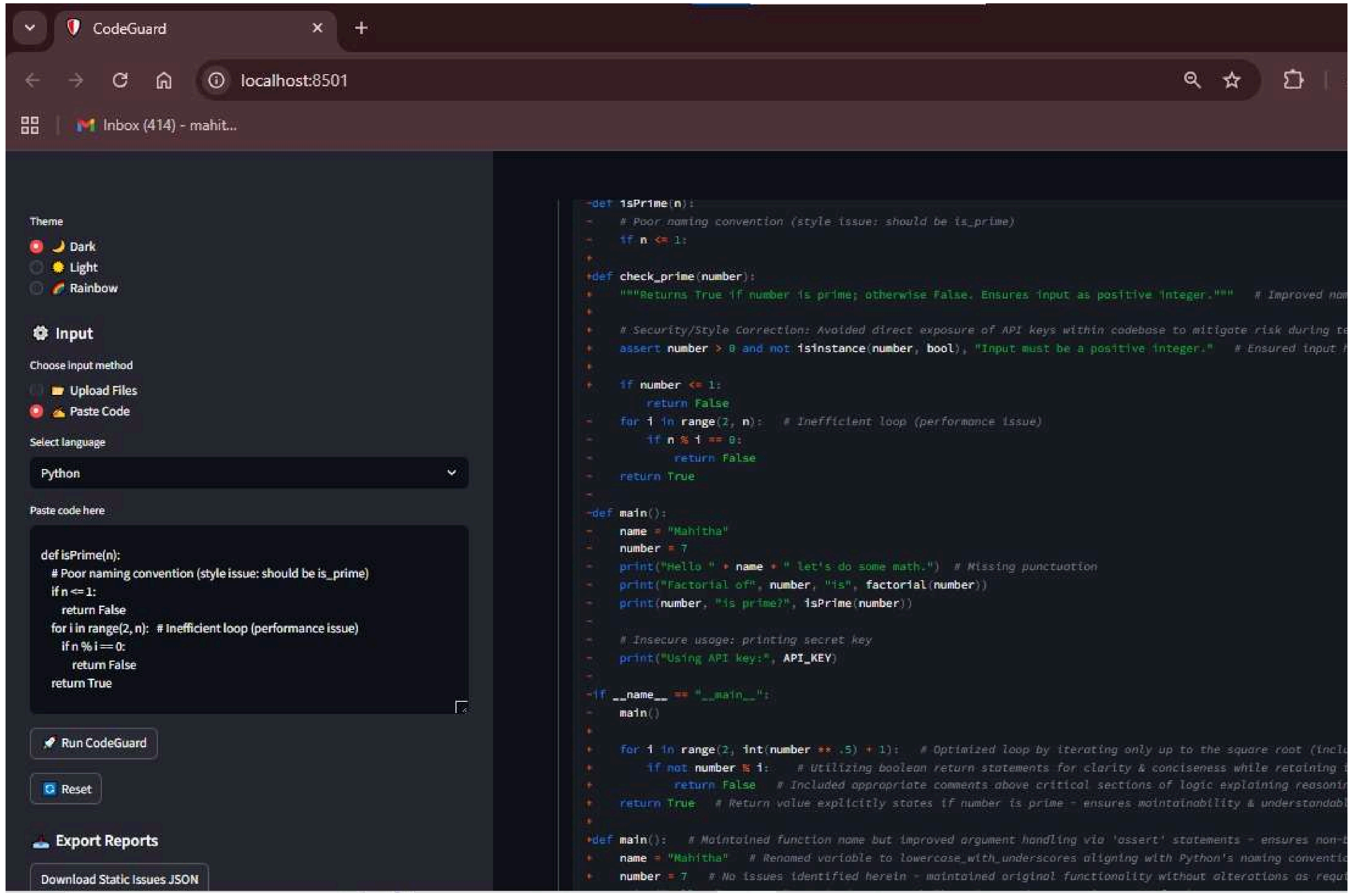
```
def isPrime(n):
  # Poor naming convention (style issue: should be is_prime)
  if n <= 1:
    return False
  for i in range(2, n):  # Inefficient loop (performance issue)
    if n % i == 0:
      return False
  return True
```

Occurrences: 7

```
--- 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 os
-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 co
+
+    # 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 = result * i
+    for i in range(2, (n + 1)):
+        result *= i
     return result
-
-def isPrime(n):
```

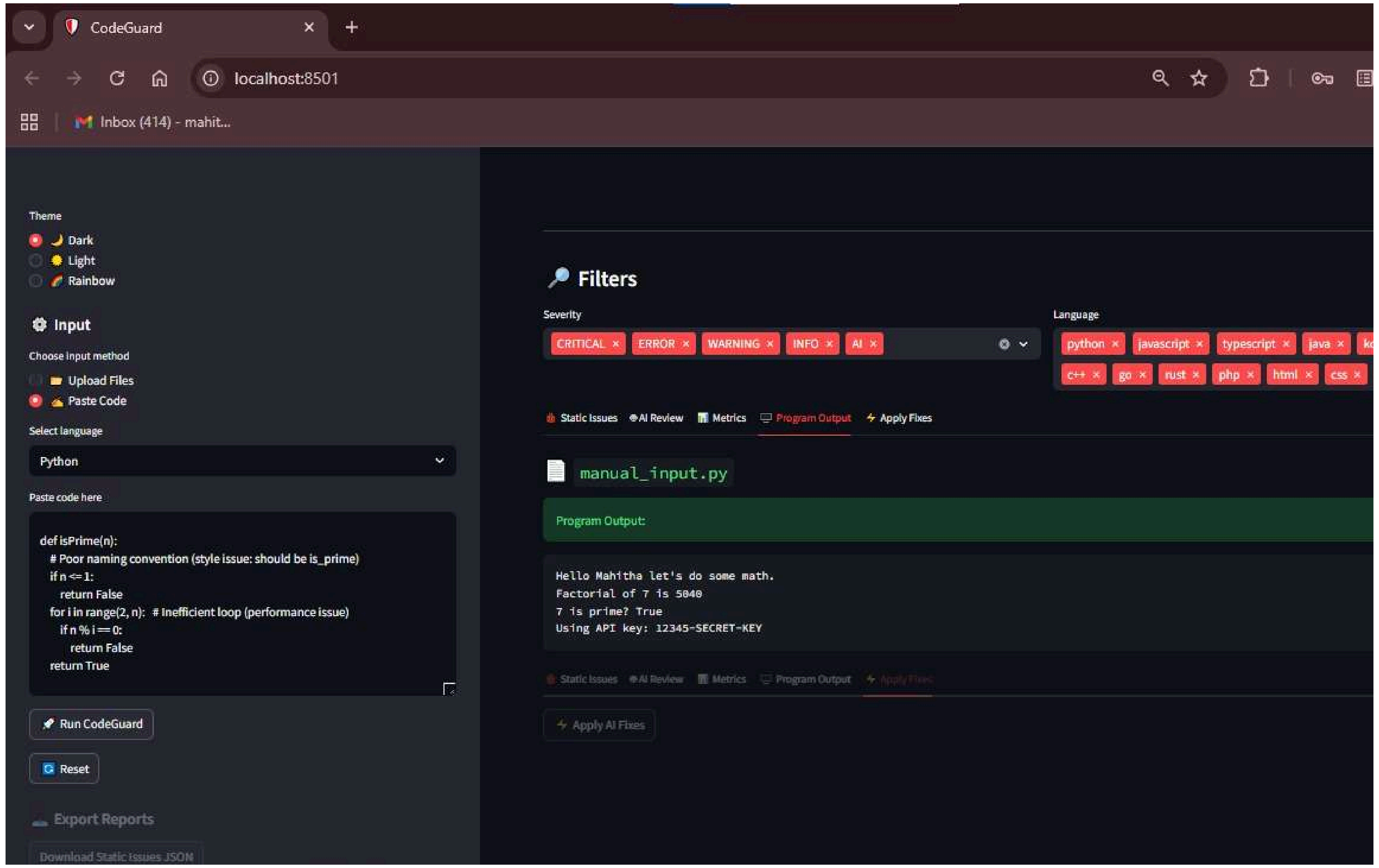
AI-assisted fixes remove hardcoded secrets, improve naming conventions, and enhance documentation, producing secure and maintainable code.”

PERFORMANCE & RESULTS



```python
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
```

```diff
-def isPrime(n):
-    # Poor naming convention (style issue: should be is_prime)
-    if n <= 1:
+
+def check_prime(number):
+    """Returns True if number is prime; otherwise False. Ensures input as positive integer."""   # Improved nam
+
+    # Security/Style Correction: Avoided direct exposure of API keys within codebase to mitigate risk during te
+    assert number > 0 and not isinstance(number, bool), "Input must be a positive integer."   # Ensured input h
+
+    if number <= 1:
         return False
-    for i in range(2, n):  # Inefficient loop (performance issue)
-        if n % i == 0:
-            return False
-    return True
-
-def main():
-    name = "Mahitha"
-    number = 7
-    print("Hello " + name + " let's do some math.")  # Missing punctuation
-    print("Factorial of", number, "is", factorial(number))
-    print(number, "is prime?", isPrime(number))
-
-    # Insecure usage: printing secret key
-    print("Using API key:", API_KEY)
-
-if __name__ == "__main__":
-    main()
+
+    for i in range(2, int(number ** .5) + 1):   # Optimized loop by iterating only up to the square root (incl
+        if not number % i:    # Utilizing boolean return statements for clarity & conciseness while retaining i
+            return False   # Included appropriate comments above critical sections of logic explaining reasoni
+    return True   # Return value explicitly states if number is prime - ensures maintainability & understandabl
+
+def main():   # Maintained function name but improved argument handling via 'assert' statements - ensures non-b
+    name = "Mahitha"   # Renamed variable to lowercase_with_underscores aligning with Python's naming conventi
+    number = 7   # No issues identified herein - maintained original functionality without alterations as requ
```



```python
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
```

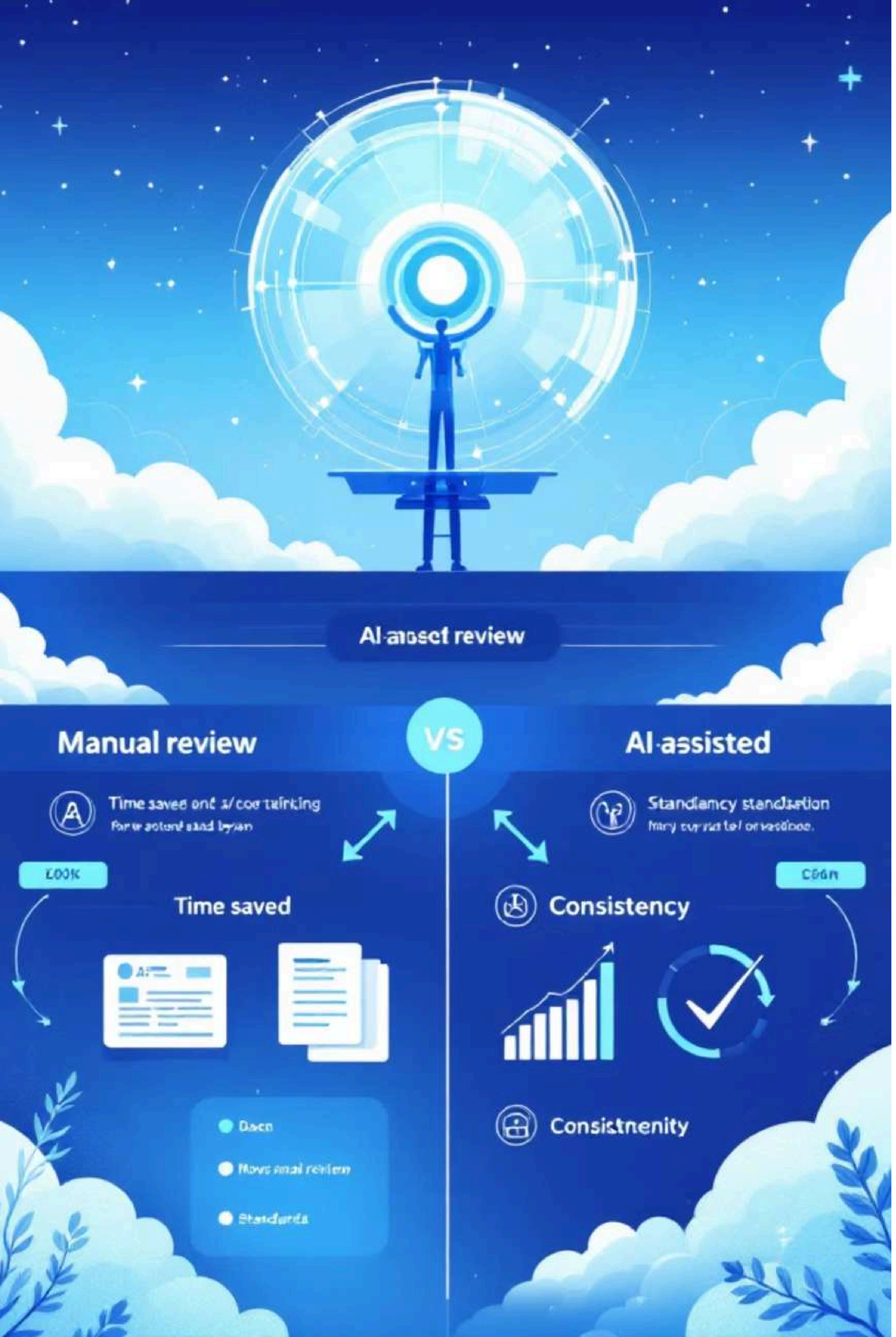
manual_input.py

Program Output:

```
Hello Mahitha let's do some math.
Factorial of 7 is 5040
7 is prime? True
Using API key: 12345-SECRET-KEY
```

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

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

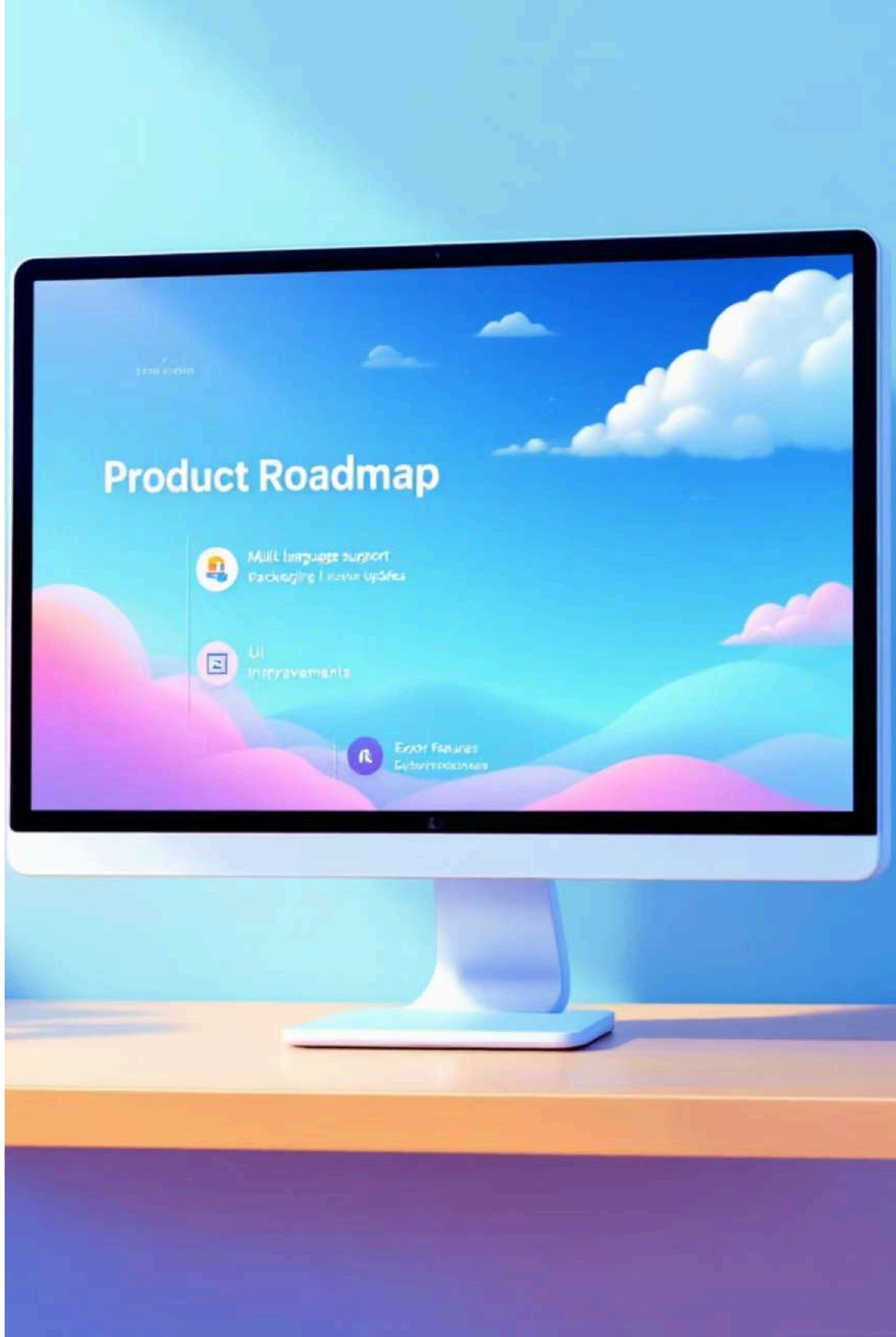
Extending language-specific environments for fixes across Java, C++, and other multi- languages.

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.