Code Analysis and Automated Feedback Systems

What is Code Analysis?

Code analysis refers to the systematic process of examining source code to identify errors, inefficiencies, or violations of coding standards **without necessarily executing the code**. It helps developers maintain clean, secure, and optimized codebases.
There are two main forms:

- **Static Analysis:** Inspects code before execution (compile-time).
- **Dynamic Analysis:** Observes code behavior during execution (run-time).

Together, these analyses form the foundation of **automated evaluation systems** that assess programming quality and provide feedback to developers or students.

Purpose of a Code Analysis and Automated Feedback System

The main objectives are:

1. **Error Detection:** Identify syntactic and semantic mistakes early in development.
2. **Code Quality Improvement:** Ensure adherence to coding standards and best practices.
3. **Performance Optimization:** Highlight inefficient constructs affecting time or space complexity along with a concise review of other criteria that can affect code performance.
4. **Educational Feedback:** In academic settings, help students understand mistakes and suggest better approaches.
5. **Learning enhancement** — helping users understand *why* their code performs poorly and how to make it better by providing an optimal solution from the predefined knowledge base.

Key terminologies

- **Syntax checking:** The most basic level of code verification that ensures source code follows language grammar rules — for example, missing colons, incorrect indentation, or unmatched brackets in Python.
- **Static Analysis:** Analyzes the program without executing it, focusing on structure, semantics, and style.
  Types of static analysis include:
  - **Code Style Analysis:** Checks adherence to style guides (e.g., PEP8). Examples of rules enforced through code style analysis include consistent indentation, naming conventions for variables and functions, and line length limits
  - **Code Quality Analysis:** Evaluates maintainability, duplication, and clarity.
  - **Security Vulnerability Detection:** Finds potential exploits (e.g., unsafe eval()).
  - **Memory Leak Detection:** Ensures proper allocation and release of resources.

- **Compliance and Standard Checking:** Ensures the code follows project or industry standards.
- **Performance and Optimization Analysis:** Identifies loops or algorithms that can be optimized.
- **Dependency Analysis:** Detects unnecessary or insecure imports.
- **Documentation and Comment Analysis:** Checks for missing or outdated docstrings.

Many of these analyses are lightweight enough to run during **Continuous Integration (CI)**, improving the development–bug detection–fix cycle.

- **Dynamic analysis:** Examines the behavior of code **during execution**, typically used to measure runtime efficiency, detect memory leaks, or test edge cases through automated test suites.
- **Feedback generation:** The process of interpreting analysis results and transforming them into understandable insights for users — e.g., “Your function has a high cyclomatic complexity; try simplifying nested loops.”
- **Code smells:** Patterns in code that indicate deeper structural issues (e.g., long methods, duplicate code, or large classes). While not necessarily bugs, they often lead to maintainability or performance problems.

Concepts

Concept	Explanation
Cyclomatic Complexity	A measure of the number of independent paths in the code; high values indicate complex, harder-to-test code.
Code Quality Metrics	Quantitative measures such as maintainability index, duplication ratio, and test coverage.
Readability	Subjective ease of understanding code; influenced by naming conventions, indentation, and documentation.
Best Practices	Accepted coding patterns that enhance maintainability and performance.
LOC (Lines of Code)	A basic metric to assess program length and sometimes complexity.
Time Complexity	Estimates algorithm performance relative to input size (e.g., O(n), O(log n)).
Space Complexity	Evaluates how much memory a program consumes as input grows.
PEP8 Guidelines	Python’s official style guide defining standards for indentation, naming, imports, and layout.
Abstract Syntax Tree (AST)	A structured tree representation of code used for parsing and analysis — key to static analysis tools.

Challenges in the domain

- **Measuring Code Readability:**
  Readability is subjective and influenced by experience, naming style, and formatting, making it hard to quantify.
- **Context-Based Evaluation:**
  Automated tools struggle to adapt scoring to the *intent* of a program — e.g., multiple correct implementations of the same logic.

- **Balancing Strictness and Learning:**
  Excessive penalization may discourage beginners; the system must explain *why* code is suboptimal rather than just scoring low.
- **Integration with Knowledge Bases:**
  Building and maintaining a reference library of optimal code requires consistent structure and normalization for comparison.
- **Student-Friendly Feedback:**
  The ultimate goal is to design a system that not only grades code but acts as a **learning guide** — explaining efficiency, style, and optimization possibilities.

Literature Survey

1. Evaluating Python Static Code Analysis Tools Using FAIR Principles

Author: Hassan Bapeer Hassan, Qusay Idrees Sarhan, Árpad Beszedes
Publication: IEEE, 2024
Summary: This paper assesses various Python static analysis tools against FAIR principles (Findable, Accessible, Interoperable, Reusable). It benchmarks tools like Pylint, Flake8, and Bandit, highlighting their strengths and weaknesses in maintainability, transparency, and data availability.
Takeaways: Helps identify reliable static analysis tools for academic or industrial projects; emphasizes reproducibility and data accessibility in tool evaluation. Highlights major areas of importance in terms of code analysis like Compliance and Standard checking, Performance Optimization Analysis, Memory Leak Detection, etc. Also provides a list of top scoring tools for static analysis along with their advantages which helped with selecting the tools for the project.

2. Automated Assessment System for Programming Courses: A Case Study for Teaching Data Structures and Algorithms

Author: Andre L. C. Barczak, Anuradha Mathrani, Binglan Han, Napoleon H. Reyes
Publication: Springer, 2023
Summary: The paper presents an automated assessment framework integrated into programming education to evaluate students' algorithmic problem-solving. It emphasizes automatic grading, code analysis, and instant feedback to enhance learning outcomes.
Takeaways: Demonstrates how automated feedback systems can improve students' understanding; relevant for feedback-based analysis in our project. Utilizes the concept of model solution similar to our knowledge base along with test cases to evaluate coding assignments on data structure.

3. Survey on Static Analysis Tools of Python Programs

Author: Zoltán Porkoláb, Hristina Gulabovska
Publication: CEUR Workshop Proceedings, 2020
Summary: Provides a comprehensive overview of static analysis tools for Python including Pylint, Pyflakes, and Mypy. Discusses their coverage, limitations, and evolving roles in code quality assurance.
Takeaways: Useful for identifying features and limitations of popular tools when designing a new static analysis framework.

4. Franc: A Lightweight Framework for High-Quality Code Generation

Author: Mohammed Latif Siddiq, Beatrice Casey, Joanna C. S. Santos
Publication: IEEE, 2024
Summary: Franc introduces a modular framework for automated high-quality code generation using syntactic and semantic consistency checks. It applies static analysis concepts to ensure error-free output.
Takeaways: Highlights how lightweight frameworks can combine static analysis with intelligent generation; applicable to integrating rule-based checks. Can be useful for future implementations of the project which would include code generation.

5. Unambiguity of Python Language Elements for Static Analysis

Author: Bence Nagy, Tibor Brunner, Zoltán Porkoláb
Publication: IEEE, 2021
Summary: Investigates Python's syntax and semantics to identify ambiguities affecting static analysis. Suggests parsing strategies to handle dynamic features such as duck typing and runtime imports.
Takeaways: Improves understanding of Python's structural challenges, helping refine AST-based parsing in our tool since Abstract Syntax Tree (AST) forms the basis for static analysis by representing the code into a tree like structure for easier parsing.

6. Towards More Sophisticated Static Analysis Methods of Python Programs

Author: H. Gulabovska, Z. Porkoláb
Publication: IEEE, 2019
Summary: Explores novel static analysis techniques tailored for Python's flexibility. The paper introduces modular rule-based checking and hybrid static-dynamic methods.
Takeaways: Shows potential for combining multiple analysis dimensions (syntax, complexity, performance) in educational tools to make them more robust and wide spanning in terms of analysis criteria.

7. ExcePy: A Python Benchmark for Bugs with Python Built-in Types

Author: X. Zhang, R. Yan, J. Yan, B. Cui, J. Yan and J. Zhang
Publication: IEEE, 2022
Summary: ExcePy provides a benchmark dataset of Python bugs involving built-in types to test static analysis tools' accuracy and detection power.
Takeaways: Offers insights into designing datasets or benchmarks for evaluating static analysis performance.

8. Large Language Models (GPT) for Automating Feedback on Programming Assignments

Author: Maciej Pankiewicz, Ryan S. Baker
Publication: ICCE 2023
Summary: Examines the use of GPT models for generating automated, human-like programming feedback. Evaluates the accuracy, tone, and relevance of AI-generated feedback.
Takeaways: Demonstrates AI's role in complementing rule-based analysis for context-aware Feedback and how good prompt engineering can help make more transparent GPT based projects that can better assist pupils with their learning.

9. Automated Assessment in Programming Courses: A Case Study during the COVID-19 Era

Author: Barra E, López-Pernas S, Alonso Á, Sánchez-Rada JF, Gordillo A, Quemada J.
Publication: MDPI, 2020
Summary: Discusses the implementation of automated coding assessment systems during remote
learning, focusing on reliability, fairness, and usability.
Takeaways: Reinforces the importance of automated evaluation for scalable, accessible learning environments. Making an evaluation tool that can run locally in the user's computer without requiring internet access. All the content along with the scores can be later uploaded to an LMS.

10. Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation

Author: Tung Phung, Victor-Alexandru Pădurean, Anjali Singh, Christopher Brooks, José Cambronero, Sumit Gulwani, Adish Singla, Gustavo Soares
Publication: arXiv, 2023
Summary: Proposes a dual-model framework for generating and validating feedback similar to human tutors using LLMs like GPT-3.5 and GPT-4.

Takeaways: Using LLMs for generating human-like feedback: relevant for designing explainable feedback in our system.

11. Enhancing Programming Education with ChatGPT: A Case Study on Student Perceptions and Interactions in a Python Course

Author: Boxaun Ma, Li Chen, Shin'ichi Konomi
Publication: arXiv, 2024
Summary: Analyzes how students perceive ChatGPT as an educational assistant for learning Python. Finds improvement in engagement but notes over-reliance risks.
Takeaways: Highlights the educational role of AI assistants and the need for balance between automation and user control. Implementing AI and LLMs to evaluate and analyze coding programs and to generate feedbacks on the basis of which users can rectify their codes removing the risk go over-reliance.

12. Analysis of the Tools for Static Code Analysis

Author: D. Nikolić, D. Stefanović, D. Dakić, S. Sladojević and S. Ristić
Publication: IEEE Xplore, 2021
Summary: Performs a comparative analysis of multiple static code analysis tools, evaluating them on detection capabilities (bugs, code smells), ease of use, configurability, and integration with development workflows. It examines their detection coverage across different categories of issues and discusses how tool outputs can be interpreted by developers. The study highlights differences in rule sets, false positive rates, and recommended configurations.
Takeaways: Use a **combination** of analyzers (style + security + complexity) rather than relying on a single tool. Provide clear, student-friendly explanations in feedback so learners aren't overwhelmed.

13. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software

Author: M. Beller, R. Bholanath, S. McIntosh and A. Zaidman
Publication: IEEE Xplore 2016
Summary: This large-scale empirical study runs static analysis tools across a broad corpus of open-source projects to measure real-world effectiveness. It reports on common patterns of issues detected, the prevalence of true vs false positives, and how tool performance varies by project characteristics (language, size, domain). The paper emphasizes that default configurations often yield noisy results and that precision/recall greatly depend on tuning and project context.
Takeaways: To expect variance in analyzer performance depending on code style and project size. Incorporate statistics (e.g., frequency of issue types) into feedback so students see how common a problem is. Including guidelines for how the analyzers are configured when reporting results (transparency helps reproducibility).

14. The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction

Author: F. Wedyan, D. Alrmuny and J. M. Bieman
Publication: IEEE Xplore
Summary: This empirical work evaluates how well automated static analysis tools detect real faults and predict where refactoring is needed. The paper compares tool output against ground-truth bugs and refactorings in established datasets and reports precision/recall metrics for different issue categories. It often finds that static tools are useful for detecting certain classes of defects (stylistic, API misuse) but less effective for deeper logical faults.
Takeaways: Static analysis is complementary, not a replacement for reasoning about correctness; emphasize this in feedback. Using static findings to suggest refactorings (e.g., reduce cyclomatic complexity), but avoid claiming certainty for semantic correctness. When scoring, weight issues according to tool reliability for that issue class

15. Hints-In-Browser: Benchmarking Language Models for Programming Feedback Generation

Author: Nachiket Kotalwar, Alkis Gotovos, A. Singla
Publication: NeurIPS (formerly NIPS)
Summary: This paper benchmarks multiple language models on the task of generating programming hints in a browser-based learning environment. It evaluates hint relevance, usefulness, and potential for hallucination, comparing model outputs to human tutor hints. The experiments assess how well LLMs can produce stepwise, actionable hints and measure differences across prompts and model sizes.
Takeaways: LLMs can produce useful hints, but quality varies — include a validation layer (automated or human-in-the-loop) before surfacing hints. Use structured prompt templates to reduce hallucination (and present model provenance/uncertainty in UI). Combine deterministic static-analysis suggestions with LLM-generated explanations to get both precision and pedagogical richness.