Codecard: Leveraging LLMs to Evaluate AI Model Code Development with the System Cards Framework

Tadesse K. Bahiru and Ioannis A. Kakadiaris

Computational Biomedicine Lab, Department of Computer Science,
University of Houston, TX, USA
{tbahiru,ioannisk}@uh.edu

Abstract. As AI systems are increasingly deployed in high-stakes domains such as finance, healthcare, and education, their responsible use requires scrutiny of how they are engineered, not just how they perform. Although there are now documentation standards for datasets and models, the source code that governs data handling, training, and deployment is rarely audited systematically. To address this gap, this paper introduces Codecard as a part of the System Cards Framework. Codecard is a pipeline that evaluates AI codebases against five criteria: reproducibility, design transparency, documentation quality, privacy, and testing practices. It parses each repository, segments its artifacts, and queries a large language model with structured prompts to produce a scorecard containing numeric ratings, supporting evidence, and targeted recommendations. Codecard's evaluation of 12 public machine learning repositories revealed that only a minority achieved strong reproducibility and documentation, while testing and modular design were consistently weak. These findings show that code-level audits complement existing dataset and model documentation and can guide concrete engineering improvements.

Keywords: Codebase · Codecard · AI code evaluation · LLM-based auditing · Responsible AI.

1 Introduction

Artificial intelligence has rapidly evolved from a research concept into a practical tool shaping real-world decisions in key areas such as finance, healthcare, criminal and justice. Its integration has brought long-standing concerns about bias, fairness, and transparency to a sharper focus [1]. In these high-stakes environments, model accuracy alone is insufficient. Developers and institutions are expected to justify how their systems work, ensure that their output is traceable, and address failures when they arise. As a result, responsible AI development has expanded to include issues such as reproducibility, interpretability, and the enforceability of safeguards, expectations that go well beyond traditional performance metrics [2].

The AI community has made significant progress in addressing some of these challenges. Initiatives such as model cards [3], datasheets for datasets [4], and the EU's trustworthy AI guidelines [5] have helped formalize documentation practices, encouraging developers to provide structured information on how models are built, evaluated, and intended to be used. These tools improve transparency for trained models and datasets, but often overlook a more operational layer of the AI pipeline: the source code. The codebases contain the logic that governs how models are implemented, how training is configured, how data is handled, and how outputs are generated. These decisions directly affect reproducibility, privacy, and auditability, yet most open-source codebases are released without rigorous documentation or validation protocols. Dependency files may be missing or partially specified. Tests are often informal or absent. Privacy controls, if present, are rarely explicit. In practice, code that facilitates high-impact decision-making is frequently shared as little more than a prototype. Although formal methods exist for static code analysis, they focus on syntax correctness, security vulnerabilities, or code quality metrics. What they do not capture is whether a codebase aligns with responsible AI development goals, whether it supports reproducibility, contains meaningful documentation, or enforces privacy-aware practices. Bridging that gap requires a deeper understanding of code structure, design intent, and best practices in AI engineering.

This study introduces Codecard, a repository-level audit that brings the principles of our previous framework, System Cards [6], into the source code. It parses each AI codebase end-to-end, groups related files into coherent segments, and combines large language models with rule-based checks to judge five accountability dimensions: reproducibility, design transparency, documentation, privacy, and testing. The findings are assembled into a scorecard that pairs every numeric score with a concise rationale and recommendations, giving maintainers a clear roadmap for responsible improvement.

The contributions of this paper are summarized below.

- We presented a rubric that maps five criteria from the System Cards Framework to measurable signals within AI code repositories.
- We introduced *Codecard*, a pipeline that combines large language models with deterministic checks to produce structured scorecards containing numeric ratings and actionable recommendations.
- We applied Codecard to evaluate responsible development practices in AI codebases and generate interpretable scorecards that highlight areas for improvement.

The remainder of the paper is organized as follows—section 2 reviews related work. Section 3 sets out the methodology. Section 4 presents the experimental setup and results. Section 5 discusses implications and limitations. Section 6 concludes and notes future directions.

2 Related Work

Recent work in responsible AI has established a suite of governance artifacts for documenting, evaluating, and monitoring models and datasets. For instance,

Model Cards [3] and Datasheets [4] prompt developers to record data provenance and intended uses, while the AI Data Development Scorecard [7] provides a structured rubric for assessing dataset development practices. Tools such as accountability checklists [8] and the AI Incident Database [9] further support post-deployment oversight. While these instruments improve transparency at the model and data level, they leave a key component unexamined: the source code that orchestrates the entire ML pipeline.

Traditional tools designed to analyze this code, however, also have limitations. Static analyzers such as SonarQube [10] and CodeQL [11] are effective for general code hygiene, flagging style violations and security vulnerabilities. Even modern systems using learned heuristics, such as DeepCode [12], do not assess signals unique to machine learning development, such as environment pinning, stochastic seed control, or dataset traceability. More recent approaches leverage large language models for code evaluation, ranging from judging function-level correctness with tools such as CodeEval [13] and CodeJudge [14] to flagging architectural issues with repository-scale agents such as RepoAudit [15]. Importantly, these efforts are designed for general software engineering and rely on ad hoc criteria. As a result, their findings are not explicitly aligned with the core principles of accountability and responsible AI. This leaves a clear gap for a tool that can audit model code through a responsible development lens.

To address this gap, we introduce the Codecard: a scorecard designed to produce an auditable and comparable profile for AI model projects. Unlike a flat list of lint warnings, the scorecard format makes repositories comparable, links each finding to a specific accountability criterion, and reports evidence and potential fixes in a compact and actionable format. The Codecard maps these responsible AI criteria to measurable signals within a repository, combining deterministic checks with content-aware review to create a comprehensive assessment of a project's alignment with best practices.

3 Methodology

We evaluated accountable development practices in machine learning codebases as a part of the System Cards Framework using five criteria: Reproducibility, Code (C131), Design Transparency, Code (C132), Documentation, Code (C133), Privacy, Code (C231), and Testing Cards (C233). Each dimension is evaluated with a defined set of metrics and rubrics. The metric set and their relative weights were established through a two-round delphi with three annotators, two Ph.D. students, and one postdoctoral researcher specializing in responsible AI. A curated set of public repositories was selected for evaluation, and each was processed through a pipeline that extracts source code, configuration files, documentation, and tests. The resulting metrics are aggregated into criterion-level scores and combined into a repository profile that summarizes strengths and weaknesses. Figure 1 shows an overview of the pipeline.

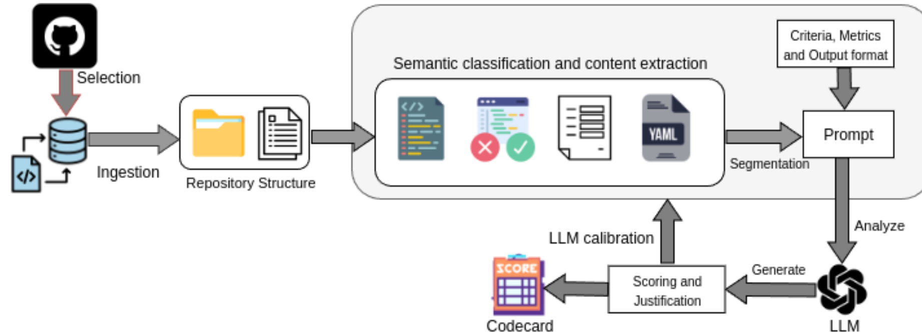

Fig. 1. The Codecard pipeline analyzes codebases with LLMs and produces structured scorecards.

3.1 Evaluation Metrics and Assessment Methods

Table 1 presents the criteria, metrics, and weights derived from the delphi process. The panel assigned weights by judging the relative importance of each signal for accountable code development. Within code reproducibility, environment specification received the highest weight because a pinned environment is the strongest safeguard against dependency drift; dependency pinning followed since version constraints limit drift inside that environment, while reproducibility controls and dataset access were rated lower because they help stabilize runs and clarify inputs but cannot compensate for an ill-specified setup. In design transparency, code complexity and modularity were prioritized since structure governs reviewability and change risk. In contrast, comment density was down-weighted because comments aid comprehension but are often incomplete or outdated. In code documentation, the panel treated README completeness, docstring coverage, model-architecture description, and licensing or citation as equally important since the absence of any one of these breaks reuse or attribution. In code privacy, data handling carried the most weight due to leakage risks in loaders, logs, and temporary files; anonymization and privacy documentation sat in the middle as evidence of safeguards and disclosure; access control was lower because many repositories legitimately avoid credentials, although any exposed secrets are still penalized. In testing cards, unit test presence and test coverage were weighted highest because executed tests provide the most substantial evidence of correctness, CI/CD configuration came next because it enforces tests on every change, and test documentation was lowest since prose alone does not ensure tests are run.

Our evaluation method uses a language model guided by deterministic preprocessing to support consistency and factual accuracy. Quantitative metrics are first extracted using rule-based scripts, including dependency pinning ratios from environment files, docstring coverage via abstract syntax tree parsing, and cyclomatic complexity scores. These metrics, along with selected source code and documentation, are embedded in structured prompts. The language model

then evaluates qualitative aspects that require contextual interpretation, such as the clarity of installation instructions and the presence of privacy disclosures. It synthesizes both the extracted metrics and the contextual content to assign criterion-level scores and provide justifications. This method preserves the reasoning capabilities of the language model while grounding its outputs in pre-validated evidence.

Table 1. Evaluation criteria, metrics, weights, and rubric.

Criteria	Metrics	Weight	Rubric
Reproducibility, Code (C131)	Environment Specification	0.35	1.0: Version-pinned environment files present 0.0: No environment file present.
	Dependency Pinning	0.25	1.0: At least 90% of dependencies are version-pinned. 0.5: Between 50% and 89% are pinned. 0.0: Less than 50% are pinned.
	Reproducibility Controls	0.20	1.0: Seed and determinism clearly set. 0.5: Only seed setting found. 0.0: No controls present.
	Dataset Access	0.20	1.0: Dataset clearly referenced or downloadable. 0.0: No dataset or access method provided.
Design Transparency, Code (C132)	Comment Density	0.30	1.0: At least 20% of code lines are comments. 0.5: Between 10% and 19%. 0.0: Less than 10%.
	Code Complexity	0.35	1.0: 95th percentile cyclomatic complexity is 10 or below. 0.5: Between 11 and 15. 0.0: Any function exceeds 15.
	Modularity	0.35	1.0: At least 95% of files have five or fewer function/class definitions. 0.5: All files have 10 or fewer. 0.0: Any file has more than 10.
Documentation, Code (C133)	README Completeness	0.25	1.0: README contains both Installation and Usage sections. 0.5: Only one of Installation or Usage is present. 0.0: None of the sections are present.
	Docstring Coverage	0.25	1.0: At least 80% of function/class definitions include docstrings. 0.5: Between 50% and 79%. 0.0: Less than 50%.
	Model Architecture Description	0.25	1.0: Detailed structure and components are explained. 0.5: Basic mention or overview is provided. 0.0: No description.
	Licensing and Citation	0.25	1.0: Both license file and citation guidance are provided. 0.5: Only license or citation is included. 0.0: Both License and citation missing.
Privacy, Code (C231)	Anonymization	0.25	1.0: Explicit anonymization applied to sensitive data. 0.5: No sensitive data processed and no unsafe patterns found. 0.0: Sensitive information is exposed or used without anonymization.
	Data Handling	0.30	1.0: Secure temporary storage, no raw data in logs, and verification present. 0.5: No sensitive data processed and no unsafe patterns found. 0.0: Unsafe practices (e.g., hardcoded paths, raw data in logs).
	Privacy Documentation	0.25	1.0: A privacy statement or disclosure of safeguards is provided. 0.5: No sensitive data processed and no unsafe patterns found. 0.0: No privacy-related documentation and potential risks.
	Access Control	0.20	1.0: Sensitive credentials (e.g., API keys, tokens) are externalized and excluded from version control. 0.0: Tokens, API keys, or credentials are hardcoded or exposed in the repository.
Testing Cards (C233)	Test Coverage	0.30	1.0: Coverage report exists and total coverage is 80% or above. 0.5: Report exists and coverage is below 80%. 0.0: No coverage report found.
	Unit Test Presence	0.30	1.0: Test files found in a structured directory. 0.0: No test files found.
	CI/CD Configuration	0.25	1.0: At least one valid CI/CD workflow file found in `.github/workflows/` 0.0: No CI/CD configuration detected in the codebase.
	Test Documentation	0.15	1.0: README or `TESTING.md` explains test goals, commands, and edge cases. 0.0: No testing explanation provided.

3.2 Preprocessing and Artifact Extraction

We extract and normalize the full contents of each repository to prepare it for evaluation. This includes all source code files, configuration scripts, environment declarations, documentation, and testing infrastructure. We collect scripts written in python and Jupyter formats, capture configuration files such as JSON, YAML, and INI, and parse dependency files such as requirements.txt, environ-

ment.yml, and dockerfile. Documentation assets, including README files, contributor guidelines, test instructions, and licenses are also included. The test structure is collected from test folders, test scripts, coverage reports, and continuous integration (CI) configurations. All files are read in plain text format, preserving their original layout, indentation, and filenames. We do not filter or exclude components, ensuring that the evaluation reflects the complete development context. Each file is labeled with its path and functional role and organized in a way that maintains the structure of the codebase. The entire repository is consolidated into a single annotated text file that captures both the content and the organization of the original project. This prepares the codebase for consistent and reproducible analysis by language models in the next stage.

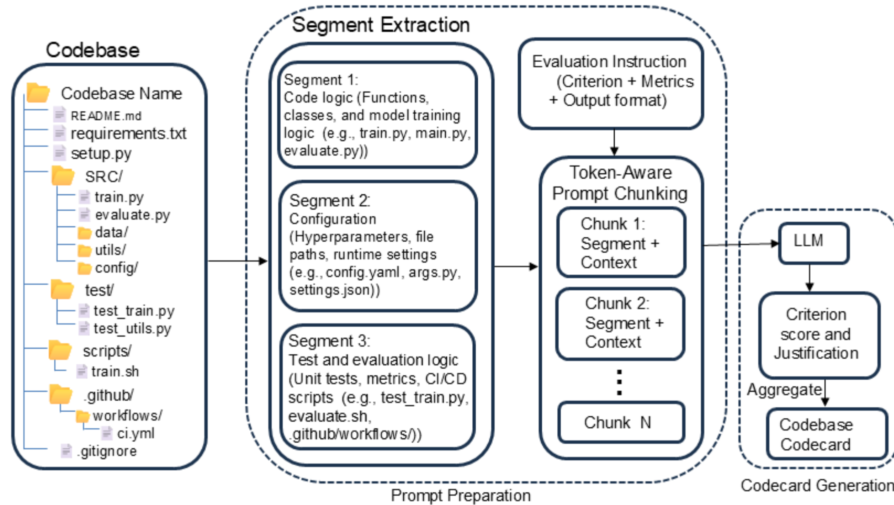

Fig. 2. LLM prompt preparation pipeline illustrating how segmented code, context, and evaluation instructions are bundled into token-aware chunks for consistent and reproducible zero-shot evaluation.

3.3 Prompt Preparation

Semantic-aware segmentation is used to structure long codebases' content in a way that respects the token limitations of LLMs while preserving contextual integrity. Codebase artifacts are first parsed to identify logical segments such as complete functions, configuration blocks, or documentation sections. Each segment is enriched with contextual metadata including the file name, relative path, type, and functional role to maintain semantic continuity throughout the codebase. When the overall content exceeds the model's token window, prompt chunking is applied by bundling coherent segments together with their surrounding structure, such as module hierarchy and relevant dependencies, allowing the

model to retain a holistic understanding. Each prompt is constructed with a standardized structure containing the task introduction, evaluation criterion, and detailed scoring rubric. This structure is then populated with the curated package of pre-computed metrics and relevant code artifacts. We use a zero-shot, chain-of-thought format to guide the model's reasoning process. To ensure reproducible and focused outputs, all evaluations use fixed generation parameters (temperature=0.2, top-p=0.9). As illustrated in Figure 2, our pipeline bundles these metrics and artifacts into token-aware prompts for efficient and scalable evaluation.

3.4 Codecard Generation

Each codebase is documented through a structured Codecard that captures the result of the whole evaluation process. The card contains three main components: metadata (including project name, domain, repository link, directory layout, and evaluation date); a criterion-level breakdown with numeric scores and supporting rationale; and a set of targeted recommendations. Scores for each criterion are computed as weighted averages of the corresponding metric scores, the weights reflecting the practical importance of each development signal in responsible AI engineering. A three-tier color scale is used to interpret the final scores: scores $t_i \geq t_1 = 0.80$ are shown in green, indicating strong alignment with responsible development practices. Scores in the range $t_2 = 0.60 \leq t_i < t_1$ appear in yellow, signaling that the codebase meets baseline standards but has areas needing improvement. And scores $t_i < t_2$ are marked red, reflecting critical deficiencies that undermine transparency, reproducibility, or overall development quality. These thresholds were selected through experimental calibration based on pilot evaluations conducted during method development. The Codecard template, shown in Figure 3, provides a standardized format to present these evaluations.

Metadata

Codebase Name:	[Name of the codebase]
Evaluation Date:	[Date of evaluation]
Domain:	[Domain of the code]
Description:	[Short description of the codebase]
Codebase URL:	[Link to the codebase]
Evaluator LLM:	[LLM Name]

Assessment

Reproducibility (C131):	Score [X.XX] – Color: Explanation
Design Transparency (C132):	Score [X.XX] – Color: Explanation
Documentation (C133):	Score [X.XX] – Color: Explanation
Privacy (C231):	Score [X.XX] – Color: Explanation
Testing Cards (C233):	Score [X.XX] – Color: Explanation

Improvement Suggestions

Reproducibility (C131):	[Suggestion]
Design Transparency (C132):	[Suggestion]
Documentation (C133):	[Suggestion]
Privacy (C231):	[Suggestion]
Testing Cards (C233):	[Suggestion]

Fig. 3. Codecard template with three sections: metadata, assessment scores, and improvement suggestions.

The language model used to generate the evaluations was selected through an expert-guided, human-in-the-loop benchmarking process. Two self-contained

codebases with clear structure and documentation were independently annotated by domain experts using the rubric defined in Section 3.1. Inter-annotator reliability was measured using the intraclass correlation coefficient (ICC [2,3]), and the averaged expert scores served as the human reference. To assess model–human alignment, we applied Lin's Concordance Correlation Coefficient (CCC) [16], which evaluates both correlation and bias to quantify how closely the model's evaluations align with human judgments. For explanation and recommendation alignment, a human-in-the-loop process was used in which domain experts reviewed the model's outputs and provided iterative feedback to refine reasoning quality and interpretive accuracy. The model demonstrating the highest concordance and most coherent human-aligned reasoning was then selected as the default evaluator for Codecard generation.

4 Experiments

4.1 Codebase Selection

We collected 20 open-source Python codebases from GitHub, focusing on machine learning applications in areas such as computer vision, natural language processing, and time-series forecasting. From this initial pool, we selected 12 repositories that showed active maintenance, strong community adoption, and practical engineering relevance. Each has at least 5k stars and 0.5k forks, used a permissive license such as MIT or Apache 2.0, and fell within the processing constraints of language models, with fewer than 100k lines of code and no more than 200 files. We also ensured that every codebase was technically self-contained and included enough documentation to support the full-pipeline evaluation. Table 2 presents an overview of the selected codebases along with metadata.

4.2 Experimental Results

Expert-Annotated Benchmarks. The nanoGPT and CLIP codebases were manually evaluated by the three experts who rated the metrics and served as calibration references for model validation. Inter-rater agreement was high, with an ICC score of 0.98, indicating strong consistency across all five criteria. Where scores differed, the average value was taken to avoid bias toward optimistic or pessimistic extremes. Justifications and improvement suggestions were broadly aligned; when experts raised distinct but valid perspectives, both were retained to capture complementary reasoning. For privacy, scores were applicability-adjusted: if a repository did not handle sensitive or personally identifiable data, the criterion was marked as not applicable and excluded from the macro-average. However, exposed credentials or unsafe handling still incurred penalties. The expert Codecards, shown in Figure 4, served as references for LLM output comparison.

Table 2. Selected codebases for evaluation

Codebase	Stars	Forks	Application-Area	Description
Whisper [17]	81.9k	9.9k	Speech Recognition	Multilingual speech recognition and translation.
Stable-Diffusion [18]	70.6k	10.4k	Text-to-Image Generation	High-res text-to-image synthesis using latent diffusion.
NanoGPT [19]	41.3k	6.8k	Natural Language Processing	Minimal GPT training codebase for educational use and prototyping.
ControlNet [20]	32.3k	2.9k	Image Generation	Conditioning framework to guide Stable Diffusion with structural prompts.
CLIP [21]	29.0k	3.6k	Computer Vision	Language–image pretraining model for zero-shot tasks using contrastive learning.
Segment-Anything [22]	50.2k	5.9k	Image Segmentation	A zero-shot segmentation model by Meta AI
DeepFace [23]	19.1k	2.6k	Face Recognition	Unified face recognition and analysis (emotion, age, race).
DETR [24]	14.4k	2.6k	Computer Vision	Transformer-based end-to-end object detector using bipartite set matching.
AlphaFold [25]	13.5k	2.4k	Bioinformatics	DeepMind's protein structure predictor achieving atomic-level accuracy.
MAE [26]	7.8k	1.3k	Computer Vision	Vision Transformer pretraining via masked autoencoding.
CleanRL [27]	7.1k	0.75k	Reinforcement Learning	Readable single-file RL baselines with reproducible training pipelines.
Informer2020 [28]	5.9k	1.2k	Time-Series Forecasting	Efficient long-series forecasting using ProbSparse transformers.

Metadata
Codebase Name: nanoGPT
Evaluation Date: May 13, 2025
Domain: Language Modeling
Description: Minimal GPT training framework with PyTorch
URL: `github.com/karpathy/nanoGPT`
Evaluator LLM: N.A

Assessment
Reproducibility: Score [0.38] – Red: No environment file or version pinning; lacks seed control
Design Transparency: Score [0.83] – Green: High comment density and modularity, but complexity too high in one function
Documentation: Score [0.63] – Yellow: Good README and model description; lacks docstrings and citation information
Privacy: Score [0.50] – Red: No sensitive data handled, but missing privacy note and secrets management.
Testing Cards: Score [0.00] – Red: No tests or CI/CD present

Improvement Suggestions
Reproducibility: Add a requirements.txt or environment.yml with pinned versions and implement random seed controls in training scripts
Design Transparency: Reduce complexity in high-complexity functions.
Documentation: Add docstrings to all functions and classes, and include citation metadata
Privacy: Add a privacy note, document secrets in `.env.example`, enable CI secret scanning, and include log redaction tests.
Testing Cards: Implement unit tests for core components and integrate CI workflow using GitHub actions

Metadata
Codebase Name: CLIP
Evaluation Date: May 13, 2025
Domain: Vision-Language Models
Description: Contrastive Language-Image Pretraining
URL: `github.com/openai/CLIP`
Evaluator LLM: N.A

Assessment
Reproducibility: Score [0.18] – Red: Dependencies unpinned and no seed controls, though data access is clear
Design Transparency: Score [0.34] – Red: Modular and clean structure; moderate inline documentation
Documentation: Score [0.62] – Yellow: Architecture and usage are documented; function-level docstrings missed
Privacy: Score [0.50] – Red: No sensitive data or unsafe logging, but lacks a privacy note and documented secrets management
Testing Cards: Score [0.55] – Red: CI-enabled unit tests exist, but test coverage and documentation are not found

Improvement Suggestions
Reproducibility: Add explicit seed setting for full experimental reproducibility.
Design Transparency: Refactor large functions into smaller testable components and annotate edge cases.
Documentation: Increase docstring coverage and consolidate model commentary inline
Privacy: Add a privacy note, document secrets in `.env.example`, enable CI secret scanning.
Testing Cards: Add test coverage reporting and explain test assumptions in README or `TESTING.md`

Fig. 4. Benchmark Codecards for (L) nanoGPT and (R) CLIP codebases, presenting criterion-level scores, labels, and improvement suggestions of expert evaluations.

LLM Evaluation. First, we applied the five language models to the two expert-scored codebases, nanoGPT and CLIP, and compared their outputs with manual evaluations. As shown in Table 3, GPT-4o showed strong alignment in both scoring and explanation. It captured key gaps noted by the experts, such as the lack of testing in nanoGPT and the limited docstring coverage in CLIP. While some of its justifications and recommendations were worded differently, the core observations were consistent. In a few places, it even highlighted issues that the human evaluators had missed, showing attention to subtle implementation details. Claude 3 Haiku also performed reasonably well. Its scores were mostly in line with the benchmark, and its rationale showed some overlap with the experts, especially in documentation and privacy. The remaining models, LLaMA 3.3 and Gemma, did not perform as well. They tended to overrate specific criteria, and their explanations often missed the point or introduced suggestions that were not supported by the codebase content.

Table 3. Codebase scores across evaluation criteria for the two codebases.

Codebase	LLM	C131	C132	C133	C231	C233
nanoGPT	GPT-4o	**0.40**	**0.82**	**0.50**	**0.50**	**0.00**
	Claude 3 Haiku	1.00	1.00	0.75	0.50	0.07
	LLaMA 3.3 70B	0.20	1.00	0.75	0.50	0.00
	Gemini 1.5 Pro	0.30	0.68	0.62	0.50	0.00
	Gemma 2	0.90	0.68	1.00	0.65	0.55
CLIP	GPT-4o	**0.20**	**0.35**	**0.62**	**0.50**	**0.55**
	Claude 3 Haiku	0.90	0.35	0.75	0.50	1.00
	LLaMA 3.3 70B	0.00	0.35	0.75	0.50	0.00
	Gemini 1.5 Pro	0.45	0.35	0.75	0.50	0.55
	Gemma 2	0.90	0.68	0.87	0.00	0.25

After identifying GPT-4o as the most consistent and rubric-aligned evaluator, we used it to score the remaining ten codebases and generate a Codecard for each. As reported in Table 4, GPT-4o achieved a Lin's CCC of 0.982, indicating near-perfect agreement with the expert benchmark. Reproducibility and documentation were strongest in repositories such as Stable-Diffusion, ControlNet, and AlphaFold, which included complete environment files and well-structured READMEs. Testing and design transparency remained weak for Segment-Anything, MAE, and CLIP, where GPT-4o accurately identified missing test suites, limited modularity, and sparse inline explanations. DeepFace showed uneven performance, with strong reproducibility but poor modularity and testing due to missing coverage reports. Privacy scores centered around 0.50 across projects, consistent with experts' assessments that the repositories did not handle sensitive data but lacked explicit safeguards. GPT-4o also captured finer implementation issues, such as inconsistent directory structure and minimal error-handling logic in several models, which further differentiated high- and low-performing repositories.

Table 4. Codecard scores of codebases as computed by the GPT-4o.

Codebase	C131	C132	C133	C231	C233
Whisper	0.50	0.18	0.75	0.50	0.67
Stable-diffusion	0.80	0.35	0.75	0.50	0.00
nanoGPT	0.40	1.00	0.50	0.50	0.00
ControlNet	1.00	0.35	0.75	0.50	0.00
CLIP	0.20	0.35	0.63	0.50	0.55
Segment-Anything	0.20	0.17	0.75	0.50	0.00
DeepFace	0.68	0.18	0.63	0.20	0.60
DETR	0.30	0.35	0.62	0.50	0.30
AlphaFold	0.90	0.35	0.88	0.50	0.00
MAE	0.30	0.50	0.50	0.50	0.00
cleanrl	1.00	0.18	0.85	0.50	1.00
Informer2020	0.90	0.35	0.75	0.50	0.00

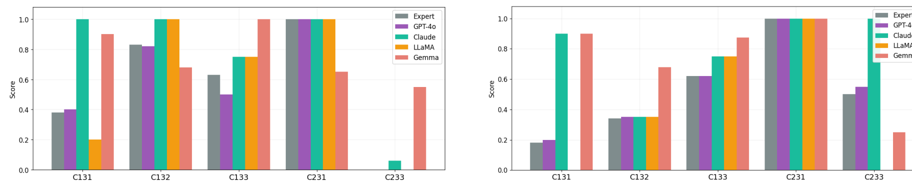

Fig. 5. Comparison of expert and language model evaluations for nanoGPT (L) and CLIP (R) across five criteria (code reproducibility, design transparency, documentation, privacy, and testing cards).

Metadata	
Codebase Name:	Informer2020
Evaluation Date:	2025-05-21
Domain:	Time-Series Forecasting
Description:	Pytorch implementation of Informer for long sequence time-series.
Codebase URL:	https://github.com/zhouhaoyi/Informer2020
Evaluator LLM:	GPT-4o
Assessment	
Reproducibility (C131):	Score: 0.90 – Green: The environment and dependencies are well-specified, with version pinning in both environment.yml and requirements.txt. Datasets are clearly referenced in the README. However, reproducibility controls are only partially implemented, with seed setting present but no explicit determinism.
Documentation (C132):	Score: 0.75 – Yellow: The README includes installation and usage instructions, a detailed model architecture description, and both license and citation information. However, docstring coverage is below expectations.
Design Transparency (C133):	Score: 0.35 – Red: The code structure shows a lack of modularity and high complexity, making the design less understandable and harder to maintain.
Privacy (C231):	Score [0.50] – Red: No sensitive data processed and no unsafe patterns found, but no privacy documentation or safeguards are provided.
Testing Cards (C233):	Score: 0.00 – Red: The repository lacks test coverage reports, unit tests, CI/CD configuration files, and test documentation.
Improvement Suggestions	
Reproducibility (C131):	Implement explicit determinism controls in the codebase.
Documentation (C132):	Add detailed docstrings to all major functions and classes to improve clarity and maintainability.
Design Transparency (C133):	To improve transparency, increase the use of comments and enhance modularity to simplify the code structure and make it more accessible.
Privacy (C231):	Add a privacy note and basic safeguards (e.g., secrets management, log redaction).
Testing Cards (C233):	Implement unit tests, add CI/CD workflows, and include test documentation in the README or a dedicated file.

Fig. 6. GPT-4o generated Codecard for the Informer2020 codebase, summarizing its development quality across five evaluation dimensions.

Figure 5 summarizes agreement patterns across all criteria for nanoGPT and CLIP. Privacy showed the highest consistency across models, while documentation, transparency, reproducibility, and testing revealed noticeable divergence, with weaker models inflating scores based on superficial cues. The generated Codecards (e.g., Figure 6) compile these results by presenting metadata, criterion-level scores, brief justifications, and targeted recommendations, highlighting strengths such as reproducibility in Informer2020 and identifying gaps in modularity, testing, and docstring coverage.

5 Discussion

The evaluation shows that, although many AI repositories provide reproducible environments, most still fall short of accountable development practices. Testing remains the weakest dimension, with few projects offering unit tests, coverage reports, or continuous integration workflows. Design transparency is also limited, and many codebases rely on tightly coupled modules that hinder review and maintenance. Privacy practices are largely implicit. While most repositories avoid sensitive data, they rarely document safeguards, credential handling, or privacy-aware design choices.

Codecard's hybrid structure, which combines deterministic signals with language model reasoning, enabled consistent alignment with expert annotations and captured shortcomings that purely rule-based checks would miss. The study is constrained by the small set of Python-based repositories, and broader application to more diverse, multi-language projects would strengthen the generality of the approach. Expanding the framework to include additional dimensions such as security, efficiency, and dependency risk will further develop Codecard into a more comprehensive auditing tool.

6 Conclusion

This paper presented Codecard, a method for evaluating the quality of AI codebase development using five well-defined criteria. The approach combines a structured scoring rubric with large language models to produce criterion-level assessments with concise explanations and targeted recommendations. Applied to 12 open-source codebases, the results showed that many widely used projects lack essential practices such as environment versioning, modular design, and comprehensive test coverage. Despite these gaps, the method aligned closely with expert annotations and provided reliable diagnostic insight. Each Codecard functions as a compact accountability profile that highlights overlooked engineering concerns. Future work will extend the evaluation to more diverse repositories, including larger and multi-language projects, and incorporate additional criteria such as security and performance.

Acknowledgment

We thank our colleagues, Professors Ryan Kennedy, Andrew C. Michaels, and Lydia B. Tied, for their valuable support. We are also grateful to the CRASA project community advisors team, including Michael O. Adams, Katherine A. Franco, Sharon A. Israel, Jay Jenkins, James Marcella, Paula C. Pineda, and Jani Maselli Wood.

The National Science Foundation, under award number 2131504, partly supported this research. The views and conclusions expressed in this paper are those of the authors and do not necessarily reflect the official policies or endorsements, either expressed or implied, of the National Science Foundation.

The authors acknowledge the use of large language models (LLMs) for language refinement and formatting support. The analysis, interpretations, and conclusions remain solely those of the author.

References

1. E. Ferrara: Fairness and Bias in Artificial Intelligence: A Brief Survey of Sources, Impacts, and Mitigation Strategies. *Science*, **6**(1), 3 (2024).
2. A. Jobin, M. Ienca, E. Vayena: The Global Landscape of AI Ethics Guidelines. *Nature Machine Intelligence*, **1**(9), 389–399 (2019).
3. M. Mitchell, S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I.D. Raji, T. Gebru: Model Cards for Model Reporting. In: *Proc. ACM Conference on Fairness, Accountability, and Transparency*, pp. 220–229. Atlanta, GA, USA (January 29–31, 2019).
4. T. Gebru, J. Morgenstern, B. Vecchione, J.W. Vaughan, H. Wallach, H. Daumé III, K. Crawford: Datasheets for Datasets. *Communications of the ACM*, **64**(12), 86–92 (2021).
5. High-Level Expert Group on AI: Ethics Guidelines for Trustworthy AI. https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai, last accessed 2025/05/11.
6. F. Gursoy, I.A. Kakadiaris: System Cards for AI-Based Decision-Making for Public Policy. *arXiv preprint* arXiv:2203.04754 (2022).
7. T.K. Bahiru, H. Tibebu, I.A. Kakadiaris: AI Data Development: A Scorecard for the System Card Framework. *arXiv preprint*, arXiv:2506.02071 (2025).
8. I.D. Raji, A. Smart, R.N. White, M. Mitchell, T. Gebru, B. Hutchinson, J. S-Loud, D. Theron, P. Barnes: Closing the AI Accountability Gap: Defining an End-to-End Framework for Internal Algorithmic Auditing. In: *Proc. ACM Conference on Fairness, Accountability, and Transparency*, pp. 33–44. Barcelona, Spain (Jan 27–30, 2020).
9. Partnership on AI: AI Incident Database. https://incidentdatabase.ai, last accessed 2025/05/01.
10. SonarSource SA: *SonarQube: Continuous Inspection of Code Quality*. https://www.sonarsource.com/products/sonarqube, last accessed 2025/05/11.
11. GitHub Security Lab: *CodeQL: Code Analysis Engine for Code Security*. https://securitylab.github.com/tools/codeql, last accessed 2025/05/11.
12. Snyk Ltd.: *DeepCode: AI-Powered Code Review*. https://snyk.io/product/developer-security/code-review/, last accessed 2025/05/11.

13. S. Chen, Y. Zhang, X. Lin: CodeEval: Evaluating Code Generation Models Using Multi-Level Semantic Similarity. *arXiv preprint* arXiv:2301.12955 (2023).
14. W. Tong, T. Zhang: CodeJudge: Evaluating Code Generation with Large Language Models. *arXiv preprint* arXiv:2410.02184 (2024).
15. J. Guo, C. Wang, X. Xu, Z. Su, X. Zhang: RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing. In: *Proc. 42nd International Conference on Machine Learning*, Vancouver, Canada (July 13–19, 2025).
16. L. I.-K. Lin: A Concordance Correlation Coefficient to Evaluate Reproducibility. *Biometrics*, **45**(1), 255–268 (1989).
17. A. Radford, J.W. Kim, C. Hallacy, J. Clark, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin: Robust Speech Recognition via Large-Scale Weak Supervision. *OpenAI Technical Report* (2023).
18. R. Rombach, A. Blattmann, D. Lorenz, P. Esser, B. Ommer: High-Resolution Image Synthesis with Latent Diffusion Models. In: *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10684–10695. New Orleans, LA, USA (June 19–24, 2022).
19. A. Karpathy: nanoGPT. https://github.com/karpathy/nanoGPT, last accessed 2025/05/11.
20. L. Zhang, A. Rao, M. Agrawala: Adding Conditional Control to Text-to-Image Diffusion Models. In: *Proc. IEEE/CVF International Conference on Computer Vision*, pp. 3813–3824. Paris, France (October 2–6, 2023).
21. A. Radford, J.W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, I. Sutskever: Learning Transferable Visual Models from Natural Language Supervision. *arXiv preprint* arXiv:2103.00020 (2021).
22. A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, P. Dollár, R. Girshick, K. He, P. Krähenbühl, C.L. Zitnick: Segment Anything. *arXiv preprint* arXiv:2304.02643 (2023).
23. S.I. Serengil, A. Ozpinar: LightFace: A Hybrid Deep Face Recognition Framework. In: *Proc. Innovations in Intelligent Systems and Applications Conference*, pp. 1–5. Istanbul, Turkey (October 15–17, 2020).
24. N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, S. Zagoruyko: End-to-End Object Detection with Transformers. In: *Proc. European Conference on Computer Vision*, vol. 12346, pp. 213–229. Glasgow, UK (August 23–28, 2020).
25. J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S.A.A. Kohl, A.J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A.W. Senior, K. Kavukcuoglu, P. Kohli, D. Hassabis: Highly Accurate Protein Structure Prediction with AlphaFold. *Nature*, **596**(7873), 583–589 (2021).
26. K. He, X. Chen, S. Xie, Y. Li, P. Dollár, R. Girshick: Masked Autoencoders Are Scalable Vision Learners. In: *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 16000–16009. New Orleans, USA (June 19–24, 2022).
27. S. Huang, Z. Huang, S. Yuan, V. Nguyen, W. Zhang, S. Jain: CleanRL: High-Quality Single-File Implementations of Deep Reinforcement Learning Algorithms. In: *NeurIPS Reproducibility Challenge*, New Orleans, USA (2022).
28. H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, W. Zhang: Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting. In: *Proc. AAAI Conference on Artificial Intelligence*, vol. 35, no. 12, pp. 11106–11115. Vancouver, Canada (February 2–9, 2021).