Rapid Code Quality Analyzer (Rcqa) A Lightweight Multi-Language Static Analysis Platform for Academic and Small-Scale Development Environments

Vijay V[1], Mrs. K. Krishna Veni [2], Ms. Divya. S[3]
[1]*III Sem Student, Department of Computer Science and Applications S.E.A. College of Science, Commerce and Arts, Bangalore*
[2]*Assistant Professor, Department of Computer Science and Applications S.E.A. College of Science, Commerce and Arts, Bangalore*
[3]*VI Sem Student, CSE S.E.A. College of Engineering and Technology*

***Abstract*—Code quality assurance is a fundamental requirement in modern software engineering to ensure maintainability, security, and reliability. Traditional manual review methods are time-consuming and inconsistent, while enterprise-grade static analysis tools such as SonarQube impose infrastructure complexity and licensing costs. This paper presents the design and implementation of the Rapid Code Quality Analyzer (RCQA), a lightweight, web-based static analysis system designed for academic institutions and small development teams.**

RCQA supports six programming languages—Python, Java, JavaScript, TypeScript, C, and C++—using Abstract Syntax Tree (AST) analysis for Python and regex-based heuristic analysis for other languages. The system evaluates source code using metrics including Cyclomatic Complexity, Cognitive Complexity, Halstead Effort, Maintainability Index, Lines of Code, and security vulnerability detection. A weighted scoring model produces a unified quality score (0–100), classifying code from Enterprise Ready to High Risk.

The platform is implemented using FastAPI, SQLite, SQLAlchemy, JWT authentication, bcrypt hashing, and a responsive HTML/CSS/JavaScript frontend with Chart.js visualization. Experimental evaluation demonstrates accurate detection of complexity and security issues with efficient performance for typical academic-scale codebases. The study concludes that effective multi-language static analysis can be delivered using open-source technologies without enterprise overhead.

***Index Terms*—Static Code Analysis, Software Quality Metrics, Cyclomatic Complexity, Maintainability Index, FastAPI, Secure Coding, Multi-language Analysis.**

I. INTRODUCTION

As software systems grow in scale and complexity, ensuring high code quality becomes increasingly challenging. Poor code quality leads to maintainability issues, security vulnerabilities, technical debt accumulation, and increased lifecycle costs. Traditional manual code reviews, while valuable, suffer from subjectivity, inconsistency, and scalability limitations.

Enterprise tools such as SonarQube and Coverity provide comprehensive analysis but require complex server infrastructure, configuration overhead, and often involve paid licensing models. This creates accessibility barriers for academic institutions and small development teams.

To address this gap, the Rapid Code Quality Analyzer (RCQA) was developed as a lightweight, easily deployable static analysis system that:

- Performs automated static code analysis without execution
- Supports multiple programming languages
- Computes standardized quality metrics
- Detects common security vulnerabilities
- Generates unified quality scores
- Provides downloadable PDF reports
- Includes an analytics dashboard for trend monitoring

The primary objective of RCQA is to deliver industry-inspired quality analysis capabilities in an accessible academic-friendly architecture.

II. LITERATURE REVIEW

Static code analysis has long been a research focus in software engineering. McCabe (1976) introduced Cyclomatic Complexity as a quantitative measure of control flow complexity. Halstead (1977) proposed software science metrics based on operators and operands. Oman and Hagemeister (1992) formalized the Maintainability Index (MI), combining complexity and size metrics.

Modern tools such as SonarQube integrate these metrics with rule-based vulnerability detection. However, existing tools show the following limitations:

Tool	Limitation
SonarQube	Requires Java server setup, complex configuration
ESLint / Pylint	Language-specific, no unified scoring
Code Climate	Subscription-based cloud service
Coverity	High-cost commercial solution
Manual Review	Subjective and non-quantitative

Research indicates that lightweight static analysis systems can significantly improve early defect detection while minimizing integration overhead. RCQA builds upon established metric theory while focusing on usability and deployment simplicity.

III. SYSTEM ARCHITECTURE

RCQA follows a layered architecture:

3.1 Architectural Layers

1. Presentation Layer
   - HTML5, CSS3, JavaScript
   - Chart.js dashboards
   - Responsive UI
2. API Layer
   - FastAPI REST endpoints
   - JWT-based authentication
   - Role-based access control
3. Analysis Engine
   - Python AST parsing (deep analysis)
   - Regex-based multi-language detection
   - Security vulnerability scanning
   - Code smell detection
   - Duplicate block detection
4. Scoring Engine
   - Weighted formula combining:
     - Cyclomatic Complexity
     - Cognitive Complexity
     - Halstead Effort
     - Maintainability Index
     - Security findings
5. Persistence Layer
   - SQLite database
   - SQLAlchemy ORM
   - User and report entities
6. Reporting Module
   - PDF generation using ReportLab

IV. METHODOLOGY

4.1 Static Analysis Strategy

RCQA performs static analysis without executing uploaded code, ensuring security.

- Python files are parsed using the built-in AST module.
- Other languages use control-flow keyword heuristics.
- Security vulnerabilities are detected via regex patterns.
- Code duplication uses block hashing.
- Long methods and deep nesting are flagged.

4.2 Quality Metrics

Cyclomatic Complexity (CC)

Measures the number of independent paths:

$$CC = E - N + 2P$$

Halstead Effort (E)

$$Volume = (N_1 + N_2) \cdot \log_2(n_1 + n_2)$$
$$Effort = Difficulty \cdot Volume$$

Maintainability Index (MI)

$$MI = 171 - 5.2\ln(HV) - 0.23(CC) - 16.2\ln(LOC)$$

Normalized to 0–100.

4.3 Weighted Scoring Model

Final Quality Score:

$$Score = w_1(MI) + w_2(Security) + w_3(Complexity) + w_4(CodeSmells) + w_5(Duplication)$$

Classification:

Score Range	Classification
85–100	Enterprise Ready
70–84	Production Ready
50–69	Needs Refactoring
<50	High Risk

V. IMPLEMENTATION

5.1 Technology Stack

Technology	Purpose
Python 3.11	Backend
FastAPI	REST API
SQLite	Database
SQLAlchemy	ORM
JWT (python-jose)	Authentication
bcrypt	Password hashing
ReportLab	PDF generation
Chart.js	Dashboard visualization

5.2 Modules

- Static Analyzer
- AI Suggestion Engine
- Scoring Engine
- Report Generator
- Database Layer
- Authentication Module
- File Validator
- REST API Endpoints
- Frontend UI

VI. RESULTS AND PERFORMANCE

6.1 Functional Outcomes

- Supports 6 languages
- Detects 15 security patterns
- Generates detailed PDF reports
- Provides analytics dashboard
- Maintains persistent analysis history

6.2 Performance

- Analysis time: <2 seconds for typical academic files (<1000 LOC)
- Low memory footprint
- No external server dependencies

VII. DISCUSSION

RCQA successfully demonstrates that:

- Effective static analysis can be implemented using open-source tools.
- AST-based analysis improves metric precision for Python.
- Weighted scoring simplifies quality interpretation.
- Regex-based heuristics are sufficient for baseline multi-language support.

However, limitations include:

- Heuristic-based non-Python analysis
- No real-time IDE integration
- No CI/CD pipeline hooks
- Limited large-scale scalability compared to enterprise systems

VIII. CONCLUSION

This research presents RCQA as a practical, lightweight alternative to enterprise static analysis tools. The system integrates established software metrics, security detection, AI-assisted suggestions, and dashboard analytics within a deployable academic-scale architecture.

The project validates that multi-language static analysis, secure authentication, and automated reporting can be implemented using entirely open-source technologies.

RCQA bridges the gap between manual code review and enterprise-grade tools, making structured code quality assessment accessible to students and small teams.

IX. FUTURE WORK

Future enhancements may include:

- GitHub repository integration

- Real-time editor plugin support
- Machine-learning-based vulnerability prediction
- CI/CD integration
- Docker-based deployment packaging
- Support for additional languages (Go, Rust)

REFERENCES

[1] McCabe, T. J. (1976). A Complexity Measure. IEEE Transactions on Software Engineering.
[2] Halstead, M. H. (1977). Elements of Software Science.
[3] Oman, P., & Hagemeister, J. (1992). Metrics for Assessing a Software System's Maintainability.
[4] FastAPI Documentation.
[5] SQLAlchemy Documentation.
[6] OWASP Top 10 Security Risks.