# Measuring Accuracy and Consistency of Large Language Models for OOAD Principles

**Federico Ortega Riba, Sheetal Sharma**
Departments of Linguistics and Computer Science
University of Colorado at Boulder
{federico.ortegariba, sheetal.sharma}@colorado.edu

## Abstract

Large language models (LLMs) can generate code that passes tests, yet their judgments about object-oriented design quality remain unclear. In this study, we explore whether LLMs "understand" OOAD's SOLID principles by proposing a two-part evaluation framework: (1) accuracy between model and human ratings and labeling, and (2) consistency of model labels under minimal prompt perturbations. Using an already existing dataset of 240 curated code snippets, we focus on Java/Python items and add human annotations to include a $\leq$ 80-word explanation, an adherence score (1–5), and a violation-severity score (1–5). We elicit structured JSON outputs from a reproducible open Llama model. Accuracy is measured with Spearman's $\rho$ (rank agreement) and Pearson's $r$ (calibration) on $N$ = 50 items. Our results show weak alignment for adherence (Spearman $\rho$ = 0.279, Pearson $r$ = 0.278) and negligible, non-significant alignment for severity (Spearman $\rho$ = 0.124, Pearson $r$ = 0.080). Our results suggest current LLMs capture only a weak signal of SOLID adherence and struggle to calibrate severity, motivating benchmarks that target design-level reasoning beyond functional correctness. We release prompt, code, and scripts for full reproducibility. In a complementary analysis of categorical SOLID-violation labels, we find that the model is highly self-consistent (macro label agreement $\approx$ 0.96) yet only moderately accurate relative to human labels (macro-F1 = 0.79, accuracy = 0.80), with most residual errors arising from confusion between DIP and other principles. Our results suggest current LLMs capture only a weak signal of SOLID adherence and struggle to calibrate severity. We release prompt, code, and scripts for full reproducibility.

## Introduction

In recent years, modern LLMs have demonstrated remarkable ability to generate functionally correct code: they pass tests and fulfill the immediate requirements (Chen et al. 2021). But correctness in terms of functionality is only part of what matters in software engineering. Code must also adhere to sound architectural and design principles, especially when we talk about long-term maintainability and scalability (Martin 2023).

Among the most influential design guidelines for object-oriented software are the SOLID principles: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion. These principles aim to keep code modules cohesive, loosely coupled, extendable without modification, and adequate for reuse (Martin 2023). Yet, despite the power of LLMs, there is evidence that they may produce code which is functionally correct but design-flawed, violating one or more of these SOLID principles. In other words: the code works, but its architecture is brittle (Idrisov and Schlippe 2024).

From a software engineering perspective, this is a serious issue: such violations often lead to harder refactoring and increased maintenance cost over time (Palomba et al. 2018). This leads us to the motivation for our project: a framework to evaluate both accuracy and consistency of LLM judgments on SOLID principles.

In order to do this, we performed human annotation for explanations of code snippets as well as adherence and violation severity scores. Our results are compared to a modern LLM which outputs another Likert score in the same scale of 1 to 5. Following this, accuracy is measured with Spearman and Pearson correlation, and consistency is computed running two prompts (AB and BA) and measuring label agreement. We discuss where sensitivity is highest and how well a human gold-standard annotation compares to an LLM's judgment. This framework offers a compact recipe to test whether LLMs "get" SOLID and whether their answers remain accurate under minimal prompt changes.

## Related Work

Machine learning techniques for code understanding and summarization have been studied for over a decade, laying the foundation for today's LLM-based tools. Early work framed source code as a form of natural language, leveraging the "naturalness" of software as surveyed extensively by Allamanis et al. (2018). Researchers applied neural sequence models to generate code summaries and predict code properties. For example, Iyer et al. (2016) and Allamanis, Peng, and Sutton (2016) developed neural attention models to produce short descriptions of code snippets. Subsequent approaches learned richer code representations by incorporat-

ing program structure. Notably, the code2vec and code2seq models encoded paths in abstract syntax trees to create vector embeddings of code, enabling tasks like method name prediction and documentation generation (Alon et al. 2019b; 2019a). These foundational works demonstrated that source code's semantics can be captured by data-driven models, presaging the recent surge in large-scale code intelligence.

With the advent of large pre-trained models, the intersection of LLMs and object-oriented programming (OOP) advanced significantly. CodeBERT (Feng et al. 2020) introduced a bimodal Transformer trained jointly on natural language and source code, supporting tasks like code search and documentation generation. CodeT5+ (Wang et al. 2023b) further extended encoder–decoder models with multiple pretraining objectives, achieving strong performance across over twenty code benchmarks. General-purpose LLMs such as GPT-3 (Brown et al. 2020) showed robust few-shot programming capabilities, while Codex (Chen et al. 2021), fine-tuned on billions of lines of code, markedly improved functional code synthesis accuracy on the HumanEval benchmark. Qwen2.5-Coder (Hui et al. 2024) scaled these ideas further by training on over 5.5 trillion tokens, resulting in competitive performance on code generation, reasoning, and repair benchmarks. These developments established the broad coding competence that underlies later efforts to evaluate design-level understanding.

Recent empirical studies have examined how LLMs generalize across programming languages and reasoning tasks. Baltaji et al. (2025) provide the largest study to date on cross-lingual transfer in programming languages, showing that pretraining in high-resource languages improves performance in lower-resource ones across tasks like clone detection, error detection, and code repair. Xu et al. (2025) introduced CRUXEVAL-X, a multilingual benchmark of 19 languages with fully automated test-guided construction. Their findings show non-trivial cross-language generalization, even for Python-only pretrained models. These results indicate that modern LLMs capture structural and semantic programming concepts, though these benchmarks focus largely on functional correctness and reasoning, not OOP design principles, which is the focus of our study.

Researchers have increasingly applied LLMs to improve object-oriented code quality, including detection of design flaws or code smells. Wu et al. (2024) introduced iSMELL, a Mixture-of-Experts architecture combining static analysis tools with an LLM to detect complex smells such as God Class, Feature Envy, among others. Their model achieves an F1 score of 75.17%, and produces refactorings rated positively by human evaluators. Sadik and Govind (2025) benchmark GPT-4 and DeepSeek-V3 on smell detection, showing competitive precision and recall relative to specialized detectors. Khajezade et al. (2024) evaluate LLMs for clone detection and demonstrate that large models outperform earlier baselines but remain limited in scalability. In design pattern recognition, Schindler and Rausch (2025) show that LLMs can identify structural roles in patterns like Factory or Observer with moderate success, although sensitivity to prompt phrasing remains a limitation. These studies collectively suggest that LLMs encode a meaningful amount of structural and architectural knowledge relevant to object-oriented design.

Work specifically targeting SOLID principles and design violations has emerged only recently. Martins, Firmino, and De Mello (2024) present a GitHub-integrated code review bot that uses an LLM to detect violations of SOLID principles in pull requests. Their case study indicates that the system successfully flags potential design issues and assists reviewers, although the evaluation remains qualitative. The first systematic benchmark is introduced by Pehlivan et al. (2025), who curate labeled datasets of SOLID violations and evaluate multiple LLMs across prompting strategies. Their results show that LLMs can explain violations coherently but vary substantially in detection accuracy across principles and prompts, raising questions about consistency and robustness in design-level reasoning. These two works most directly motivate the present study, which extends empirical investigation of LLM-based design assessment.

LLMs have also been explored in software maintenance. For example, Idrisov and Schlippe (2024) evaluate human- and AI-generated code on correctness, efficiency, and maintainability across Java, Python, and C++. They find that while only 20.6% of AI-generated solutions are fully correct, many incorrect ones require minimal fixes, providing significant development time savings. Traditional software engineering work (e.g., Palomba et al., 2018) demonstrates that the diffuseness of smells negatively affects maintainability, connecting LLM-based smell detection to real-world maintenance outcomes.

Another important line of research evaluates LLMs as judges. In natural language evaluation, Wang et al. (2023a) show that ChatGPT judgments correlate strongly with human ratings across summarization and story generation tasks. GPTScore (Fu et al. 2023; 2024) proposes an instruction-based scoring framework across 22 evaluation aspects and 37 datasets, demonstrating competitive performance with traditional metrics. G-Eval (Liu et al. 2023) further improves correlation by prompting GPT-4 with chain-of-thought and structured forms. Related to robustness, Elazar et al. (2021) show that LLMs are often inconsistent on paraphrased factual prompts, and Lu et al. (2022) demonstrate extreme sensitivity to prompt order in few-shot settings. Neuro-symbolic methods such as Calanzone, Teso, and Vergari (2025) aim to enforce logical consistency in LLM outputs. These findings are highly relevant because evaluating design principles requires consistent, reasoning-based judgments rather than pattern matching.

Collectively, these studies establish that (1) LLMs possess strong code understanding and generation capabilities, (2) they can detect many object-oriented smells and design flaws, (3) emerging benchmarks show mixed performance on SOLID principle detection, and (4) LLM evaluators require careful handling due to sensitivity and inconsistency. This motivates our investigation into whether LLMs can serve as reliable evaluators of object-oriented design quality.

# Methodology

## Original Dataset and LLM Choice

The dataset we have primarily used for the experiment is from the paper *Are We SOLID Yet?* (Pehlivan et al. 2025)[1]. It contains 240 manually validated examples covering all five SOLID principles, organized as follows:

- **Languages:** Java, Python, Kotlin, and C#.
- **Difficulty levels:** Easy, Moderate, Hard.
- **Composition:** 4 examples per principle × 3 difficulty levels × 4 languages = 48 examples per JSON file, for a total of 240 examples.
- **Principles covered:** SRP, OCP, LSP, ISP, DIP (balanced across the dataset).

Each record contains a single intentional violation and is paired with a non-violating (refactored) version. In order to perform human annotation, we have filtered by language to retrieve only Java and Python examples, since these are the most popular programming languages and are familiar to most software engineers worldwide (PYPL 2025). In addition to this, our language selection was not restricted to only Java because current research has demonstrated that a diverse selection of programming languages can enhance LLMs' capabilities to generate accurate code-related inference (Baltaji et al. 2025; Xu et al. 2025).

We input each code snippet which violates a certain OOAD principle into our structured evaluation prompt described in the following section. The refactored snippets will act as post-analysis positive controls, while the violating versions will serve as negative controls (low adherence and high violation severity). This setup enables direct comparison between LLM and human consistency across paired examples.

For this study, we used open models from the Llama4 (Meta 2025) family for the sake of reproducibility. Current literature also presents close-source models like GPT or Claude as a more suitable option for inference; however, these models require paid API keys, which makes them not suitable for this proyect. Specifically, we evaluated the performance of our framework with `llama-4-maverick-17b-128e-instruct-fp8` to make calls for inference.

## Human Annotation for OOAD Principles

Using LLMs as self-judges that output a brief rationale plus numeric ratings is well established in the current literature (Wang et al. 2023a; Fu et al. 2024; Liu et al. 2023). Building up on the idea of LLMs as self-evaluators, we conducted human annotation and added the following fields to the original dataset:

- `explanation` ($\leq$ 80 words)
- `adherence_score` (Likert score 1–5, higher is better)
- `violation_severity` (Likert score 1–5, higher is worse)

[1]Replication package can be found here: https://zenodo.org/records/17008547 (Accessed 10/11/2025)

The dual scores capture both a positive "how SOLID is this?" view and a negative "how serious are any violations?" view; together they make it easy to compare models even when one inflates positives or negatives. Scales are anchored as follows: 1 = very poor / negligible adherence (or negligible severity) and 5 = exemplary adherence (or severe violations). Models are told to output only the required JSON-structure.

Our prompting strategy utilizes zero-shot prompting as the example that can be find in the Appendix A. Our initial strategy only uses zero-shot for open models, since the label agreement experiments already introduce a few-shot settings nature in this task.

The ability to use and understand OOAD concepts lies in our own human evaluation and annotation for the same dataset we provided in this section, as well as the pipeline utilized for our experiments. Each code item is independently annotated by one of the two authors in the same field structure as the expected LLM's output. An example of our own annotation for a dataset example can be seen in Appendix B.

## Evaluation of Accuracy

Our tasks enforce a strict structured prediction JSON output for every model and annotated example. Prompts are provided in zero-shot form only, but this can be modified in the original prompt module.

To evaluate accuracy, comparisons with the LLMs' answers will be conducted with Spearman correlation and Pearson Correlation Coefficient for `adherence_score` and `violation_severity`. Accuracy of explanations with a qualitative analysis is left for future work.

## Evaluation of Consistency

LLMs are sensitive to the order of demonstrations and to minor prompt template changes (Elazar et al. 2021). Shuffling examples can shift predictions even when content is constant. Prior work shows large variance from example ordering and template choices across families and sizes of LLMs, so a rigorous accuracy study should report consistency under permutations, not just mean performance (Lu et al. 2022). Following this idea, for each test item in the dataset that we will annotate, we use two few-shot exemplars A and B and build two prompts that differ only in example order: AB and BA. Each prompt is executed k $\geq$ 2 times with identical decoding (e.g., temperature = 0). Every run must return the expected JSON format. To compute a single categorical label per run, we define the primary label as the element in `violation`. Label Agreement (LA) for an item is the fraction of the 2k runs that share the most common primary label, described as follows:

$$\mathrm{LA} = \frac{\max_{\ell \in \{\text{SRP,OCP,LSP,ISP,DIP,NONE}\}} \mathrm{count}(\ell)}{2k}$$

This metric directly captures how robust the model's categorical judgment is. We then measure label accuracy of the LLM's predictions.

## Experimental Setup

### Experiments for Accuracy

We evaluate accuracy by comparing human Likert-scale judgments with LLM self-judgments on the same code snippets. The prompt constrains rationales to $\leq 80$ words and coerces both scores to 1–5 integers to align with our annotation rubric. After obtaining model outputs, we compute Spearman's $\rho$ (rank agreement) and Pearson's $r$ (linear calibration) for both `adherence_score` and `violation_severity`, and render a 5x5 matrix to visualize matches.

Our implementation consists of small modules in Python:

- `solid_accuracy_evaluator_strings_only.py`: builds the system/user prompts and calls the model via LlamaJudgeClient, then extracts the first valid JSON object with JsonExtractor and clips scores to the 1–5 range.
- `solid_accuracy_evaluator_lists_strings.py` is a batch wrapper that accepts a single string, a JSON array of strings, or a loose TXT file.
- `extract_json_values.py`: to compare humans vs. model, this writes a clean JSON array.
- `spearman_pearson_correlation_from_txt.py` aligns records by index, drops missing pairs, computes $\rho$ and $r$ (with $p$-values), and saves a figure.

Defaults (temperature=0.0, fixed schema, pretty/non-pretty JSON) are chosen for determinism and reproducibility. The code follows OOAD principles in that each class/module has one job: prompt templating, model I/O, JSON extraction, correlation/plotting. Models and prompt limits are configurable via CLI and new metrics or schemas can be added without modifying core logic. `SolidAccuracyEvaluator` depends on an abstracted `LlamaJudgeClient` wrapper rather than a concrete backend, making the evaluator portable across providers. The full implementation, including scripts and READ.ME with exact commands, is available in our GitHub repository[2] for replication and extension.

### Experiments for Consistency

We evaluate OOAD label quality in two stages: (1) self-consistency, where the model labels each snippet multiple times under AB/BA few-shot permutations, and (2) accuracy, where LLM-generated labels are compared against human gold labels on the five SOLID principles (SRP, OCP, LSP, ISP, DIP). For consistency, we report the per-snippet agreement score (fraction of runs that choose the majority label) and its mean over the dataset. For accuracy we compute overall and per-label precision, recall, F1, and accuracy, and visualize results with a 5×5 confusion matrix (gold vs. LLM labels).

Our implementation is decomposed into small Python modules:

[2]https://github.com/fede-ortega/OOAD-accuracy-consistency-LLMs.git

| Metric | Spearman $\rho$ | $p$ | Pearson $r$ | $p$ |
|---|---|---|---|---|
| **Adherence** | 0.279 | 0.050 | 0.278 | 0.051 |
| **Violation** | 0.124 | 0.391 | 0.080 | 0.581 |

Table 1: Correlation between human and LLM's annotations ($N = 50$).

- `consistency_label_eval.py`: builds the system and few-shot prompts, calls the model via `LlamaJudgeClient` for both AB and BA permutations, extracts the JSON `"violation"` with `JsonExtractor`, and computes a label_agreement score for each code snippet.
- `label_agreement_stats.py`: is a lightweight reader that takes the JSON output of the consistency script and reports the mean `label_agreement` over all instances.
- `ooad_label_evaluator.py`: loads gold labels and LLM outputs, chooses one prediction per item via a configurable `LabelSelectionStrategy` (majority vote), computes multi-class metrics (micro/macro precision, recall, F1, accuracy), and prints both numeric results and a confusion matrix; it can also render the matrix as a PNG heatmap.

As in the experiments for accuracy, our code follows OOAD principles in that each class or module has a single concern: prompt, model I/O, agreement, metrics, and higher-level evaluators depend on abstract interfaces (`LlamaJudgeClient`, `LabelSelectionStrategy`). This makes the evaluation pipeline easy to extend (e.g., testing more expensive closed-source models) without modifying core logic.

## Results and Discussion

With $N$ = 50 items, we observe only weak agreement for `adherence_score` (Spearman ($\rho$ = 0.279, $p \approx 0.050$); Pearson ($r$ = 0.278, $p \approx 0.051$)), and negligible, non-significant agreement for `violation_severity` (Spearman ($\rho$ = 0.124, $p$ = 0.391); Pearson ($r$ = 0.080, $p$ = 0.581)). In other words, the LLM's rankings and magnitudes are barely aligned with human judgments for adherence, and not meaningfully aligned for severity. See Table 1 for the statistics and Figures 1–2 for the 5×5 agreement matrices.

Using both Spearman's ($\rho$) and Pearson's ($r$) gives complementary evidence for accuracy of the LLM relative to human Likert ratings. Spearman's ($\rho$) tests whether the rank order of snippets is the same for humans and the model, ideal for ordinal 1–5 scales. Pearson's ($r$), by contrast, evaluates linear agreement in magnitudes under the common interval assumption for 5-point Likert about whether the model's scores are calibrated to human scores (not just ordered similarly), and detecting systematic offsets in the scale. Analyzing them together provides an interpretable picture of model–human agreement on both metrics.

In this experiment, we also study the self-consistency of the model's SOLID labels as a function of the number of runs per permutation, $k$. For each code snippet we query the

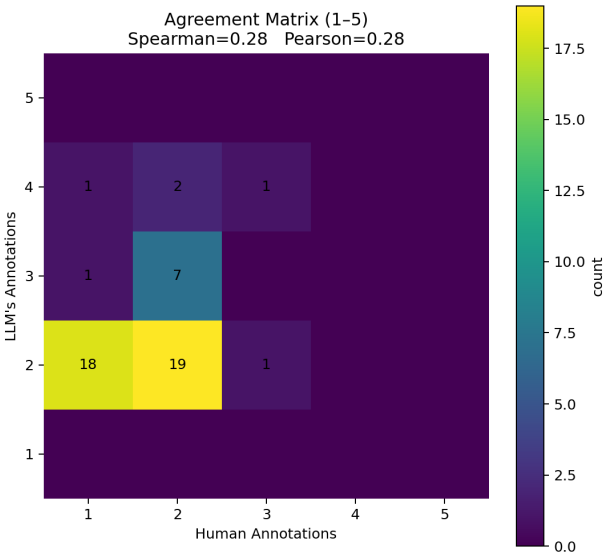


Figure 1: Agreement on `adherence_score` of the 50 annotated examples comparing human and LLM's results.

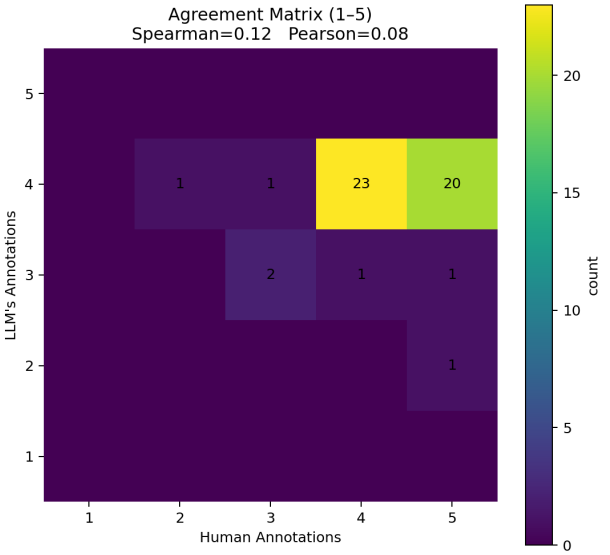


Figure 2: Agreement on `violation_severity` of the 50 annotated examples comparing human and LLM's results.

| $k$ (runs per permutation) | Macro-LA |
|---|---|
| 2 (4 labels) | 0.955 |
| 3 (6 labels) | **0.960** |
| 5 (10 labels) | 0.958 |

Table 2: Macro label agreement for different numbers of runs $k$ per AB/BA permutation. Each item receives $2k$ labels in total.

model under both AB and BA few-shot orderings and repeat each ordering $k$ times, yielding $2k$ labels per snippet. For every item we then compute a `label_agreement` score, defined as the fraction of runs that select the majority label, and `label_agreement_stats.py` reports the mean of this score over the dataset (macro-label agreement).

Table 2 shows that macro-label agreement is consistently high across all settings, ranging from 0.955 to 0.960. Even with $k = 2$ (only 4 labels per snippet), the model is already highly stable in the label it assigns. Increasing $k$ to 3 or 5 slightly changes the estimate but does not materially affect the overall level of agreement, suggesting that our self-consistency measure is robust and that a relatively small number of runs is sufficient to approximate the model's "most likely" label for each snippet. In practice, this means we can use moderate values of $k$ to balance computational cost and reliability without sacrificing consistency.

Table 3 reports macro-averaged precision, recall, F1, and accuracy when we vary the number of runs per permutation $k$. For each code snippet we query the model $k$ times under both AB and BA orderings, aggregate the resulting $2k$ labels by majority vote, and then compare the resulting prediction against the human gold label using `ood_label_evaluator.py`.

Overall, the model achieves solid multi-class performance on the five SOLID labels. Even with $k = 2$ (only four labels per item), macro-precision is already $0.85$, macro-recall and macro-accuracy are $0.78$, and macro-F1 is $0.76$, indicating that the majority-voted prediction is usually correct. Increasing $k$ to 3 does not change the macro metrics, suggesting that the estimator is quite stable. With $k = 5$ (ten labels per item), we obtain our best baseline: macro-precision $0.86$, macro-recall and macro-accuracy $0.80$, and macro-F1 $0.79$. These gains are modest but consistent, showing that additional runs slightly improve robustness while confirming that a relatively small $k$ already captures most of the accuracy benefits.

Figure 3 shows the confusion matrix for the best baseline ($k = 5$). The dataset is balanced (10 instances per label), so diagonal counts correspond directly to accuracy per class. The model is nearly perfect on ISP and SRP (10/10 correct for each) and very strong on LSP and OCP (8/10 correct). Most errors concentrate on DIP, where only 4/10 examples are correctly classified; the remaining DIP cases are mainly confused with OCP and SRP. Despite this weakness on DIP, the diagonal dominates the matrix, which matches the aggregate metrics and indicates that the model generally identifies the violated OOAD principle correctly when evaluated against the human gold annotations.

| $k$ (runs per permutation) | Macro-P | Macro-R | Macro-F1 | Macro-Acc |
|---|---|---|---|---|
| 2 (4 labels) | 0.8529 | 0.78 | 0.7634 | 0.78 |
| 3 (6 labels) | 0.8529 | 0.78 | 0.7634 | 0.78 |
| 5 (10 labels) | **0.8598** | **0.80** | **0.7902** | **0.80** |

Table 3: Macro-averaged precision (P), recall (R), F1, and accuracy as a function of the number of runs $k$ per AB/BA permutation. Each item receives $2k$ labels in total.

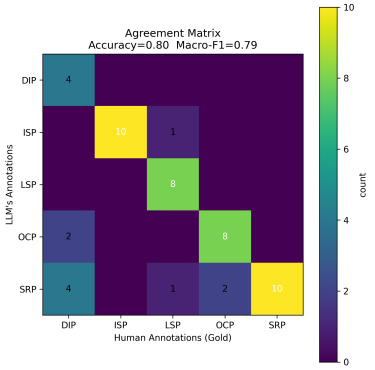


Figure 3: Confusion matrix for the best baseline ($k = 5$). The x-axis shows human (gold) labels and the y-axis shows LLM predictions.

Misclassifications in the confusion matrix are often understandable from an OOAD perspective. Several SOLID principles share overlapping surface patterns, so the same snippet can plausibly be framed as violating more than one principle. For example, code that hard-codes concrete dependencies through conditionals may be annotated as OCP by one annotator, but can also be interpreted as a DIP violation. Similarly, classes with a lot of code that mix responsibilities may be read as SRP violations even when the gold label by the authors of the original corpus emphasizes another aspect such as interface segregation. Because our prompt forces the model to choose exactly one "most severe" violation, the model sometimes selects a different principle than the gold standard. These disagreements highlight that the task is not purely objective: boundaries between SOLID principles can be fuzzy, and even human experts may disagree on which label best captures the dominant design smell in a given snippet.

As future work, errors in labeling depending on the difficulty of each code snippet should be evaluated to better understand, in a quantitative analysis, why this might be happening.

## Conclusion

This work analyzed whether current large language models "understand" object-oriented design quality in the sense of the SOLID principles. We proposed a two-part evaluation framework that measures (1) agreement between LLM and human ratings on adherence and violation severity, and (2) accuracy and self-consistency of categorical SOLID-violation labels under small prompt perturbations. Using a curated subset of Java and Python examples from the *Are We SOLID Yet?* dataset, we added human explanations and Likert-style scores, and implemented a reproducible pipeline that enforces structured JSON outputs and OOAD-oriented prompts.

Our quantitative results indicate that the open Llama model we evaluated captures only a weak signal of SOLID adherence. Correlations with human scores are small for adherence and negligible for violation severity, suggesting poor calibration and ranking alignment. In contrast, the model is highly self-consistent when asked to assign categorical SOLID labels: macro label agreement is roughly $0.96$ across different sampling budgets, and majority voting over repeated AB/BA runs yields moderate accuracy against a gold standard. The confusion matrix reveals that most labels are identified reliably (especially ISP and SRP), while DIP is frequently confused with related principles such as OCP and SRP.

The model exhibits a stable internal notion of which SOLID principle is violated, but this notion does not reliably match human severity or adherence ratings, and it sometimes emphasizes a different principle than the gold label. As a result, current LLMs should not be treated as authoritative judges of OOAD quality, but they may still serve as useful assistants for surfacing candidate design issues, especially when combined with human review. For researchers, our results underscore the importance of evaluating both accuracy and robustness when using LLMs as code-quality evaluators rather than code generators only.

This study has several limitations. Our human annotations cover a relatively small sample ($N = 50$) and come from two annotators, and we focus on a single open model and on SOLID violations. Future work should extend the benchmark with more items and multiple annotators. It will also be important to compare a broader family of models (closed and open source, not only Llama) and study the quality of natural-language explanations. Our dataset and evaluation code provide a starting point for more systematic evaluation frameworks for SOLID principles.

## References

Allamanis, M.; Barr, E. T.; Devanbu, P.; and Sutton, C. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys* 51(4).

Allamanis, M.; Peng, H.; and Sutton, C. 2016. A convolutional attention network for extreme summarization of source code.

Alon, U.; Brody, S.; Levy, O.; and Yahav, E. 2019a.

code2seq: Generating sequences from structured representations of code.

Alon, U.; Zilberstein, M.; Levy, O.; and Yahav, E. 2019b. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* 3(POPL):40:1–40:29.

Baltaji, R.; Pujar, S.; Mandel, L.; Hirzel, M.; Buratti, L.; and Varshney, L. 2025. Cross-lingual transfer in programming languages: An extensive empirical study.

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D. M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; and Amodei, D. 2020. Language models are few-shot learners.

Calanzone, D.; Teso, S.; and Vergari, A. 2025. Logically consistent language models via neuro-symbolic integration. In Yue, Y.; Garg, A.; Peng, N.; Sha, F.; and Yu, R., eds., *International Conference on Representation Learning*, volume 2025, 54495–54522.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating large language models trained on code.

Elazar, Y.; Kassner, N.; Ravfogel, S.; Ravichander, A.; Hovy, E.; Schütze, H.; and Goldberg, Y. 2021. Measuring and improving consistency in pretrained language models. *Transactions of the Association for Computational Linguistics* 9:1012–1031.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A pre-trained model for programming and natural languages. In Cohn, T.; He, Y.; and Liu, Y., eds., *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. Online: Association for Computational Linguistics.

Fu, J.; Ng, S.-K.; Jiang, Z.; and Liu, P. 2023. Gptscore: Evaluate as you desire.

Fu, J.; Ng, S.-K.; Jiang, Z.; and Liu, P. 2024. GPTScore: Evaluate as you desire. In Duh, K.; Gomez, H.; and Bethard, S., eds., *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 6556–6576. Mexico City, Mexico: Association for Computational Linguistics.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Dang, K.; et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Idrisov, B., and Schlippe, T. 2024. Program code generation with generative ais. *Algorithms* 17(2).

Iyer, S.; Konstas, I.; Cheung, A.; and Zettlemoyer, L. 2016. Summarizing source code using a neural attention model. In Erk, K., and Smith, N. A., eds., *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2073–2083. Berlin, Germany: Association for Computational Linguistics.

Khajezade, M.; Wu, J. J.; Fard, F. H.; Rodríguez-Pérez, G.; and Shehata, M. S. 2024. Investigating the efficacy of large language models for code clone detection.

Liu, Y.; Iter, D.; Xu, Y.; Wang, S.; Xu, R.; and Zhu, C. 2023. G-eval: NLG evaluation using gpt-4 with better human alignment. In Bouamor, H.; Pino, J.; and Bali, K., eds., *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2511–2522. Singapore: Association for Computational Linguistics.

Lu, Y.; Bartolo, M.; Moore, A.; Riedel, S.; and Stenetorp, P. 2022. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In Muresan, S.; Nakov, P.; and Villavicencio, A., eds., *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 8086–8098. Dublin, Ireland: Association for Computational Linguistics.

Martin, R. C. 2023. *Functional design: principles, patterns, and practices*. Addison-Wesley Professional.

Martins, G. F.; Firmino, E. C.; and De Mello, V. P. 2024. The use of large language model in code review automation: An examination of enforcing solid principles. In *International Conference on Human-Computer Interaction*, 86–97. Springer.

Meta. 2025. Introducing llama 4: Advancing multimodal intelligence.

Palomba, F.; Bavota, G.; Di Penta, M.; Fasano, F.; Oliveto, R.; and De Lucia, A. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *Proceedings of the 40th International Conference on Software Engineering*, 482–482.

Pehlivan, F.; Ergüzen, A. Ü.; Yengejeh, S. M.; Lami, M.; and Koyuncu, A. 2025. Are we solid yet? an empirical study on prompting llms to detect design principle violations. *arXiv preprint arXiv:2509.03093*.

PYPL. 2025. Popularity of programming language index.

Sadik, A. R., and Govind, S. 2025. Benchmarking llm for code smells detection: Openai gpt-4.0 vs deepseek-v3.

Schindler, C., and Rausch, A. 2025. Llm-based design pattern detection.

Wang, J.; Liang, Y.; Meng, F.; Sun, Z.; Shi, H.; Li, Z.; Xu, J.; Qu, J.; and Zhou, J. 2023a. Is ChatGPT a good NLG evaluator? a preliminary study. In Dong, Y.; Xiao, W.; Wang, L.; Liu, F.; and Carenini, G., eds., *Proceedings of the 4th New Frontiers in Summarization Workshop*, 1–11. Singapore: Association for Computational Linguistics.

Wang, Y.; Lê, H.; Gotmare, A.; Bui, N.; Li, J.; and Hoi, S. 2023b. Codet5+: Open code large language models for code understanding and generation.

Wu, D.; Mu, F.; Shi, L.; Guo, Z.; Liu, K.; Zhuang, W.; Zhong, Y.; and Zhang, L. 2024. ismell: Assembling llms with expert toolsets for code smell detection and refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, 1345–1357. New York, NY, USA: Association for Computing Machinery.

Xu, R.; Cao, J.; Lu, Y.; Wen, M.; Lin, H.; Han, X.; He, B.; Cheung, S.-C.; and Sun, L. 2025. CRUXEVAL-X: A benchmark for multilingual code reasoning, understanding and execution. In Che, W.; Nabende, J.; Shutova, E.; and Pilehvar, M. T., eds., *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 23762–23779. Vienna, Austria: Association for Computational Linguistics.

## Appendix A

```
def build_system_prompt(self) -> str:
    return (
        "You are a strict software design reviewer
        familiar with OOAD principles. "
        "Judge the provided code's adherence to
        SOLID principles with focus on ACCURACY: "
        "how well responsibilities, abstractions,
        and coupling in the code align with SOLID. "
        "Respond ONLY with valid JSON
        using the keys: "
        "explanation, adherence_score,
        violation_severity. "
        f"Limit 'explanation'
        to <= {self.max_words} words. "
        "Use integers for scores (1{5). "
        "No extra prose or code fences."
    )

def build_user_prompt(self, code: str) -> str:
    scale = (
        "Scales:\n"
        "- adherence_score (1{5): 1=largely
        non-adherent; 2=poor; 3=mixed; 4=good;
        5=excellent.\n"
        "- violation_severity (1{5): 1=negligible;
        2=low; 3=moderate; 4=high; 5=critical.\n\n"
        "Evaluate the following code
        for SOLID ACCURACY. "
        "Cite concrete issues (e.g.,
        SRP violations, tight coupling,
        hard-coded deps, magic numbers,
        leaky abstractions). "
        "If mixed, balance strengths and weaknesses.
        Examine the provided examples in the system
        to make judgements. Return ONLY JSON."
    )
    schema_hint = (
        "Return JSON like:\n"
        "{"
        "\"explanation\": \"<=80 words rationale\", "
        "\"adherence_score\": 1, "
        "\"violation_severity\": 3"
        "}"
    )
    return f"{scale}\n\n{schema_hint}\n\n
    CODE TO REVIEW:\n\n{code}"
```

## Appendix B

```
{
    "input": "class PayPalGateway:\n
    def process_payment(self, amount):\n
    print(f\"Processing ${amount} via PayPal\")\n
    return True\n\nclass PaymentProcessor:\n
    def __init__(self):\n
    self.paypal_gateway = PayPalGateway()\n\n
    def process_transaction(self, amount):\n
    return
    self.paypal_gateway.process_payment(amount)",

    "output": "from abc import ABC, abstractmethod
    \n\nclass PaymentGateway(ABC):\n
    @abstractmethod\n
    def process_payment(self, amount):\n
    pass\n\nclass PayPalGateway(PaymentGateway):\n
    def process_payment(self, amount):\n
    print(f\"Processing ${amount} via PayPal\")\n
    return True\n\nclass PaymentProcessor:\n
    def __init__(self, payment_gateway):\n
    self.payment_gateway = payment_gateway\n\n
    def process_transaction(self, amount):\n
    return
    self.payment_gateway.process_payment(amount)",

    "level": "EASY",

    "language": "PYTHON",

    "violation": "DIP",
    "explanation": "PaymentProcessor constructs
    and depends on PayPalGateway directly.
    There is no abstraction or injection;
    the high-level module is tied
    to one concrete implementation.
    Replacing gateways or unit testing
    requires editing PaymentProcessor,
    indicating a straightforward
    DIP and IoC violation.",

    "adherence_score": 1,

    "violation_severity": 4
}...
```