CodeGuard – AI Powered Code Reviewer and Quality Assistant

Introduction

Software quality depends heavily on consistent code reviews, but manual reviews are often time-consuming, inconsistent, and prone to human oversight. To address these challenges, this project introduces an AI-assisted code review tool that combines static analysis with Ollama Phi-3 feedback to deliver actionable suggestions. The tool integrates seamlessly with CLI, Streamlit UI, and Git workflows, providing developers with automated checks, natural-language explanations, and auto-fixes.

Module 1: Static Analysis

Purpose: This module performs static analysis of Python files to detect maintainability, complexity, documentation, naming, and security issues. It uses Python's built-in `ast` (Abstract Syntax Tree) library to parse code and analyze its structure without executing it.

Code Explanation:

1. **Imports**
   - `os`: Used to handle file paths and extensions.
   - `ast`: Provides tools to parse Python source code into an abstract syntax tree for analysis.
2. **ComplexityVisitor Class**
   - Inherits from `ast.NodeVisitor`.
   - Tracks **cyclomatic complexity** by incrementing a counter whenever control flow structures (`if`, `for`, `while`, `try`, `with`) are encountered.
   - Maintains `max_depth` and `current_depth` to measure nesting depth of code blocks.
   - This helps identify overly complex functions or code segments.
3. **analyze_python(file_path)**
   - Reads the file safely, returning a critical issue if the file cannot be opened.
   - Parses the code into an AST; if a syntax error occurs, it reports it as a **CRITICAL issue**.
   - Uses `ComplexityVisitor` to calculate cyclomatic complexity.
     - If complexity > 10, adds a **WARNING** about high complexity.
   - Walks through AST nodes to check:
     - **Function length**: Flags functions longer than 50 lines.
     - **Docstrings**: Reports missing docstrings as **INFO**.
     - **Type hints**: Reports missing return type hints as **INFO**.
     - **Class naming**: Ensures PascalCase naming; warns if violated.
   - Scans raw code for **security risks**:
     - Hardcoded secrets (`password`, `api_key`, `token`, etc.) → **CRITICAL**.
     - Use of `eval` or `exec` → **CRITICAL**.

4. **analyze_file(file_path)**
    - Wrapper function that dispatches analysis based on file extension.
    - Currently supports `.py` files only; other extensions return an **INFO issue** stating "Language not supported yet."

Outputs:

- Returns a list of issues, each with:
    - `issue`: Description of the problem.
    - `severity`: INFO, WARNING, or CRITICAL.
    - `line`: Line number (if applicable).
    - `category`: Type of issue (complexity, maintainability, documentation, naming, security, etc.).

Module 2: AI Review Engine

Purpose: This module integrates **Ollama Phi-3** to provide human-like code review feedback. It takes the static analysis results from Module 1, classifies issues, and generates natural-language explanations and suggested fixes. It also supports a feedback loop where developers can accept or reject AI suggestions.

Code Explanation:

1. **Imports**
    - `json`, `os`: For file handling and structured output.
    - `requests`: To send prompts to the local Ollama API server.
    - `defaultdict`: For grouping issues by category.
2. **Knowledge Base (Fallback Templates)**
    - `ISSUE_KB`: A dictionary of predefined reviews and suggestions for common categories (documentation, style, security).
    - Used when AI is disabled or unavailable, ensuring consistent feedback.
3. **Classification Logic**
    - `classify_issue(category)`: Maps issue categories (e.g., "doc", "style", "security") to standardized keys.
    - Ensures issues are grouped consistently before review.
4. **Ollama Integration**
    - `ollama_generate(issue_text, code_snippet, model="phi3")`:
        - Builds a **prompt** for the Ollama Phi-3 model, asking it to explain the issue and provide corrected code.
        - Sends the prompt to Ollama's local API (`http://localhost:11434/api/generate`).
        - Streams responses line-by-line, decoding JSON output.
        - Splits the response into **explanation** (human-readable feedback) and **code fix** (suggested correction).
        - Returns either the AI's output or an error message if Ollama is unavailable.

5. **Main Review Function**
    - `generate_ai_review(static_results, use_llm=True, model="phi3")`:
        - Iterates over static analysis results.
        - Groups issues by category using `defaultdict`.
        - For each group:
            - Builds a `review_entry` with severity, occurrences, line number, and code snippet.
            - If `use_llm=True`, calls Ollama Phi-3 for natural-language explanation and code fix.
            - If `use_llm=False`, falls back to `ISSUE_KB` templates.
        - Collects all reviews into a structured JSON output per file.
6. **Feedback Loop**
    - `log_feedback(file, issue_type, decision, log_file="feedback_log.json")`:
        - Records developer decisions (accept/reject) for each issue.
        - Appends feedback entries to a JSON log file.
        - Helps track AI suggestion effectiveness and developer trust.
7. **Demo Runner**
    - Provides a sample `static_results` input with documentation, style, and security issues.
    - Runs `generate_ai_review` with `use_llm=True` and prints structured JSON output.
    - Demonstrates how Ollama Phi-3 generates explanations and fixes.

Outputs:

- JSON structure per file with:
    - `type`: Issue category.
    - `severity`: INFO, WARNING, ERROR, CRITICAL.
    - `review`: Human-like explanation.
    - `suggestion`: Code fix or best practice.
    - `auto_fix_recommended`: Boolean flag for auto-fix applicability.

Module 3: Validation & Metrics

Purpose: This module computes project-level and file-level metrics to quantify code quality. It enforces **quality gates** (minimum thresholds for Quality Score and Maintainability Index), identifies best and worst files, and provides severity-weighted scoring to prioritize issues. The results are summarized for easy reporting and integration.

Code Explanation:

1. **Severity Weights**
    - Defined in `severity_weights = {"CRITICAL": 20, "ERROR": 15, "WARNING": 10, "INFO": 5}`.
    - Each issue deducts points from a file's base score (100).
    - Higher severity issues reduce the score more significantly.

2. **File-Level Metrics** For each file in `static_results`:
    - **Quality Score**: Starts at 100, reduced by severity weights.
    - **Severity Breakdown**: Counts of CRITICAL, ERROR, WARNING, INFO issues.
    - **Maintainability Index**: Weighted formula penalizing complexity and severity counts.
    - **Average Severity**: Numeric average of severity levels (CRITICAL=4, ERROR=3, WARNING=2, INFO=1).
    - **Issue Density**: Issues per 100 lines of code.
    - **Quality Gate Check**: File passes if `score ≥ 70` and `maintainability_index ≥ 50`.
3. **Project-Level Summary** After analyzing all files, the module computes:
    - **Average Quality Score** across files.
    - **Average Maintainability Index**.
    - **Total Issues** detected.
    - **Files Analyzed** count.
    - **Compliance Rate**: Percentage of files passing quality gates.
    - **Best File**: File with highest quality score.
    - **Worst File**: File with lowest quality score.
    - **Category Distribution**: Frequency of issues by category.
4. **Return Structure**
    - Returns a dictionary with:
        - `"files"`: List of per-file metrics.
        - `"summary"`: Project-level aggregated metrics.

Outputs:

- Each file entry includes:
    - `quality_score`, `cyclomatic_complexity`, `issue_count`, `severity_breakdown`, `maintainability_index`, `average_severity`, `issue_density_per_100_lines`, `passed_quality_gate`.
- Project summary includes averages, compliance rate, best/worst files, and category distribution.

Module 4: CLI & Configuration

Purpose: This module provides a **command-line interface (CLI)** for CodeGuard, enabling developers to interact with the tool directly from the terminal. It wires together the static analysis (Module 1), AI review (Module 2), and metrics (Module 3), while also offering commands for auto-fixing and file comparison. It ensures seamless integration into developer workflows.

Code Explanation:

1. **Imports**
    - `click`: Used to define CLI commands and arguments.
    - `json`, `os`: For file handling and structured output.
    - `analyze_file`, `generate_ai_review`, `compute_metrics`: Imported from Modules 1–3 to provide functionality.

2. **CLI Group**
    - `@click.group()` defines the root command `main()`.
    - All subcommands (`scan`, `review`, `apply`, `report`, `diff`) are registered under this group.
3. **Command: scan**
    - `scan(path)`: Runs static analysis on the given file path.
    - Calls `analyze_file(path)` and prints issues in JSON format.
    - Useful for quick checks without AI involvement.
4. **Command: review**
    - `review(path)`: Runs AI-powered review using Ollama Phi-3.
    - Wraps static analysis results into a structured format.
    - Calls `generate_ai_review()` with `use_llm=True`.
    - Outputs JSON with human-like explanations and suggestions.
5. **Command: apply**
    - `apply(path)`: Auto-fixes code using AI suggestions and formats with **Black**.
    - Steps:
        - Run static analysis.
        - Generate AI review suggestions.
        - Apply corrected code snippets directly to the file.
        - Run `black` for consistent formatting.
    - Outputs messages indicating which fixes were applied.
    - Currently supports only Python files.
6. **Command: report**
    - `report(path)`: Generates metrics report for the file.
    - Calls `compute_metrics()` from Module 3.
    - Prints per-file metrics and project summary in JSON format.
    - Useful for quality gate enforcement and reporting.
7. **Command: diff**
    - `diff(file1, file2)`: Compares two files line by line.
    - Prints differences with line numbers, showing removed (`-`) and added (`+`) lines.
    - Useful for reviewing changes between original and AI-fixed code.
8. **Entry Point**
    - `if __name__ == "__main__": main()` ensures the CLI runs when the script is executed directly.

Outputs:

- JSON reports for `scan`, `review`, and `report`.
- Applied fixes and formatting messages for `apply`.
- Line-by-line differences for `diff`.

Module 5: VCS & CI Integration

Purpose: This module integrates CodeGuard into **version control workflows**. It ensures that code quality checks run automatically during commits (via pre-commit hooks) and in CI/CD pipelines (e.g., GitLab). This enforces consistency and prevents low-quality code from entering the repository.

GitLab CI Pipeline (`.gitlab-ci.yml`)

```
stages:
  - quality

codeguard_quality:
  stage: quality
  image: python:3.10
  script:
    - pip install .
    - codeguard scan .
    - codeguard review --llm || true
    - codeguard report || true
  artifacts:
    paths:
      - module1_report.json
      - ai_review.json
      - module3_metrics.json
```

Explanation:

- **Stage:** Defines a `quality` stage for code review.
- **Image:** Uses `python:3.10` Docker image for execution.
- **Script:**
  - Installs CodeGuard (`pip install .`).
  - Runs `scan` for static analysis.
  - Runs `review` with Ollama Phi-3 (`--llm`).
  - Runs `report` to generate metrics.
- **Artifacts:** Stores JSON outputs (`module1_report.json`, `ai_review.json`, `module3_metrics.json`) for later inspection.

This ensures every pipeline run produces **analysis, AI review, and metrics reports**.

Pre-Commit Hook (`.pre-commit-config.yaml`)

```
#!/bin/bash
# Pre-commit hook for CodeGuard

# Get staged Python files (recursively)
FILES=$(git diff --cached --name-only --diff-filter=ACM | grep -E '\.py$')

if [ -n "$FILES" ]; then
  echo "Running CodeGuard scan on staged Python files..."
  for f in $FILES; do
    if [ -f "$f" ]; then
      OUTPUT=$(codeguard scan "$f")
      echo "$OUTPUT"

      # Block commit if CRITICAL or ERROR issues are found
      if echo "$OUTPUT" | grep -q '"severity": "CRITICAL"' || \
         echo "$OUTPUT" | grep -q '"severity": "ERROR"'; then
        echo "❌ Commit blocked: CodeGuard found CRITICAL/ERROR issues in
$f"
        exit 1
      fi
```

```
      # Warn if INFO or WARNING issues are found
      if echo "$OUTPUT" | grep -q '"severity": "INFO"' || \
         echo "$OUTPUT" | grep -q '"severity": "WARNING"'; then
        echo "⚠ Commit warning: CodeGuard found INFO/WARNING issues in
$f"
      fi
    fi
  done
fi

echo "✅ Commit passed CodeGuard checks"
exit 0
```

Explanation:

- **repo: local** → Defines custom hooks for CodeGuard.
- **Hooks:**
  - `codeguard-scan`: Runs static analysis before commit.
  - `codeguard-review`: Runs AI review with Ollama Phi-3.
  - `codeguard-report`: Generates metrics report.
- **language: system** → Executes commands directly in the environment.
- **types: [python]** → Ensures hooks run only on Python files.

This setup blocks commits if **CRITICAL/ERROR issues** are detected, while allowing commits with INFO/WARNING issues (depending on your quality gate configuration).

Outputs:

- **CI/CD:** JSON artifacts for reports.
- **Pre-commit:** Immediate feedback in developer workflow, preventing bad code from being committed.

Example Workflow:

1. Developer commits code → Pre-commit hook runs CodeGuard.
2. If CRITICAL/ERROR issues are found → Commit is blocked.
3. If commit passes → GitLab CI pipeline runs CodeGuard again, generating reports for maintainers.

Module 6: Streamlit Review UI

Purpose: This module provides an **interactive dashboard** for developers to visualize static issues, AI reviews, metrics, and program outputs. It also allows users to apply AI-generated fixes directly within the interface, making the review process more intuitive and developer-friendly.

Important Functions & Blocks:

- `run_code(file_path, language)` Executes uploaded code depending on the language (Python, JavaScript, Java, C/C++). Captures program output and errors for display.

- `detect_language(filename)` Identifies the programming language based on file extension, used to tag files and route them correctly.
- `auto_apply_fixes(file_path, issues, ai_results)` Uses Ollama Phi-3 to generate corrected code for detected issues. Applies fixes directly to the file and formats Python code with Black.
- `generate_ai_review_batched(static_results, use_llm=True)` Runs AI review in batch mode, combining multiple issues into a single prompt for efficiency. Produces explanations and suggested fixes per file.
- **Main Run Block (`if run_btn:`)** Handles uploaded or pasted code, runs static analysis (Module 1), AI review (Module 2), metrics (Module 3), and program execution. Stores results in `st.session_state` for use across tabs.
- **Tabs (`st.tabs`)**
    - **Static Issues Tab:** Displays issues detected by static analysis with severity badges.
    - **AI Review Tab:** Shows Ollama Phi-3 explanations, suggested fixes, diffs, and feedback buttons.
    - **Metrics Tab:** Visualizes quality scores, maintainability index, complexity, and issue counts with Plotly charts.
    - **Program Output Tab:** Displays stdout and stderr from executed code.
    - **Apply Fixes Tab:** Allows users to apply AI fixes interactively and view diffs before/after.
- **Export Reports (Sidebar)** Provides download buttons for JSON reports (static issues, AI review, metrics) and a unified ZIP export.

Performance & Results

- Parser accuracy ≥95%
- Docstring coverage ≥90%
- Pre-commit hook speed <5s
- End-to-end execution <3 minutes for medium projects (~2k functions)
- Developer acceptance rate ≥80% for generated docstrings
- Streamlit UI outputs: Static Issues, AI Review, Metrics, Program Output, Error vs Clean comparison.

Conclusion

CodeGuard is a complete ecosystem that unifies static analysis, AI-powered review, metrics, CLI commands, CI/CD integration, and an interactive Streamlit UI into one platform. By combining rule-based checks with Ollama Phi-3's natural-language feedback and auto-fixes, it makes code reviews faster, clearer, and more reliable. Each module contributes to a seamless workflow: detecting issues, explaining them in human-friendly language, quantifying quality, automating checks in pipelines, and providing an intuitive dashboard for visualization and fixes. Together, these components empower developers to maintain high code quality, improve productivity, and ensure consistency across projects, demonstrating how AI can transform the code review process into a more efficient and developer-friendly experience.