An Empirical Study of LLM-Based Source Code Quality Assessment under ISO/IEC 5055:2021

Daniel Pérez-Morera[1,2], **Enrique Vílchez-Lizano**[1,2], **Keilor Rodriguez-Artavia**[2], **Christian Quesada-López**[2], **Marcelo Jenkins**[2]

[1]Universidad de Costa Rica
Centro de Investigaciones en Tecnologías de Información y Comunicación (CITIC)
- San José - Costa Rica

[2]Universidad de Costa Rica
Posgrado en Computación e Informática,
Escuela de Ciencias de la Computación e Informática
- San José - Costa Rica

{daniel.perezmorera, enrique.vilchezlizano, keilor.rodriguezartavia, cristian.quesadalopez, marcelo.jenkins}@ucr.ac.cr

Abstract. *Nowadays, many software companies are required to meet high quality standards in the development of their applications. Rather than replacing traditional SAST tools based on fixed rules, approaches such as GemCA leverage a Large Language Model (LLM) to perform semantic reasoning over code in relation to the ISO/IEC 5055:2021 standard, providing a complementary decision-support mechanism for early-stage quality assessment. This paper presents GemCA and reports an empirical analysis evaluating its accuracy on a dataset of known code weaknesses. The goal is to examine the framework's ability to apply the criteria defined by the ISO/IEC 5055:2021 standard. Results show that GemCA achieves an average accuracy of 81% across 15 repetitions, with significantly higher performance in C# than PHP. However, accuracy varies substantially across weakness categories ($p = 0.0000$), indicating sensitivity to CWE type and programming language. These findings highlight both the potential and the current limitations of LLM-based analysis for ISO/IEC 5055:2021 compliance.*

1. Introduction

Software plays a vital role in nearly all aspects of modern life–from mobile apps to critical systems–raising growing concerns about software quality[Karg et al. 2009]. A single error in the code can result in financial losses, reputational damage, service outages, or even risks to user safety. Ensuring high-quality software is increasingly recognized as essential for supporting operational continuity.

Historically, two key challenges have hindered software quality measurement. First, there was no widely accepted standard for evaluating code quality [CISQ 2025, ISO 2021], leaving companies uncertain about how to define "quality" during client negotiations. Second, existing standards often relied on external indicators rather than analyzing the source code itself, making practical application during development difficult [ISO 2021]. To address these issues, the Consortium for Information & Software

Quality (CISQ) introduced ISO/IEC 5055:2021, a standard that enables structural source code quality assessment and supports automated analysis tools[ISO 2021, CISQ 2025]. It provides clear minimum quality requirements and explicitly identifies unacceptable weaknesses [CISQ 2025].

While tools like SonarQube Enterprise, CAST, and Bitegarden support this standard, they are commercial and may lack full coverage of the CWE weaknesses referenced in ISO/IEC 5055:2021. In parallel, the emergence of Large Language Models (LLMs) has created new opportunities for automated source-code reasoning. LLMs offer natural-language explainability, flexible integration into DevOps pipelines, and the ability to analyze heterogeneous code without language-specific rule engineering. Yet, despite their rapid adoption in software engineering tasks, there is limited empirical evidence on whether LLMs can reliably detect structural weaknesses as defined by ISO/IEC 5055:2021, especially across different programming languages and weakness categories. To the best of our knowledge, only two studies explicitly address the ISO/IEC 5055:2021 standard, underscoring the lack of systematic evaluations of LLM-based compliance tools [Curtis et al. 2022, Borg 2023].

Motivated by this gap and the need for accessible, complementary solutions beyond commercial static analysis platforms, this paper introduces GemCA, a model that leverages Gemini 2.0 Flash to automatically identify code weaknesses aligned with ISO/IEC 5055:2021. We evaluate GemCA on two representative datasets (C# and PHP) containing known vulnerabilities and conduct a multi-run empirical analysis to assess accuracy, variability, and weakness-specific behavior. This study provides one of the first quantitative assessments of an LLM applied directly to ISO/IEC 5055:2021 compliance, highlighting both the promise and the limitations of LLM-driven structural quality analysis. We evaluate GemCA's accuracy on C# and PHP datasets and report findings and implementation challenges to assess its feasibility as an accessible, complementary solution for code-quality analysis.

Despite the existence of ISO/IEC 5055:2021 and supporting commercial tools, adoption remains limited due to the standard's recent release (2021), the high cost or incomplete coverage of available platforms, and the scarcity of empirical studies. Moreover, no prior work has evaluated the use of Large Language Models (LLMs) for automated compliance with ISO/IEC 5055:2021.

The main contributions of this paper are threefold: (i) the proposal of *GemCA*, an LLM-based model for automated detection of code weaknesses aligned with ISO/IEC 5055:2021; (ii) one of the first quantitative empirical evaluations of LLM performance under the ISO/IEC 5055:2021 framework, including multiple runs to assess accuracy and variability; and (iii) an analysis of performance differences across programming languages and weakness categories, providing insights into the strengths and limitations of LLM-driven code quality assessment.

The remainder of this paper is organized as follows. Section 2 presents an overview of the ISO/IEC 5055:2021 standard. Section 3 reviews related work on LLM-based code analysis and software quality assessment. Section 4 describes the proposed methodology, including the GemCA framework, datasets, and evaluation metrics. Section 5 reports the experimental results and their analysis. Finally, Section 6

concludes the paper and outlines directions for future work.

2. ISO/IEC 5055:2021

The ISO/IEC 5055:2021 standard specifies product-level software metrics focused on source code quality [ISO 2021]. It provides engineering guidelines to guide the detection and interpretation of critical structural and architectural flaws [Nikolov 2021]. Building on the ISO/IEC 25010 quality model, it evaluates structural software quality—also known as internal quality—across four attributes: Security, Reliability, Maintainability, and Performance Efficiency.

The standard measures quality by identifying and quantifying severe code weaknesses at architectural and component levels. These weaknesses are classified according to the Common Weakness Enumeration (CWE) maintained by MITRE [ISO 2021]. Detection relies on formalized patterns that represent code structures violating best practices and quality rules, potentially causing operational or cost issues. Each detection pattern corresponds to a Quality Measure Element, termed an Automated Source Code Quality Measure (ASCQM). This framework underpins automated tools capable of analyzing source code to detect weaknesses and assess quality [ISO 2021, Nikolov 2021].

3. Related Work

Most work on static code analysis with LLMs employs cue engineering methods such as zero-shot, few-shot prompting, chain-of-thought, and Retrieval-Augmented Generation (RAG). Fine-tuning pre-trained LLMs for specific security tasks is common, offering a resource-efficient alternative to training from scratch [Zhang et al. 2024a, Zhu et al. 2024]. Some approaches use cascading inference pipelines, applying lightweight models first, followed by more powerful ones when needed [Abtahi and Azim 2025, Wang et al. 2024]. This variety reflects the dynamic, multifaceted nature of current research [Abtahi and Azim 2025]. High-quality datasets are critical for training and evaluating LLMs in security domains. Sources include real-world repositories like GitHub, synthetic data from LLMs, and established security databases such as NVD, SARD, and MITRE's CWE, CAPEC, and ATT&CK [Zhang et al. 2024a, Zhu et al. 2024]. LLMs have been increasingly studied for identifying, categorizing, and resolving vulnerable code. Recent work explores their ability to perform static analysis and detect security flaws, mapping weaknesses to taxonomies like CWE [Ezenwoye et al. 2024, Cao and Jun 2024]. These studies cover multiple languages including C/C++, Java, Python, and Android, often employing prompt engineering to boost performance.

Regarding ISO/IEC 5055:2021, only two studies have been identified. One outlines the standard's development as an industry response to the need for international source code quality metrics applicable to embedded and enterprise systems [Curtis et al. 2022]. Another addresses technical debt reduction via explicit requirements and measurements, positioning ISO/IEC 5055:2021 as a proactive tool compared to ISO/IEC 25010, which evaluates operational quality post-deployment [Borg 2023]. To our knowledge, no prior work evaluates LLMs specifically against ISO/IEC 5055:2021 quality measures, nor examines their performance across programming languages and CWE types.

Relative to prior work, which mostly focuses on vulnerability detection, our study explicitly evaluates: (i) Framework for vulnerability detection, emphasizing code weakness identification across multiple languages and CWEs; (ii) comparison of framework with traditional static analysis tools, highlighting semantic reasoning versus fixed-rule approaches; and (iii) explicit evaluation within the ISO/IEC 5055:2021 framework, enabling standard-compliant quality assessment rather than ad hoc or post-hoc metrics. This framing clarifies how our work complements and extends existing research while addressing gaps in empirical evaluation and standard adherence.

4. Methodology

This study proposes a model called *GemCA* (*Gemini Code Analyzer*) that leverages a large language model (LLM) to perform code weaknesses analysis in source code, based on the ISO/IEC 5055:2021 standard. This study is guided by the following research question:

What is the accuracy rate of the GemCA model in classifying code snippets according to the structural weakness categories defined by ISO/IEC 5055:2021?

4.1. Datasets

This study uses datasets from the SARD database [NIST 2025], which includes test programs with known weaknesses. Two test suites were selected to evaluate the model's accuracy: the "C# Vulnerability Test Suite" (2016)[1] and the "PHP Vulnerability Test Suite" (2015)[2]. Each contains code files with associated vulnerabilities, ranging from small snippets to larger applications. The choice of datasets was based on two criteria. First, both C# and PHP rank among the top 10 most-used languages on GitHub and within the top 15 in Stack Overflow's annual survey [GitHub Staff 2024, Stack Overflow 2025], suggesting that a tool trained on these languages could support a wide developer audience. Second, both datasets have been successfully used in prior work [Stivalet and Fong 2016, Fidalgo et al. 2020, Johnson et al. 2024, Arıkan et al. 2024], reinforcing their relevance.

4.2. Framework

This work proposes a framework focused on the development stage of the software development life cycle. The objective of the framework is to support software engineers, regardless of their level of expertise in security, during the identification of security weaknesses in source code. This framework represents an initial instance within a broader research effort aimed at strengthening security-related practices.

The framework targets the early identification of common security weaknesses during the software development phase. It relies on prompt engineering techniques to provide contextual and normative knowledge to a large language model. The analysis process combines three inputs: (i) a formal specification of common weaknesses, (ii) a structured prompt defining the role and objectives of the analysis, and (iii) the source code to be examined.

In this paper, *GemCA* which is based on Google's *Gemini 2.0 Flash* large language model, and designed for the detection of code vulnerabilities in compliance

[1] `https://samate.nist.gov/SARD/test-suites/105`
[2] `https://samate.nist.gov/SARD/test-suites/103`

with the ISO/IEC 5055:2021 standard, is used to exemplify the framework. The *GemCA* framework is shown in Figure 1. *Gemini 2.0 Flash* is used in this research because it is offered in a free-tier API by Google, and there is no budget to purchase other competing models, and it has a large context window of approximately 1 million tokens which allows for the incorporation of large source code and documents. [Pichai 2024].

As shown in Figure 1, the framework pipeline consists of three main components: (i) the CISQ CWE Specification, provided as a PDF document, (ii) a structured prompt defining the role of the model and the analysis task, and (iii) the input source code file. These elements are passed to the language model through the Gemini Python API.

The framework employs prompt engineering techniques to guide the model toward the identification of structural weaknesses aligned with ISO/IEC 5055 [Boonstra 2025]. A one-shot prompting strategy is used, providing the model with a representative example of the expected output. Detected weaknesses are produced using a structured XML format, where each weakness is described using the following elements: type, weakness, description, severity, file, code, and justification.

Rather than relying on line numbers—which large language models handle unreliably [Fu et al. 2024, Zhang et al. 2024b, Ball et al. 2024]—the framework instructs the model to include complete code fragments. XML is preferred over JSON to mitigate formatting issues caused by special characters commonly present in source code, such as quotes and line breaks, which frequently disrupt JSON parsing. In practice, XML proved more robust in preserving code structure.

The generated XML output is subsequently parsed using regular expressions to extract a comprehensive weakness report, including the CWE classification, severity level, affected file, code snippet, and an explanatory justification. The framework is designed as a modular and extensible artifact; although the instance presented in this work focuses on the development stage, future integration with complementary instances deployed at other stages of production line is envisioned to provide end-to-end support across the software development life cycle.

To support reproducibility, the inference configuration was explicitly specified. All experiments were conducted using the default parameter settings recommended by Google, including a temperature of 1.0, a top-p value of 0.95, a fixed top-k value of 64, and a candidate count of 1. These parameters were kept constant across all runs. The exact prompt used to guide the model is provided in Appendix A, and the CISQ CWE specification [3] was included as part of the input context. All experiments were conducted under identical configuration settings to minimize variability introduced by the inference process.

Formally, we define our model ***GemCA*** as a structured semantic analysis function over source code:

$$\text{GemCA} : \mathcal{X} \rightarrow \mathcal{R}^* \tag{1}$$

where:

[3] https://www.it-cisq.org/standards/code-quality-standards/

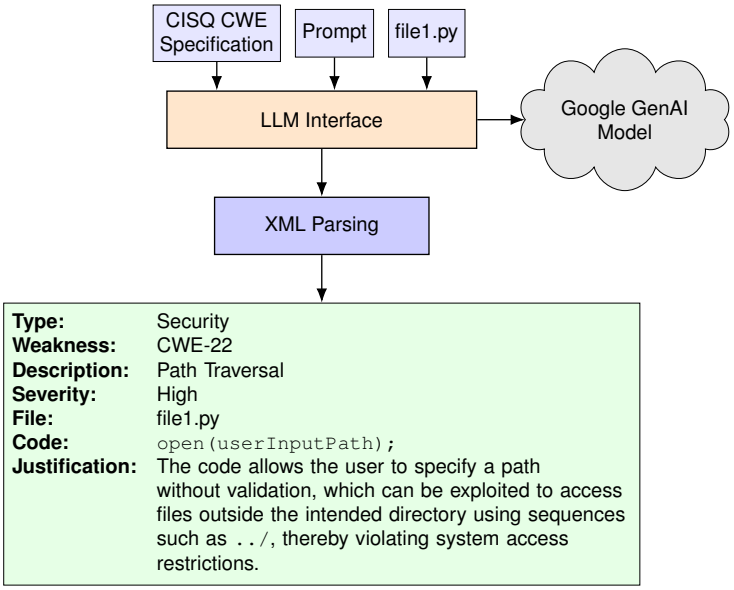

Figure 1. Workflow for Weakness Analysis Using a Cloud-Based LLM

- $\mathcal{X}$ denotes the set of all valid source code files (e.g., written in languages such as Java, C++, Python, etc.).
- $\mathcal{R}$ denotes the set of possible weakness reports, where each element $r \in \mathcal{R}$ is a tuple:

$$r = (\text{type}, \text{weakness}, \text{description}, \text{severity}, \text{file}, \text{code}, \text{justification})$$

- $\mathcal{R}^*$ denotes the set of finite lists over $\mathcal{R}$, that is, all possible collections of reports associated with a given source file.

  Thus, for a source file $x \in \mathcal{X}$, the model produces:

$$\text{GemCA}(x) = [r_1, r_2, \ldots, r_n] \in \mathcal{R}^* \tag{2}$$

where each r_i represents a distinct weakness detected in x, structured according to the criteria defined by the ISO/IEC 5055:2021 standard. Each prompt included the entire source file to preserve contextual information.

4.3. Metrics

The dataset consists of multiple test cases represented as pairs (x, y), where:

- x: denotes the source code file to be analyzed.
- y: is a tuple containing the vulnerability identifier (v) and the line number (l) where it is located.

As a result, the model produces an output r_i that contains $\hat{y}$, a tuple composed of the predicted vulnerability identifier ($\hat{v}_i$) and the code fragment in which it was detected ($\hat{c}_i$). Based on this information, an accuracy metric was defined as follows:

$$\text{Accuracy} = \frac{1}{n} \sum \text{CorrectPrediction}(y_i, \hat{y}_i) \times 100 \tag{3}$$

with

$$\text{CorrectPrediction}(y_i, \hat{y}_i) = \begin{cases} 1 & \text{if } v_i = \hat{v}_i \text{ and } l_i \in \hat{c}_i \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

where:

- $y_i = (v_i, l_i)$ is the ground truth annotation: the actual vulnerability identifier and the line number where it occurs.
- $\hat{y}_i = (\hat{v}_i, \hat{c}_i)$ is the model's prediction: the predicted identifier and the returned code fragment (interpreted as a set of lines).
- $l_i \in \hat{c}_i$ indicates that the line l_i is contained within the predicted code fragment $\hat{c}_i$.

Note that this metric only considers a prediction correct if the model successfully identifies both the vulnerability type and the corresponding code fragment. Additionally, to mitigate comparison errors—stemming from the non-deterministic nature of language models—a character-level similarity metric was introduced to compare code fragments.

To address this, a *sliding window* technique was employed by comparing the shorter string (between l_i and c_i) over the longer one. At each shift, similarity was computed using the *Gestalt pattern matching* algorithm. A similarity threshold of 0,8 was applied to consider two strings as equivalent.

The formula used to compute the similarity is:

$$\text{Similarity} = \frac{2 \times M}{|S_1| + |S_2|} \tag{5}$$

where:

- M is the number of matching characters between the two strings.
- $|S_1|$ and $|S_2|$ are the lengths of the two sequences being compared.

Accuracy was selected as the primary metric because each test case contains exactly one known weakness, making multi-label metrics such as precision and recall less informative in this context.

4.4. Evaluation

For the evaluation, a total of 200 test cases were randomly selected—100 from each dataset ("C# Vulnerability Test Suite" and "PHP Vulnerability Test Suite")—ensuring that the included weaknesses corresponded to CWEs defined by the ISO/IEC 5055:2021 standard. A sanitization step was applied to prevent the LLM from inferring vulnerabilities based on metadata. Specifically, all comments (e.g., // flaw) indicating the presence of weaknesses were removed, and file names were replaced with artificial identifiers, as original names contained CWE codes. This change had no impact on model performance since each test case was limited to a single file.

Each test case (*prompt + CWE specification + code*) was submitted individually to the LLM, which returned structured XML reports of the detected weaknesses. The entire process was repeated 15 times to assess consistency. Results were computed separately for each dataset. The implementation is publicly available at: gemca-llm-code-analyzer.

4.5. Threats to Validity

This study is subject to several threats to validity that should be considered when interpreting the results.

External validity concerns the generalizability of the findings beyond the experimental setting. The study relies on a controlled subset of SARD test cases and focuses on two programming languages (C# and PHP), which may not fully represent the diversity of real-world industrial systems. Although SARD provides curated ground truth aligned with the Common Weakness Enumeration (CWE) taxonomy, it does not capture the scale, architectural complexity, or contextual variability of large production environments. Furthermore, the evaluation focuses on a subset of security-related CWEs, whereas ISO/IEC 5055:2021 defines quality measures across multiple attributes, including reliability, maintainability, and performance efficiency. As a result, the findings should be interpreted as evidence of CWE detection capabilities under a standard-informed prompting strategy, rather than a comprehensive assessment of ISO/IEC 5055:2021 compliance. Future work should incorporate additional datasets, including real-world open-source repositories, and expand coverage to other quality attributes defined by the standard.

Construct validity may be influenced by how the task and evaluation criteria were operationalized. In this study, the problem is framed as the detection and classification of CWE instances guided by ISO/IEC 5055:2021, rather than a full standard-compliant quality assessment. A prediction was considered correct only when the model simultaneously identified the appropriate CWE identifier and the corresponding vulnerable code fragment. While this formulation enables controlled evaluation, it does not capture higher-level aggregation mechanisms required by the standard, such as quality scoring across multiple attributes. Moreover, because LLM outputs are sensitive to prompt formulation, alternative prompting strategies (e.g., few-shot, specification summarization, or retrieval-based approaches) could lead to different performance outcomes. Future work should explore these variations through systematic ablation studies.

Internal validity threats arise from evaluation-related design choices and the absence of comparative baselines. Although repeated executions were performed to reduce the impact of non-determinism, the study does not include comparisons against traditional static analysis tools (e.g., SAST solutions) or alternative large language models. Consequently, the reported accuracy cannot be directly contextualized in relation to established approaches. Additionally, no ablation analysis was conducted to isolate the contribution of individual components, such as the inclusion of the CWE specification or the one-shot prompting strategy. Future work should incorporate baseline comparisons and controlled ablation experiments to better understand the relative impact of each design decision.

Finally, conclusion validity may be affected by the limited number of programming languages and the uneven distribution of CWE categories. Although statistical tests such as Shapiro–Wilk, Levene, t-tests, ANOVA, and Kruskal–Wallis were applied where appropriate, deviations from normality and homoscedasticity in certain cases may influence statistical power. A broader experimental design incorporating additional datasets, languages, evaluation metrics, and model configurations would

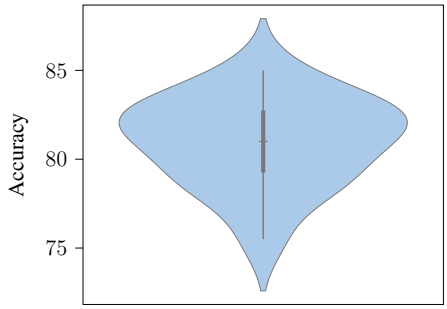

Figure 2. Accuracy of the *GemCA* model over 15 repetitions

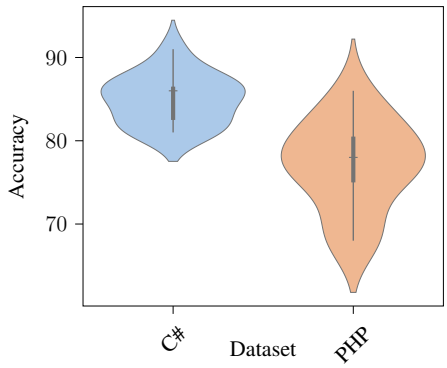

Figure 3. Accuracy of the *GemCA* model over 15 repetitions for each programming language

further strengthen the robustness and reliability of the conclusions.

5. Results

5.1. Accuracy in classifying code snippets

A violin plot(Figure 2) shows both the median and mean accuracy at 81%, with a standard deviation of 2,5. Accuracy ranged from 75,7% to 85%, suggesting strong performance with moderate variability, likely due to the heterogeneity of code complexity across cases. *GemCA* shows reasonably strong classification performance within the scope of the evaluated datasets as defined by ISO/IEC 5055:2021. To classify code fragments, 200 test cases were analyzed per iteration and a Shapiro–Wilk normality test yielded a p-value of 0,6966, indicating no significant deviation from normality and supporting parametric analyses.

The analysis using a violin plot (Figure 3) shows that the model achieves an average accuracy of 84,93% for C# and 77,06% for PHP, reinforcing that *GemCA* performs better on C# code. Regarding variability, accuracy values for PHP display greater dispersion compared to C#. The absolute difference between variances is approximately 2,35.

To assess whether the programming language affects the accuracy of the *GemCA* model when applying the ISO/IEC 5055:2021 standard for source code security analysis, a Student's t-test was conducted. Prior to the test, the assumptions of normality and homogeneity of variances were verified. Normality was assessed using the Shapiro–Wilk test, yielding p-values of 0,3429 for C# and 0,4803 for PHP. The data from both samples

can be considered normally distributed. Next, Levene's test was applied to assess the homogeneity of variances between the two groups, resulting in a p-value of 0,1355. This suggests insufficient evidence to reject the null hypothesis of equal variances, thereby satisfying a key assumption for conducting the t-test. With these assumptions confirmed, the Student's t-test was performed, producing a p-value of 0. This leads to the conclusion that statistically significant differences exist between C# and PHP regarding model accuracy. Thus, the programming language appears to significantly influence the model's performance. Although Levene's test did not reveal a statistically significant difference, the higher variability for PHP suggests that the model encounters greater challenges in consistently identifying patterns within this language.

Further analysis, illustrated by the violin plot in Figure 4, shows substantial variation in model accuracy depending on the weakness category. For instance, the model achieved consistently high accuracy with low variability for CWE-89 and CWE-79. In contrast, classes such as CWE-22 demonstrated considerable dispersion in accuracy, indicating instability in detection across iterations.

To assess whether the type of software weakness influences the accuracy of the *GemCA* model when applying the ISO/IEC 5055:2021 standard for source code security analysis, a weighted analysis of variance (weighted ANOVA) was conducted. This analysis accounts for the natural class imbalance present in the dataset, where the number of instances per weakness category varied significantly. Rather than artificially equalizing sample sizes across categories—an approach that could introduce bias or distort the model's performance profile—a weighted statistical approach was adopted. Each weakness type was assigned a weight inversely proportional to its frequency, thereby compensating for underrepresented classes while preserving the dataset's original structure.

Prior to conducting the ANOVA, standard assumptions were evaluated. The Shapiro–Wilk test for normality indicated that all categories except CWE-78 could be considered approximately normally distributed at the 0,05 significance level. CWE-78, however, yielded a p-value of 0,0086, indicating a violation of the normality assumption. Homogeneity of variances was also tested using Levene's test, which produced a highly significant p-value (p = 0.0000), confirming heteroscedasticity among the groups. Despite the violations of some ANOVA assumptions, the weighted ANOVA was carried out to account for class imbalance and unequal variances. The result was statistically significant ($p = 0.0000$), indicating that the mean accuracy varied significantly across different weakness types.

To reinforce the robustness of this conclusion, a non-parametric Kruskal–Wallis test was also performed. This test does not assume normality or equal variances and yielded the same outcome (p = 0.0000), thus confirming the ANOVA results. Together, these findings strongly suggest that the type of weakness has a significant impact on the model's classification performance.

To assess the model's stability, 100 test cases were selected, and 15 independent runs were performed for each case and for each language, keeping the parameters described in Section 4.2 constant. For each run, we recorded whether the model correctly classified the corresponding CWE. The analysis was conducted using Fleiss' Kappa

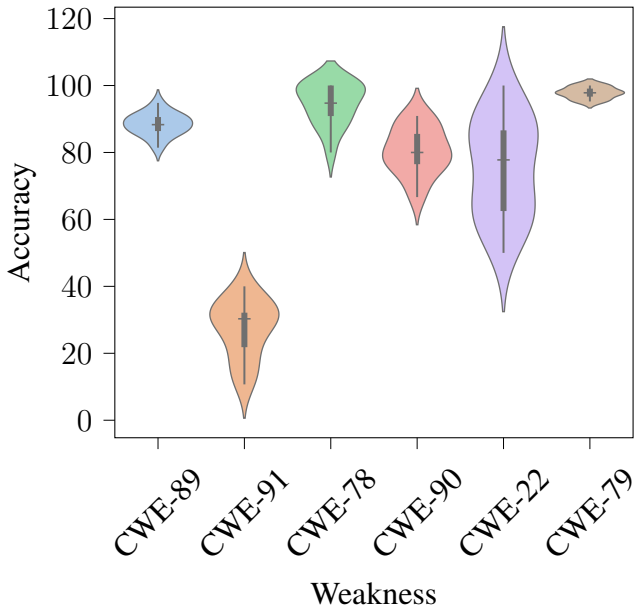

Figure 4. Accuracy of the *GemCA* model over 15 repetitions for each weakness category

coefficient, a statistical measure that allows for the evaluation of the degree of agreement among multiple evaluators when assigning categories to a set of items. In this context, each run is considered an independent evaluator, which allows for the quantification of the consistency of the GemCA model's classification decisions regarding C# and PHP code [Landis and Koch 1977].

The overall results show a Kappa value of 0.706, which corresponds to a substantial level of agreement according to [Landis and Koch 1977], suggesting highly consistent behavior across runs. At the language level, a value of 0.727 was obtained for C# and 0.684 for PHP, both within the substantial range, although with slightly greater stability in C#. The case-by-case analysis indicates that the model produces completely consistent results in 73% of cases (146 out of 200), with 82% (82 out of 100) in C# and 64% (64 out of 100) in PHP, highlighting differences in model stability between programming languages. It is important to note that CWE detection does not by itself constitute full ISO/IEC 5055 compliance, which requires aggregate scoring rules beyond the scope of this study

5.2. Discussion

Overall, the model achieved a mean accuracy of 81%, with low variability across the 15 repeated runs, indicating reasonable consistency in its outputs. Nevertheless, the 19% error rate highlights that LLM-generated assessments should be interpreted with caution, particularly in high-stakes or safety-critical contexts, also, these results should be interpreted within the controlled evaluation setup and are not yet generalizable to industrial-scale systems. Accuracy was strongly influenced by the programming language: GemCA performed substantially better on C# (median 84.93%) compared to PHP (77.06%). This difference likely reflects structural and typing differences between the languages, which may affect the model's ability to recognize patterns and semantic cues.

Weakness type also played a significant role. Common and well-defined vulnerabilities, such as CWE-89 (SQL Injection) and CWE-79 (Cross-Site Scripting), were reliably detected, whereas CWEs requiring nuanced reasoning, such as CWE-91 (XML Injection) and CWE-22 (Path Traversal), showed lower or inconsistent detection rates. Errors were often caused by confusion between semantically similar CWEs or when multiple weaknesses were present in the same code snippet. Weaknesses involving subtle control-flow dependencies, implicit execution paths, or contextual assumptions—such as input validation and authorization checks—consistently exhibited lower accuracy, suggesting that the model is more adept at recognizing explicit API misuse than reasoning about implicit semantic behavior.

These patterns suggest that LLMs may capture general vulnerability themes effectively, their precision decreases for complex or context-dependent weaknesses. This emphasizes a finding consistent with prior work on LLM reliability in security contexts.

6. Conclusions

This study investigated the feasibility and performance of using a general-purpose large language model (LLM), *GemCA* (based on *Gemini 2.0 Flash*), to detect structural code weaknesses as defined by the ISO/IEC 5055:2021 standard. The findings demonstrate that LLMs show potential to support software quality analysis tasks, achieving a median accuracy of 81% across a series of controlled experiments. Notably, the model showed strong and stable performance for certain types of vulnerabilities—particularly in strongly typed languages like C#—but exhibited significant variability across both weakness types and programming languages. These results underscore the model's potential as a supportive framework in software assurance workflows, especially for early-stage reviews or prioritization tasks. However, its limitations—including prompt sensitivity, inconsistent performance for complex weaknesses (e.g., CWE-91), and opacity in decision-making—highlight the need for human oversight and caution in critical applications.

This study contributes one of the first quantitative evaluations of LLMs in the context of ISO/IEC 5055:2021, a standard that has gained traction in both industry and research for measuring software structural quality. The proposed methodology, which integrates prompt engineering with automated evaluation and visual analysis techniques, provides a replicable approach for assessing model capabilities in structured code assessments.

Looking forward, several research directions can help mature this line of inquiry. First, expanding the dataset to include additional languages and ISO/IEC 5055:2021 weakness categories beyond security (e.g., maintainability and performance) would provide a more comprehensive evaluation. Second, comparative studies with alternative LLMs—such as *GPT-4*, *LLaMA 3*, or *Claude*—are needed to establish whether current findings generalize across architectures. Third, advanced prompting strategies and few-shot paradigms should be explored to mitigate variability and improve performance on low-frequency weaknesses. Task-specific fine-tuning on curated datasets aligned with ISO/IEC 5055:2021 may further improve model specialization. Finally, head-to-head comparisons between LLM-based systems and industrial SAST tools (e.g., SonarQube, CAST) using standardized datasets could offer critical insights into relative strengths,

weaknesses, and use-case boundaries.

From a research perspective, this work fills a gap by applying LLMs directly within the ISO/IEC 5055:2021 framework, a dimension largely overlooked in prior literature. Existing CASE tools like SonarQube, CAST, and Kiuwan remain more mature, but *GemCA* opens a new path by leveraging LLMs for dynamic, language-aware source code analysis. Further refinement is needed, particularly to expand coverage beyond security-related CWEs and to strengthen integration into industry-grade pipelines. The use of the framework is limited by a daily request cap in the free-tier version. Organizations aiming for larger-scale or continuous analyses would require a paid plan, representing a practical and financial constraint for long-term integration. Finally, while LLMs are not ready to replace traditional static analysis or expert review, their ability to interpret code semantically and adapt to specification-driven tasks like ISO/IEC 5055:2021 assessments —particularly in scenarios where speed, adaptability, and early feedback are valued.

7. Acknowledgments

This research was partially supported by Project No. 834-C4-157 ECCI-CITIC-UCR, and the Graduate Program in Computer Science and Informatics at the University of Costa Rica (UCR). During the preparation of this work, AI-based tools were used to support the refinement of academic English, improve readability, and ensure consistency in terminology across sections. After using these tools, the authors carefully reviewed, validated, and edited all generated suggestions to ensure that the manuscript reflects their own scientific contributions, interpretations, and insights. The authors take full responsibility for the integrity, accuracy, and originality of the content presented in this article.

References

Abtahi, S. M. and Azim, A. (2025). Augmenting Large Language Models with Static Code Analysis for Automated Code Quality Improvements. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 82–92, Ottawa, ON, Canada. IEEE.

Arıkan, S. M., Koçak, A., and Alkan, M. (2024). Automating shareable cyber threat intelligence production for closed source software vulnerabilities: a deep learning based detection system. *International Journal of Information Security*, 23(5):3135–3151.

Ball, T., Chen, S., and Herley, C. (2024). Can we count on llms? the fixed-effect fallacy and claims of gpt-4 capabilities.

Boonstra, L. (2025). Prompt Engineering.

Borg, M. (2023). Requirements on Technical Debt: Dare to Specify Them! *IEEE Software*, 40(2):8–12.

Cao, D. and Jun, W. (2024). Llm-cloudsec: Large language model empowered automatic and deep vulnerability analysis for intelligent clouds. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, Vancouver, BC, Canada. IEEE.

CISQ (2025). Software quality standards – iso 5055. Accessed: Apr. 14, 2025.

Curtis, B., Martin, R. A., and Douziech, P.-E. (2022). Measuring the Structural Quality of Software Systems. *Computer*, 55(3):87–90.

Ezenwoye, O., Pinconschi, E., and Roberts, E. (2024). Exploring ai for vulnerability detection and repair. In *2024 Cyber Awareness and Research Symposium (CARS)*, pages 1–9, Grand Forks, ND, USA. IEEE.

Fidalgo, A., Medeiros, I., Antunes, P., and Neves, N. (2020). Towards a Deep Learning Model for Vulnerability Detection on Web Application Variants. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 465–476, Porto, Portugal. IEEE.

Fu, T., Ferrando, R., Conde, J., Arriaga, C., and Reviriego, P. (2024). Why do large language models (llms) struggle to count letters?

GitHub Staff (2024). Octoverse: AI Leads Python to Top Language as the Number of Global Developers Surges. `https://github.blog/news-insights/octoverse/octoverse-2024/`. The GitHub Blog.

ISO (2021). Information technology - software measurement - software quality measurement - automated source code quality measures. Technical Report 5055, ISO/IEC.

Johnson, D., McDonald, J. T., Benton, R. G., and Bourrie, D. (2024). Effectiveness of image-based deep learning on token-level software vulnerability detection. In *SoutheastCon 2024*, pages 1054–1063.

Karg, L. M., Grottke, M., and Beckhaus, A. (2009). Conformance quality and failure costs in the software industry: An empirical analysis of open source software. In *2009 IEEE International Conference on Industrial Engineering and Engineering Management*, pages 1386–1390.

Landis, J. R. and Koch, G. G. (1977). The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159.

Nikolov, R. (2021). ISO 5055 Standard Explained: Is Your Software Rock Solid, Efficient, and Safe? `https://www.castsoftware.com/pulse/iso-5055-standard-explained_is-your-software-rock-solid-efficient-and-safe`.

NIST (2025). Software assurance reference dataset.

Pichai, S. (2024). Introducing gemini 2.0: Our new ai model for the agentic era. `https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/`. Google Blog.

Stack Overflow (2025). 2025 Developer Survey. `https://survey.stackoverflow.co/2025/technology/#1-programming-scripting-and-markup-languages`. Annual Online Survey.

Stivalet, B. and Fong, E. (2016). Large scale generation of complex and faulty php test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 409–415.

Wang, Y., Wang, X., Yu, H., Gao, F., Liu, X., and Wang, X. (2024). A study on c code defect detection with fine-tuned large language models. In *2024 31st Asia-Pacific Software Engineering Conference (APSEC)*, pages 437–441.

Zhang, C., Wang, L., Fan, D., Zhu, J., Zhou, T., Zeng, L., and Li, Z. (2024a). Vtt-llm: Advancing vulnerability-to-tactic-and-technique mapping through fine-tuning of large language model. *Mathematics*, 12(9).

Zhang, X., Cao, J., and You, C. (2024b). Counting ability of large language models and impact of tokenization.

Zhu, J., Ge, H., Zhou, Y., Jin, X., Luo, R., and Sun, Y. (2024). Detecting source code vulnerabilities using fine-tuned pre-trained llms. In *2024 IEEE 17th International Conference on Signal Processing (ICSP)*, pages 238–242.

A. Prompt

This appendix provides the prompt used in the GemCA framework.

```
# Task
You are a code evaluator operating under the ISO/IEC 5055:2021 standard for Automated Source Code Quality Measures.
Your task is to analyze provided source code snippets and deliver detailed feedback focusing on the following
four quality aspects defined in the standard:

- Reliability
- Security
- Performance Efficiency
- Maintainability

# Output Format
Your response must be an XML document. Each identified issue must be represented as a separate <Issue> element
with the following child elements:

<Type>           One of: "Reliability", "Security", "Performance Efficiency", or "Maintainability"
<Weakness>       A concise identifier of the weakness (e.g., "CWE-1"). Provide only the CWE identifier, not its description.
<Description>    A description of the CWE weakness (e.g., "Improper Input Validation")
<Severity>       One of: "Critical", "High", "Medium", or "Low"
<File>           The filename(s) where this issue is found. If multiple related files, separate them with commas.
<Code>           The exact source code segment where the issue occurs, preserving original formatting (indentation, line breaks).
                 Truncate to a maximum of 30 lines; append "..." if truncated.
<Justification>  A clear explanation of why this code is an issue based on the standard

Example output format:
'''xml
<Issues>
  <Issue>
    <Type>Security</Type>
    <Weakness>CWE-89</Weakness>
    <Description>SQL Injection</Description>
    <Severity>High</Severity>
    <File>login.cs</File>
    <Code>string query = "SELECT * FROM users WHERE name = '" + userInput + "'";</Code>
    <Justification>This code directly concatenates user input into a SQL query, making it vulnerable to
    SQL injection.</Justification>
  </Issue>
  <!-- Additional <Issue> entries if needed -->
</Issues>
'''

DO:

- Keep the code as it.
- You must find all potential issues in the code snippet provided, even if their severity is low.
- If you cannot find any issues, return an empty XML array:
'''XML
<Issues>
</Issues>
'''

DO NOT:

- Do not modify the code formatting, indentation, or line breaks.

# Input

You will be given the standard rules and a file or whole coding project where each file contains code.

The standard rules are:
{standard}

The code is:
{code_snippet}
```