LLM Code Audit: A Multi-Model Evaluation System for AI-Generated Code Quality

Sai Kowshik Allu
IIT Tirupati
India
cs22b004@iittp.ac.in

Gosu Jaswanth
IIT Tirupati
India
cs22b020@iittp.ac.in

Gottimukkula Nishchith
IIT Tirupati
India
nishchithb22@iittp.ac.in

Guru Rohith G
IIT Tirupati
India
cs22b022@iittp.ac.in

I Kalyan Anudeep
IIT Tirupati
India
cs22b025@iittp.ac.in

Rashmitha Veeramsetti
IIT Tirupati
India
cs22b050@iittp.ac.in

Sridhar Chimalakonda
IIT Tirupati
India
sridhar@iittp.ac.in

ABSTRACT

This project introduces a **web-based application** that enables users to input **natural language prompts** and receive corresponding code outputs generated by various **large language models (LLMs)**, such as *OpenAI's GPT*, *Meta's Code LLaMA*, and *Google's Gemini*. The system is designed to not only compare the **functional accuracy** of these model-generated outputs but also to critically assess their underlying **software quality** using a comprehensive set of **automated code evaluation metrics**. Among these metrics, particular emphasis is placed on the identification and quantification of **technical debt**—a software engineering concept that captures the *long-term cost* associated with writing code that is functional but not optimal in terms of *quality, readability, or maintainability*.

The platform offers an **intuitive interface** for users to **generate, view, and compare** code outputs from multiple LLMs in parallel. It integrates a suite of **static code analysis tools** that automatically evaluate various software quality attributes, including *cyclomatic complexity, code duplication, code smells, function length*, and *documentation coverage*. The results are visualized through **comparative dashboards** that highlight both strengths and potential issues in each model's output. This **comparative analysis** allows users to explore not only how well the models perform a given task, but also how *responsibly and efficiently* they generate code from a software engineering standpoint.

In addition to **real-time evaluation**, the platform logs **historical performance data** and supports **exportable reports** for further offline analysis or integration into larger **CI/CD pipelines**. It is also **extensible**, allowing future integration of additional metrics, new LLMs, or custom prompt templates, making it a **flexible tool** for both *research and practical software development* use.

By combining the power of **AI-driven code generation** with **rigorous software quality assessment**, this project bridges a crucial gap in current LLM tooling. It addresses the growing concern around **blindly trusting AI-generated code** by providing a **structured, metric-driven approach** to validate code quality before integration. This contributes to the ongoing evolution of **AI-assisted programming** by promoting not only *speed and automation* but also *accountability, sustainability*, and **engineering discipline** in the use of generative models.

Conference'17, July 2017, Washington, DC, USA
© 2025 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; • **Computing methodologies** → *Artificial intelligence*.

KEYWORDS

LLM code evaluation, AI-generated software, maintainability metrics, static analysis, code audit, prompt sensitivity

ACM Reference Format:
Sai Kowshik Allu, Gosu Jaswanth, Gottimukkula Nishchith, Guru Rohith G, I Kalyan Anudeep, Rashmitha Veeramsetti, and Sridhar Chimalakonda. 2025. LLM Code Audit: A Multi-Model Evaluation System for AI-Generated Code Quality. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, the emergence of **large language models (LLMs)** has significantly influenced the software development lifecycle. Models like *OpenAI's Codex*, *Meta's Code LLaMA*, and *Google's Gemini* are capable of translating human-readable prompts into functional code snippets across a variety of programming languages. These tools have the potential to **accelerate development workflows**, reduce the barrier to entry for **novice programmers**, and assist experienced developers in **repetitive or boilerplate tasks**.

As a result, LLMs are becoming embedded in **integrated development environments (IDEs)**, **coding assistants**, and **educational platforms**.

However, as LLMs become more accessible and influential in software engineering, a growing concern has emerged: while these models can produce code that is syntactically and semantically correct, they do not always generate code that adheres to principles of **clean architecture**, **efficient design**, or **maintainable structure**. Code that merely "works" may still suffer from poor **readability**, inefficient logic, **duplicated patterns**, or lack of **documentation**—all of which contribute to what is known in software engineering as *technical debt*.

Technical debt refers to the future cost of reworking code that is implemented quickly or carelessly, often trading off long-term maintainability for short-term gain. When left unchecked, technical debt can slow down development, introduce bugs, and increase onboarding time for new developers. With the growing use of LLMs in critical development pipelines, it becomes increasingly important to not only assess the **functionality** of AI-generated code but also to evaluate its **quality** and **sustainability**.

To address this need, this project proposes a novel solution: a **web-based platform** that combines code generation from multiple LLMs with an automated code quality evaluation system. The platform allows users to input prompts and receive generated code from various LLMs in real time. More importantly, it goes a step further by analyzing the generated code through a suite of **static analysis tools** that measure key software metrics—such as *cyclomatic complexity, code duplication, commenting ratio,* and adherence to *design patterns.*

One of the distinguishing features of this platform is its emphasis on comparing **technical debt** across different models for the same prompt. This allows users to explore how each model handles a task not just in terms of functionality, but in terms of **engineering excellence**. The interface presents this analysis in a **structured format**, providing insights into which models produce more **maintainable code** and where potential risks may lie.

This project is motivated by the growing reliance on AI in software creation and the corresponding need to ensure that AI-generated code meets **industry standards**. It contributes to both **academic research** and **practical development** by offering a tool that enables **empirical analysis**, **educational insight**, and **real-time feedback** on the quality of LLM-generated code. It also encourages developers to remain critical and aware of the limitations of AI assistance, promoting responsible and informed use of these powerful technologies.

2 RELATED WORK

Traditional static analysis tools like SonarQube, PMD, and ESLint focus on syntax correctness, code smells, and structural issues but are not tailored for evaluating LLM-generated code. Research on AI code evaluation is still nascent, with limited exploration into prompt sensitivity, inference variance, and duplication metrics. Our work builds on these foundations by integrating standard tools with new metrics specific to AI-generated outputs.

3 DESIGN AND DEVELOPMENT

3.1 Project Overview

LLM Code Audit is an AI-powered web application that enables developers to **compare and analyze code** generated by various **Large Language Models (LLMs)**—including ChatGPT, DeepSeek, Gemini, Mistral, and LLaMA—as well as manually entered code.

The core objective of the tool is to automate code quality assessment using a suite of custom-defined metrics:

- **ACI** — AI Complexity Index
- **ARS** — AI Readability Score
- **ADR** — AI Dependency Risk
- **AMR** — AI Maintainability Risk
- **ARF** — AI Redundancy Factor

This project addresses a major pain point faced by developers: *how to objectively choose the most maintainable and efficient AI-generated code* rather than relying solely on correctness.

3.2 Technology Stack

Layer	Technology Used
Frontend	React.js, Tailwind CSS
Backend	Firebase (Authentication, Firestore)
APIs	SonarQube Web API
Code Metrics	Custom Metrics: ACI, ARS, AMR, ADR, ARF
LLMs Integrated	ChatGPT, DeepSeek, Gemini, Mistral, LLaMA
Hosting	Firebase Hosting (Optional)

Table 1: Tech Stack Overview

3.3 Folder Structure

The project follows a clean, modular structure to improve maintainability and scalability:

```
/src

 /pages
   HomePage.jsx       # Prompt input, model selection
    Dashboard.jsx        # Display code and metrics
    HistoryPage.jsx      # Past prompt/code history
    AboutPage.jsx        # Project documentation
    AuthPage.jsx         # Firebase login/signup
   ComparisonDetail.jsx  # In-depth model/code analysis

 /components         # Reusable components (charts, tables)
 /styles                # Tailwind and custom CSS
 App.jsx                # Main routing logic
```

3.4 Key Features and Workflow

1. User Authentication

- Secure sign-in/sign-up via Firebase Authentication
- User sessions for personalized history tracking

2. Prompt Input and Model Selection

- Users enter a prompt in a text editor

- Choose LLMs for code generation (e.g., ChatGPT, Gemini, etc.)
- Optional: paste manual code for comparison

3. Code Generation

- Each model returns code for the same prompt
- Results shown with model labels and syntax highlighting

4. Code Analysis Using AI Metrics

- **ACI:** Measures cognitive complexity and nesting
- **ARS:** Readability score based on comments and formatting
- **ADR:** Risk from dependencies and imports
- **AMR:** Maintainability based on smells and documentation
- **ARF:** Checks for duplication and redundancy

5. Results Visualization

- Interactive bar charts for each metric across models
- Summary panel ranks best code output

6. History Page

- Logs all previous prompts and analysis
- Includes timestamps, results, and best-ranked model

7. About Page

- Describes project goals
- Details flaws in raw AI code (e.g., no comments, redundancy)
- Explains metric formulas and methodology

3.5 AI Metrics Formulas

The following equations define the custom AI metrics used for evaluating code quality. Each metric is calculated using static code analysis (via SonarQube) and custom logic modules.

1. AI Readability Score (ARS):

$$\begin{aligned}\text{ARS} = {} & (0.4 \times \text{Avg. Line Length}) \\ & + (0.4 \times \text{\%Lines With Comments}) \\ & + (0.2 \times \text{\%Consistent Variable Naming})\end{aligned} \tag{1}$$

2. AI Dependency Risk (ADR):

$$\begin{aligned}\text{ADR} = {} & (0.5 \times \text{Avg. Dependencies per Module}) \\ & + (0.5 \times \text{Circular Dependencies Count})\end{aligned} \tag{2}$$

3. AI Redundancy Factor (ARF):

$$\begin{aligned}\text{ARF} = {} & (0.6 \times \text{\%Duplicated Code}) \\ & + (0.4 \times \text{\%Similar Code in Other Modules})\end{aligned} \tag{3}$$

4. AI Complexity Index (ACI):

$$\begin{aligned}\text{ACI} = {} & (0.5 \times \text{Cognitive Complexity}) \\ & + (0.3 \times \text{Method Length}) + (0.2 \times \text{Nesting Level})\end{aligned} \tag{4}$$

5. AI Maintainability Risk (AMR):

$$\begin{aligned}\text{AMR} = {} & (0.4 \times \text{Code Smells per 100 LOC}) \\ & + (0.4 \times \text{Duplication Density \%}) \\ & + (0.2 \times \text{\%Methods Without Comments})\end{aligned} \tag{5}$$

Each metric outputs a normalized score where a **lower value** indicates **higher code quality and maintainability**.

.

4 USER SCENARIOS

4.1 Scenario 1: First-Time User Login and Code Prompt

- The user signs up or logs in using **Firebase Authentication**.
- Upon successful login, they are redirected to the **HomePage**.
- The user enters a code prompt describing the required functionality.
- The user selects one or more **LLMs** (e.g., ChatGPT, Gemini, DeepSeek) and may also enable **manual code input**.
- Clicking the **"Compare Models"** button navigates them to the **Dashboard**.

4.2 Scenario 2: Generating and Comparing Code

- Each selected LLM generates its own version of code for the given prompt.
- The generated code is displayed in a clean interface with **labels indicating the source model**.
- Users can **edit the prompt**, **reselect models**, or **add manual code** directly from the Dashboard.

4.3 Scenario 3: Analyzing Code Quality

- After reviewing the generated code, the user clicks on **"Analyze Code"**.
- The system computes all five custom metrics: **ACI, ARS, ADR, AMR, and ARF**.
- The results are visualized using **bar charts** comparing each model's scores.
- The app suggests the best code snippet based on **lowest technical debt and highest maintainability**.

4.4 Scenario 4: Viewing History

- The user clicks the **History** tab in the navigation menu.
- All previously used prompts are shown in a list with associated **timestamps**.
- Clicking on a past prompt reveals the complete analysis: code from all models, metric scores, and summary.
- This feature allows users to **review and learn from previous comparisons**.

4.5 Scenario 5: Understanding the Project

- The **About** page provides an overview of:
  - Common flaws in LLM-generated code (e.g., redundancy, lack of comments).
  - The importance of **comparing multiple LLM outputs**.
  - Definitions and formulas for all five custom AI metrics.
  - A **step-by-step guide** to using the tool.

4.6 Scenario 6: Profile and Logout

- The top-right corner of the application includes a dropdown with **user profile** and **Logout** options.
- The user can log out securely from any page.
- All history is tied to the user's Firebase account and **remains accessible on re-login**.

6. DISCUSSION

LLM Code Audit represents a pioneering step in the ongoing effort to rigorously evaluate AI-generated code using the lens of **software engineering principles**—particularly **quality**, **maintainability**, and **readability**. As **Large Language Models (LLMs)** become increasingly capable of producing functional code, the need to assess that code beyond simple correctness becomes critical. Our framework addresses this gap by introducing a structured, comparative, and metric-driven analysis pipeline. However, the effectiveness of the system must be considered in light of its scope, current limitations, and opportunities for future growth.

At its foundation, the system performs **static code analysis**, focusing on structural, syntactic, and logical aspects of the code. While these analyses are crucial for assessing immediate code quality, they inherently exclude runtime behavior, which is equally important in real-world software development. Parameters such as execution speed, memory consumption, and CPU efficiency remain unexplored in our current design. These runtime metrics would provide essential insight into how performant and scalable the generated code would be in practical applications. As LLM-generated code begins to enter production environments, expanding the framework to incorporate dynamic analysis or profiling tools (e.g., Valgrind, Perf, JMH) will be vital.

Another prominent challenge lies in the **prompt engineering paradox**. The quality of LLM-generated code is directly tied to the quality and structure of the prompt. Subtle variations in wording, specificity, or programming context can result in drastically different code outputs. This stochastic behavior is compounded by the probabilistic nature of LLMs. Although our system mitigates this by enforcing prompt normalization—standardizing language, formatting, and problem scope—the influence of LLM internal randomness persists. To improve transparency, we provide users with a prompt history tracking interface, allowing them to experiment iteratively and visually correlate prompt edits with corresponding metric changes. In future iterations, we plan to implement prompt embeddings and similarity scores to automatically detect and flag variations in intent across prompt versions.

The five proposed metrics—ACI (AI Code Integrity), ARS (AI Readability Score), ADR (AI Duplication Rate), AMR (AI Maintainability Rating), and ARF (AI Robustness Factor)—form the crux of our code evaluation system. Each metric attempts to abstract a core software quality attribute into a quantifiable form:

- **ACI** assesses logic integrity by combining cognitive complexity, nesting depth, and branching patterns.
- **ARS** evaluates the legibility and formatting of code, giving weight to comment usage, naming consistency, and indentation.
- **ADR** captures redundancy by measuring exact or near-exact repeated code blocks—an indicator of missed abstraction opportunities.
- **AMR** reflects maintainability by identifying code smells, unused variables, and method cohesion.
- **ARF** considers robustness through inferred exception handling patterns and testability cues.

While these metrics show promise, they are still heuristic in nature and require empirical validation. Testing across larger corpora of LLM outputs, industry-level codebases, and human-written gold standards would help tune thresholds, balance weights, and ensure alignment with actual developer expectations. Collaboration with professional developers and educators could help further benchmark and refine these scores. Additionally, integrating semantic-aware analysis—such as symbolic execution or AST-level pattern recognition—could improve metric reliability.

From a **UI/UX standpoint**, the platform is deliberately designed to be lightweight, accessible, and developer-friendly. The central dashboard includes an intuitive prompt editor, model selection interface, and real-time code rendering side-by-side. Timestamped history cards provide a snapshot of every query interaction, complete with corresponding metric scores and visual diff comparisons. This enables users to conduct rapid A/B testing between models and quickly iterate on prompt formulations. For ease of navigation, the comparison panel offers color-coded metric deltas and code collapsibility, allowing users to focus on areas of divergence between model outputs.

One of the most critical limitations of the current system is its restricted model set, which presently includes ChatGPT, Gemini, DeepSeek, LLaMA, and Mistral. These models were selected to cover a balanced spectrum of proprietary and open-source architectures, allowing for nuanced head-to-head comparisons. However, many other high-performing models—such as Claude, StarCoder, CodeLLaMA, and Mixtral—remain unsupported due to API cost, token limits, or resource bottlenecks.

Additionally, the system does not currently assess security, fairness, or ethics in code generation. As LLMs are increasingly used to automate backend logic and even infrastructure scripts, the risk of inadvertently introducing unsafe or biased patterns becomes non-negligible. Future versions of LLM Code Audit may include modules to scan for security vulnerabilities (e.g., SQL injection, unsafe API usage), license compliance violations, and bias indicators (e.g., discriminatory hardcoding in logic branches).

In summary, LLM Code Audit provides a novel, structured approach for comparing the software engineering quality of code produced by large language models. It bridges the gap between traditional code linters and AI evaluation tools by applying metric-driven analysis grounded in developer practices. While it marks a substantial advancement in AI evaluation, its continued development will depend on addressing limitations in dynamic profiling, empirical metric validation, security auditing, and model diversity.

7. LIMITATIONS

Despite the strengths and innovative contributions of LLM Code Audit, the system currently faces several limitations that outline critical areas for enhancement. Recognizing these gaps is essential for guiding future iterations and ensuring the platform evolves into a more robust and enterprise-ready solution.

- **No Runtime Evaluation:** One of the most significant limitations is the absence of runtime analysis. While static code analysis can detect structural and syntactic issues, it cannot capture dynamic behaviors such as segmentation faults, infinite loops, exception handling failures, or poor algorithmic efficiency.

- **Constraints of Static Analysis:** Static analysis tools are inherently limited in evaluating aspects like memory access patterns, multithreading behavior, and recursive stack depth—factors critical to evaluating systems-level or concurrent programming.
- **High Sensitivity to Prompt Variability:** The system's effectiveness remains highly dependent on prompt formulation. Even when using standardized prompt templates, different LLMs often interpret the same prompt inconsistently.
- **Absence of Feedback-Driven Refinement:** The current system operates in a closed-loop fashion, with no mechanism for incorporating user feedback to improve the accuracy of code assessments or the usability of the interface.
- **Scalability Bottlenecks in Infrastructure:** The platform is built on Firebase and Firestore, which may face scalability and cost-performance issues under enterprise workloads.
- **Data Privacy and Security Concerns:** The system currently stores submitted code snippets in plaintext, without applying encryption-at-rest, raising potential concerns for proprietary or sensitive code.
- **Unvalidated Evaluation Metrics:** The five proprietary evaluation metrics—ACI, ARS, ADR, AMR, and ARF—remain heuristically defined and lack empirical validation across diverse real-world codebases.
- **UI/UX and Accessibility Gaps:** The interface, while functional for desktop use, currently lacks several modern accessibility and usability features such as dark mode, screen readers, and mobile responsiveness.

8. CONCLUSION

LLM Code Audit marks a significant advancement in the responsible, transparent, and comparative use of large language models (LLMs) within the realm of software development. It establishes a novel paradigm where AI-generated code can be systematically evaluated, contrasted, and iteratively improved—paving the way for higher standards in AI-assisted programming practices.

At its core, the platform brings together three critical capabilities into a unified ecosystem: **LLM-based code generation**, **multi-dimensional code quality analysis**, and **structured model benchmarking**. This integration not only streamlines the developer workflow but also addresses the growing need for trustworthy, explainable outputs from generative AI tools.

Key contributions and innovations include:

- Seamless multi-model integration: Direct access to top-tier language models such as ChatGPT, Gemini, DeepSeek, LLaMA, and Mistral.
- Unified evaluation dashboard: A simple yet powerful interface facilitates real-time code generation, side-by-side comparison, and in-depth metric-driven analysis.
- Introduction of proprietary evaluation metrics: The ACI (AI Code Integrity), ARS (AI Readability Score), ADR (AI Duplication Rate), AMR (AI Maintainability Rating), and ARF (AI Robustness Factor) collectively offer a comprehensive view of code quality.
- Time-synchronized prompt tracking and history logging: Users can analyze the evolution of prompt phrasing, observe its effects on code quality, and iteratively refine their inputs.
- Real-time responsiveness via Firebase: Enabling rapid interactions, secure access, and persistent storage.

Looking forward, LLM Code Audit is positioned not merely as a tooling solution, but as a stepping stone toward institutionalizing AI code evaluation standards. By empowering users to make informed decisions about prompt design, model selection, and software quality trade-offs, the platform supports a more deliberate, data-informed approach to generative programming.

9. FUTURE WORK

The future trajectory for LLM Code Audit is shaped by the need to evolve alongside both AI advancements and the evolving needs of developers. Several key areas of improvement and future work include:

- **Integration of Runtime Analysis:** To complement static evaluations, the system will incorporate dynamic profiling techniques, potentially using tools like Valgrind or JMH for runtime behavior analysis.
- **Cross-Model Performance Benchmarking:** By incorporating a wider range of LLMs, including open-source models and private enterprise solutions, we can offer richer comparative insights and test the robustness of results across architectures.
- **Security and Bias Audits:** Future versions will include scans for security vulnerabilities, such as SQL injection, and checks for bias or discriminatory patterns in code generation.
- **Developer Feedback Incorporation:** Introducing a feedback mechanism to refine code assessment results, improve UI/UX, and implement continual learning within the system.
- **Cloud-based Deployment:** Extending the platform for scalable cloud deployment would ensure better handling of large codebases and integration with industry-standard CI/CD workflows.
- **Mobile and Accessibility Features:** With increasing mobile usage, future versions will prioritize mobile optimization and accessibility features to serve a broader audience.

REFERENCES

(1) B. Vasilescu, A. Serebrenik, and M. G. van den Brand, "You can't control the weather: Code commenting challenges in GitHub projects," in *Proc. Int. Conf. Mining Software Repositories*, 2015.
(2) Microsoft, "Code Quality Metrics Reference," [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values
(3) A. Hindle et al., "Automatic Code Summarization: A Literature Review," *Empirical Software Engineering*, vol. 18, no. 4, 2013.
(4) J. Buse and W. Weimer, "Learning a Metric for Code Readability," *IEEE Trans. Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.
(5) OpenAI, "ChatGPT: Optimizing Language Models for Dialogue," [Online]. Available: https://openai.com/blog/chatgpt/
(6) Google, "Gemini: Multimodal Language Model by Google DeepMind," [Online]. Available: https://deepmind.google/technologies/gemini/
(7) DeepSeek, "DeepSeek Coder Model Card," [Online]. Available: https://huggingface.co/deepseek-ai/deepseek-coder
(8) Meta AI, "LLaMA: Open Foundation and Instruction Models," [Online]. Available: https://ai.meta.com/llama/
(9) Mistral AI, "Mistral Models Overview," [Online]. Available: https://mistral.ai/news/announcing-mistral-7b/