LiCoEval: Evaluating LLMs on License Compliance in Code Generation

Weiwei Xu*, Kai Gao§, Hao He†, Minghui Zhou*‡
*School of Computer Science, Peking University, Beijing, China
*Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
§University of Science and Technology Beijing, Beijing, China
† Carnegie Mellon University, Pittsburgh, USA
xuww@stu.pku.edu.cn, kai.gao@ustb.edu.cn, haohe@andrew.cmu.edu, zhmh@pku.edu.cn

2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE) | 979-8-3315-0569-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICSE55347.2025.00052

***Abstract*—Recent advances in Large Language Models (LLMs) have revolutionized code generation, leading to widespread adoption of AI coding tools by developers. However, LLMs can generate license-protected code without providing the necessary license information, leading to potential intellectual property violations during software production. This paper addresses the critical, yet underexplored, issue of license compliance in LLM-generated code by establishing a benchmark to evaluate the ability of LLMs to provide accurate license information for their generated code. To establish this benchmark, we conduct an empirical study to identify a reasonable standard for "striking similarity" that excludes the possibility of independent creation, indicating a copy relationship between the LLM output and certain open-source code. Based on this standard, we propose LiCoEval, to evaluate the license compliance capabilities of LLMs, i.e., the ability to provide accurate license or copyright information when they generate code with striking similarity to already existing copyrighted code. Using LiCoEval, we evaluate 14 popular LLMs, finding that even top-performing LLMs produce a non-negligible proportion (0.88% to 2.01%) of code strikingly similar to existing open-source implementations. Notably, most LLMs fail to provide accurate license information, particularly for code under copyleft licenses. These findings underscore the urgent need to enhance LLM compliance capabilities in code generation tasks. Our study provides a foundation for future research and development to improve license compliance in AI-assisted software development, contributing to both the protection of open-source software copyrights and the mitigation of legal risks for LLM users.**

I. Introduction

Recent advances in Large Language Models (LLMs) have instigated revolutionary changes in the fields of artificial intelligence, natural language processing, and software engineering [1]. With billions of parameters trained on extensive corpora (both general and software engineering specific), LLMs have demonstrated extraordinary competencies in various software engineering tasks, such as code generation [2]–[4], program repair [5]–[7], and documentation generation [8]. Specifically, LLMs' remarkable code generation capabilities enabled the rapid adoption of AI coding tools in practice. As GitHub reports, 92% of US-based developers are already using AI coding tools both inside and outside of work [9].

However, the widespread utilization of LLMs for code generation has also elicited concerns regarding security [10], [11], privacy [12], [13], and legal issues [14]–[17]. A key issue among these is the potential infringement of intellectual property (IP) rights of a vast number of open-source developers [12], [17], [18]. Due to the fact that LLMs are trained on extensive volumes of open-source code governed by open-source licenses and their inherent ability to recognize and memorize patterns [12], they may generate code snippets that are similar or even identical to those in the training data under certain prompts [12], [13], [17]. Consequently, the use of these generated code must comply with the terms and conditions in the open-source license, concerning how a piece of open-source software (OSS) can be reused, modified, and redistributed [19], [20]. Licenses, which are unavoidably linked to the open-source code, serve a dual purpose: they not only enforce the developers' commitment to sharing, transparency, and openness [13], [14], but also necessitate that those who reuse the code respect and adhere to the terms set by the original authors [21], [22]. For example, under *Apache 2.0* [23], authors grant users perpetual copyright and patent rights, but users must also comply with the corresponding license terms such as including attribution notices and modification statements when redistributing the software.

Despite their extraordinary competencies, LLMs often overlook license information when they memorize and generate code identical to the works of open-source developers. This has sparked a growing wave of concern and discontent among the open-source community [24]–[27]. For example, an open-source developer accuses Copilot of *"emitting large chunks of my copyrighted code, with no attribution, no LGPL license."* [25] More than just ethical concerns, these actions by LLMs may also pose unpredictable legal risks to users. For example, if LLMs produce open-source code without providing the necessary license information, users are naturally unable to utilize such code in legal compliance processes [12], [17]. It should be noted that OpenAI, GitHub, and Microsoft are currently facing lawsuits, as GitHub Copilot [28] has been implicated in reproducing licensed code without compliance with the corresponding license terms [29]. These controversies underscore that the license compliance capabilities of LLMs, i.e., ***the ability to provide accurate license and copyright information during code generation***, play a crucial role in both protecting the IP rights of numerous open-source

‡Minghui Zhou is the corresponding author.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00052

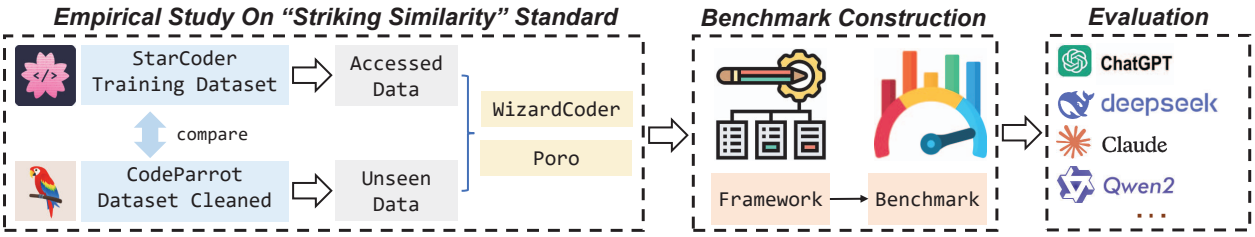


Fig. 1. Overview of this study.

developers and shielding users of such models from unforeseen legal risks. Therefore, it is important to evaluate the license compliance capability of LLMs for code generation tasks.

To the best of our knowledge, it remains unclear how well LLMs perform in terms of license compliance despite the substantial effort devoted to evaluating LLMs' performance on code completion accuracy [30]–[32], robustness [33], [34], security [10], [35], and privacy [12], [36], among others. To fill this knowledge gap, this paper makes the first attempt to evaluate the license compliance capabilities of LLMs by establishing a novel benchmark. This evaluation of LLMs encompasses considerations of not only a technical nature but also legal aspects. Specifically, **one of the most challenging aspects of assessing the compliance capabilities of LLMs is determining whether their outputs, when similar to specific open-source code snippets, are derived from those snippets or are independently created yet merely coincidentally similar.** The former scenario would entail issues of license compliance, whereas the latter does not. A key and universally acknowledged principle used in law to aid in this distinction is "*access and substantial similarity*" [37], [38], which means that potential infringement only occurs when a party has *access* to a work and the resulting product is *substantially similar* to that work. Alternatively, in the absence of evidence indicating direct access, the existence of copying can be established when the degree of similarity between two works is so striking as to preclude the possibility of independently arriving at identical outcomes [37], [39]–[41]. Considering that most LLMs' training data are undisclosed, it is challenging to determine and verify whether an LLM has accessed a specific code snippet. Therefore, we focus on scenarios where the outputs of LLMs are *strikingly similar* to open-source code, to assess their compliance capabilities in this context.

As demonstrated in Figure 1, to establish the benchmark, we first conduct an empirical study to explore a reasonable standard of *striking similarity*, to distinguish between cases where LLMs generate code without prior exposure to similar open-source instances and cases where LLMs memorize open-source code in the training data. Based on the findings of this empirical study, we propose an evaluation benchmark, LiCoEval, for LLMs' license compliance capability evaluation, and then evaluate 14 popular LLMs currently in use for code generation. We find even top-performing code generation LLMs produce a non-negligible proportion (0.88% to 2.01%) of strikingly similar output compared to existing open-source code. Most models fail to provide any license information for code snippets under copyleft licenses, with only `Claude-3.5-sonnet` demonstrating some ability to correctly provide license information for such code. These results highlight the urgent need to improve license compliance capabilities in LLMs used for code generation, particularly in handling code under copyleft licenses and providing accurate license information.

In summary, the contributions of this paper are as follows:

- We conduct an empirical study to explore a preliminary standard for "*striking similarity*" under the legal context of intellectual property infringement and LLM's capabilities in independently generating non-trivial code.
- We design a framework for evaluating the license compliance capabilities of LLMs in code generation and provide the first benchmark for evaluating this capability.
- We evaluate the license compliance capabilities of 14 popular LLMs, providing insight into improving the LLM training process and regulating LLM usage.

II. Background and Related Work

A. License Compliance and IP Infringement

Open-source software authors grant copyrights and, in some cases, patent rights within IP right through open-source licenses. However, this grant is conditional and the licenses specify the obligations to which users must comply [42]–[44]. It is important to note that open-source authors do not oppose or prohibit the reuse of their code; in fact, such reuse aligns precisely with their foundational motives for choosing to open-source their code. Nevertheless, they require users to comply with the stipulations set forth in the licenses when reusing the code. Failure to do so can lead to license non-compliance, ultimately resulting in intellectual property infringements [45].

In the context of LLM code generation, if models output licensed open-source code without providing license information, users reusing this code might face compliance risks. This concern is particularly relevant for companies and organizations, as it could lead to copyright infringement liabilities and potentially result in significant economic compensation and harm to their reputation and operations [46].

In the judicial context, it is crucial to determine whether the content generated by LLMs has a copying relationship with specific open-source code. According to existing legal principles, copying can be established when similarities

between two works are so striking that they preclude the possibility of independently arriving at the same result [37], [39]. However, the definition of "*striking similarity*" remains ambiguous. Courts consider several factors in their analysis, such as the uniqueness, intricacy, or complexity of similar sections, and the appearance of the same errors or mistakes in both works [40]. The ambiguous definition, coupled with the impressive capabilities of LLMs in independently generating non-trivial code snippets, motivates our empirical study to explore reasonable standards for identifying *striking similarity* in the context of LLM generated code.

B. Memorization in LLMs for Code

Research consistently demonstrates that general LLMs tend to memorize content from their training sets, especially in larger models [47], [48], raising significant concerns regarding IP rights violations [18], [48], [49]. This memorization also occurs in LLMs for code, potentially causing compliance issues through unintentional output of licensed code [12], [13], [17]. Recent studies on memorization [12], [13] and IP infringement [17] in LLMs for code, particularly those trained on non-public datasets, typically compare LLM outputs with existing open-source code to detect memorization. However, with undisclosed training sets, it is challenging to definitively attribute this similarity to memorization rather than LLMs' extraordinary generalization competencies. This ambiguity extends to numerous code clone detection methods, which describe the degree of similarity between code snippets from various perspectives, such as textual similarity, call graphs, and other structural features [50]. Yet, we still lack a clear understanding of what degree of similarity is sufficient to constitute "striking similarity" that can exclude the possibility of independent creation.

Furthermore, it remains uncertain whether LLMs can accurately provide the corresponding copyright and license information for code they appear to have memorized. The most relevant work by Yu et al. [17] investigates to what extent LLMs generate licensed code. However, their study was conducted under a problematic assumption that LLMs should not generate licensed code at all, which fundamentally misunderstands the nature of open-source software. Open-source code, by definition, permits reuse under specific conditions; mere reuse does not inherently constitute infringement. The key issue lies in whether the reuse complies with the specified license terms, which, however, is not investigated in their work and requires non-trivial efforts in the LLM context. To address these challenges and uncertainties, we conduct an empirical study to establish a standard of *striking similarity* and build a benchmark to evaluate LLM's compliance capability.

C. Evaluations of LLMs for Code Generation

In the realm of code generation with LLMs, numerous benchmarks have been introduced to evaluate these models' capabilities [30]–[32], [51]. These benchmarks typically focus on generating code snippets from natural language descriptions, employing metrics such as Pass@k [30] to assess the accuracy of the generated code. Furthermore, many studies separately investigate the non-functional properties of LLMs for code [52], including robustness [33], [34], security [10], [35], privacy [12], [13], [36], and explainability [53]. However, to our knowledge, there has been no evaluation focusing on the compliance capabilities of these models.This lack of evaluation is disadvantageous for users seeking models with lower legal risks and also hinders model developers from making targeted improvements in this critical area during the training process, which motivates our work.

TABLE I
MODEL SIZE AND PERFORMANCE(PASS@1(%)) ON THE HUMANEVAL [30] BENCHMARK FOR MODELS TRAINED ON FULLY OPEN-SOURCE DATASETS.

Type	Model	Size	HumanEval	Dataset
General	Poro-34B-chat [54]	34B	37.2	The Stack
Code	Starcoder2-15B-Instruct [55]	15B	72.6	The Stack V2
	WizardCoder-15B-V1.0 [56]	15B	52.4	The Stack
	StarCoder [3]	15B	33.6	The Stack
	Codeparrot [57]	1.5B	3.99	CodeParrot-Clean

III. EMPIRICAL STUDY ON STANDARD OF STRIKING SIMILARITY

A. Research Question

RQ: Where might the reasonable standard of *striking similarity* lie in the context of code generation by LLMs?

A significant challenge in evaluating the compliance capabilities of LLMs is determining whether there exists a copying relationship between the output of LLMs and existing open-source code. Due to the remarkable code generation abilities of LLMs, it is plausible that they can independently generate code that is similar to a specific open-source code snippet, even without prior exposure to it. If the code is independently generated, it clearly does not necessitate the provision of corresponding copyright information.

In accordance with the legal principle of *striking similarity* [37], [39], we aim to explore where the reasonable standard of *striking similarity* might lie for LLMs. The empirical standard explored in this study lays a foundation for our subsequent evaluation framework and benchmark. Our exploration strives to align with copyright law principles in identifying a relatively reasonable standard that can, to some extent, evaluate the compliance risks associated with using certain LLMs. Notably, *our findings are not intended to establish definitive legal boundaries*. Instead, they are intended to serve as a guide for understanding and identifying potential compliance issues, offering insights that can inform future research and development in this rapidly evolving field.

B. Selection of LLMs

We aim to select representative LLMs to explore the *striking similarity* standards, with the following principles:

- High accuracy in code generation tasks, as license compliance has no meaning without accuracy (a model can

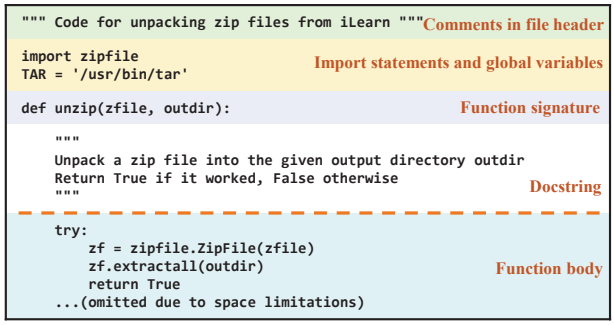


Fig. 2. Structure of function-level code snippet.

generate very chaotic, functionally incorrect code that naturally does not resemble existing open-source code).

- All training data including the instruction tuning data is public. We need to ascertain the data that the model has been exposed to.

We identify three widely used open-sourced datasets for training LLMs on the Huggingface platform, i.e., `The Stack` [58], `The Stack v2` [59], and `CodeParrot Dataset Cleaned` [60]. Based on these datasets, we find five popular LLMs trained on these datasets as listed in Table I. Among these LLMs, we select `WizardCoder-15B-V1.0` as our research subject for two reasons. First, it shows an acceptable performance with a Pass@1 score of 52.4 on the HumanEval benchmark. Second, it has a manageable size (i.e., 15B parameters) under our hardware constraints, which is comparable to, or even larger than the LLMs studied in previous work [12], [13], [17]. `WizardCoder`[1] uses an open-source instruction-following dataset (`Code-Alpaca5` [61]) to fine-tune the `StarCoder` trained on `The Stack` dataset. The complete open-source nature of its training datasets facilitates comprehensive analysis and verification. Despite `Starcoder2-15B-Instruct`'s higher Pass@1 score on HumanEval, we chose `WizardCoder` for our analysis due to its smaller, more manageable dataset (200B tokens vs. 900B tokens in `Starcoder2-15B-Instruct`'s).

C. Experiment setup

In order to detect *striking similarity*, we design an experiment as follows.

First, we construct two distinct groups of code samples, UNSEEN and ACCESSED, to simulate two different scenarios. The first scenario is where the model independently completes the corresponding code, as it has not been exposed to the code from the UNSEEN group before. The second scenario pertains to instances where the generated content cannot be regarded as independently created, potentially implicating a copying relationship. This is attributed to the fact that samples within the ACCESSED group are derived from the model's training dataset. We select two groups of code samples for this study as described in Section III-D1.

[1]In this paper, "`WizardCoder`" refers to "`WizardCoder-15B-V1.0`" in all subsequent mentions.

Second, we construct prompts using the UNSEEN and ACCESSED groups, then instruct `WizardCoder` to complete the code snippets. Our goal is to observe potential differences in similarity when the model generates code for these two distinct groups. As illustrated in Figure 2, we divide the function-level code snippets into five parts: file header comments, import statements and global variables, function signature, docstring, and function body. We combine the first four parts into a prompt with the aim of providing the model with as complete an input context as possible, and then instruct the model generate the function body. We conduct experiments in a one-shot fashion using greedy decoding (temperature set to 0).

Third, we select specific features derived from copyright law principles to characterize *striking similarity* as described in Section III-D2. By analyzing the similarity between the LLM's output and the original code snippets when completing tasks from both groups, we aim to identify a relatively reasonable standard for *striking similarity*. If the generated code meets this standard for *striking similarity*, it can be inferred that the code may not be an independent creation by the model, indicating a possible copying occurrence.

D. Method

1) Construction of Code Samples: Figure 3 illustrates the overview of the construction method. Our code samples originate from two sources: the training set of `WizardCoder`, i.e., `Starcoderdata` [62], and `Codeparrot-clean` dataset [60]. `Starcoderdata`, a subset of `The stack`, comprises 783GB of code spanning across 86 programming languages. A license filtration mechanism was implemented during the data collection process for `Starcoderdata`, which guarantees that all code files within `Starcoderdata` are sourced from the repositories under permissive licenses [3]. `Codeparrot-clean` dataset comprises 5,361,373 Python files from GitHub, including those under restrictive licenses such as *GPL-3.0* [63]. The former is utilized to construct the ACCESSED group that `WizardCoder` has been exposed to, while the latter is used to construct the UNSEEN group that `WizardCoder` has never encountered before.

Given that LLMs are not yet adept at performing code completion beyond the granularity of functions [31], we choose to conduct our experiments in alignment with well-known accuracy benchmarks such as HumanEval [30], specifically at the function-level granularity. Moreover, following the existing research on memorization of LLMs for code [12], [13], our study focuses on Python considering its prevalence [64].

To construct the ACCESSED group, we first extract 74,772,489 function-level code snippets from `Starcoderdata`'s Python files. We further select samples based on the following principles, resulting in 2,628,395 function-level code snippets: 1) the function must have a docstring and the function body must be more than six lines (the median value of all functions' lengths), which aims to provide LLMs with more comprehensive descriptions of the function's capabilities and exclude overly trivial code snippets; 2) the function should not be a class method due to their typically more complex context dependencies [31]; 3) the function-level code snippet

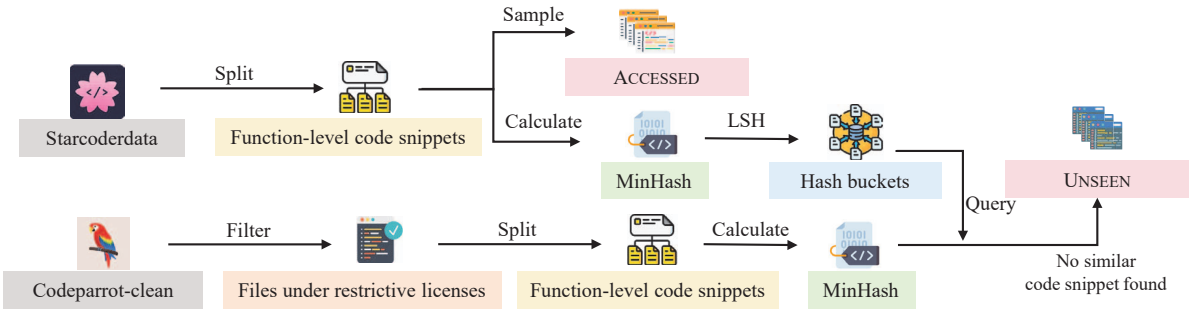


Fig. 3. Overview of code samples construction.

has no syntax errors. Eventually, we randomly sample 10,000 functions as the ACCESSED group that `WizardCoder` has been exposed to.

To construct the UNSEEN group, we select code files from the repositories authorized under restrictive licenses in `Codeparrot-clean` dataset, and segment them into functions. Given that `Starcoderdata` excludes code from such repositories, it is highly probable that `WizardCoder` has not encountered these code snippets in its training set. This makes these functions our preliminary candidate samples. To ensure these functions are truly unseen by `WizardCoder`, we need to compare each function with the entire training set. However, due to the large scale of the dataset, direct comparison of each function with every training sample is impractical. We therefore adopt the deduplication strategy used in `Starcoderdata` [3]. We calculate a MinHash [65] for each function in the training set and use Locality-Sensitive Hashing (LSH) to efficiently map similar functions into the same bucket, using 5-grams and a Jaccard similarity threshold of 0.2 (more stringent than the 0.5 threshold used in `Starcoderdata`) for this process. Then, we compute the MinHash for each function in preliminary candidate samples from `Codeparrot-clean` dataset and query these in the LSH buckets of the training set. Functions without similar matches in the training set are considered final candidates. We apply the same three principles used in constructing ACCESSED group and sample 10,000 samples from 157,273 final candidates to form the UNSEEN group.

Eventually, we obtain the following two groups of code snippets:

- UNSEEN: 10,000 Python function-level code snippets that `WizardCoder` has never been exposed to.
- ACCESSED: 10,000 Python function-level code snippets from the training set of `WizardCoder`.

Table II presents some feature statistics of these samples in two different groups, including the number of prompt lines, the number of function body lines, the cyclomatic complexity [66] of the function, and the number of comments in function body. We compare these features between two groups using Mann-Whitney U Test [67] and Cliff's Delta effect size test [68]. The Mann-Whitney U test determines whether there is a significant difference in their median values [69], [70], while the Cliff's Delta quantifies the extent of this difference [71]. Our analysis

TABLE II
STATISTICS OF FEATURES FROM FUNCTION-LEVEL CODE SNIPPETS IN THE TWO GROUPS AND THE RESULTS OF MANN-WHITNEY U TEST AND CLIFF'S DELTA EFFECT SIZE TEST.

		#prompt_lines	#body_lines	complexity	#comments
UNSEEN	Min	2	7	1	0
	Median	23	17	4	1
	Mean	27.9	28.8	6.5	3.3
	Max	119	2467	317	244
ACCESSED	Min	2	7	1	0
	Median	21	16	4	1
	Mean	25.6	24.6	5.4	2.8
	Max	188	1583	751	326
	p-value	<0.01	<0.01	<0.01	<0.01
	Cliff's δ Eff. Level	-0.07 Negligible	-0.07 Negligible	-0.10 Negligible	-0.03 Negligible

reveals that although the p-value is less than 0.01, the effect size is negligible. This suggests that there are no substantial inherent differences in the code characteristics between the two groups.

2) Features to characterize String Similarity: Considering that copyright law only protects expression rather than ideas [40], [72], we employ fundamental text similarity metrics including BLEU-4 [73], Jaccard similarity based on MinHash [3], and similarity based on edit distance [1], [74] to measure the similarity between the function bodies output by the model and the original implementations in both ACCESSED and UNSEEN groups.

Inspired by judicial considerations of *striking similarity* in fields like music [40] and code's unique traits, we incorporate additional literal features that capture both structural and stylistic elements to characterize *striking similarity*, including the number of function body lines, cyclomatic complexity, and comment similarity. The rationale behind this is as follows:

- Number of function body lines. Given the inherent nature of LLMs in generating code through token-by-token prediction [12], it implies that in scenarios of independent creation, longer code lengths reduce the likelihood of achieving occasional similarity.
- Cyclomatic complexity [66]. Determined by decision points, it increases potential paths through a function. More decision points expand the possibility space, significantly reducing the chance of occasional similarity.

- The similarity of comments. Different developers have unique habits when it comes to writing comments. Unlike code, natural language possesses a higher degree of flexibility [75], making the similarity in comments within code exceedingly rare.

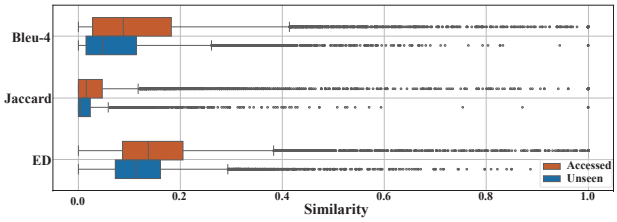


Fig. 4. The similarity between output of WizardCoder and the corresponding open-source code in two groups.

E. Results

Figure 4 shows the similarity between the outputs generated by `WizardCoder` and the corresponding open-source code in two groups. When `WizardCoder` completes code for UNSEEN group, which simulates scenarios of independent creation, it generally produces code with lower similarity to the corresponding open-source code compared to its output for the ACCESSED group, across all similarity metrics. However, we observe some UNSEEN cases with exceptionally high similarity, occasionally reaching a score of 1. These findings suggest that while text similarity metrics can provide indication of differences in generated code between the two scenarios, they are insufficient on their own to definitively determine *striking similarity* and, consequently, non-independent creation.

Figure 5(a) illustrates the distribution of similarity between the generated code and the corresponding open-source code in two groups, in relation to the number of function body lines of the open-source code. We observe that only for ACCESSED group does the LLM frequently generate highly similar code snippets (similarity $>$ 0.6) for functions with longer body lengths ($>$ 10 lines). In contrast, for longer functions from UNSEEN group, the similarity scores are generally lower. As shown in Figure 5(b), a similar phenomenon is observed with cyclomatic complexity. Meanwhile, Figure 5(c) reveals a stark contrast in generated comments between the two groups. For UNSEEN group, the LLM rarely generates comments identical to those in the open-source code. However, for the ACCESSED group, it frequently produces comments that match those in the open-source code. In extreme cases, the identical comments can exceed 15 sentences.

These observations reveal distinct patterns in code generation between ACCESSED and UNSEEN groups. Such distinctions suggest that certain combinations of these features might effectively indicate non-independent creation. To formalize this insight, we seek to establish a quantitative standard for identifying instances of *striking similarity*. Based on our analysis of `WizardCoder`'s code generation results in these two simulated scenarios, we establish an initial standard for *striking similarity* between generated code and the open-source code:

- number of function body lines $>$ 10
- cyclomatic complexity $>$ 3
- text similarity (maximum of the three metrics) $>$ 0.6
- number of identical comments $>$ 0

The first two criteria are attributes of the open-source code snippet, while the latter two describe the relationship between the generated code and the corresponding open-source code snippet. Under this standard, we identify 24 instances, all from ACCESSED group, i.e., the non-independent creation scenario. In the simulated independent creation scenario (UNSEEN group), none of the 10,000 functions generated by `WizardCoder` meets this standard.

F. Validation

Through the above experiments, we establish a preliminary standard for *striking similarity* between LLM outputs and open-source code. This standard, when met, suggests a potential copying relationship rather than independent creation. To validate the preliminary standard, we employ a three-step process: constructing new code samples, expanding our analysis to additional LLMs, and conducting an expert evaluation to assess the standard's validity and applicability.

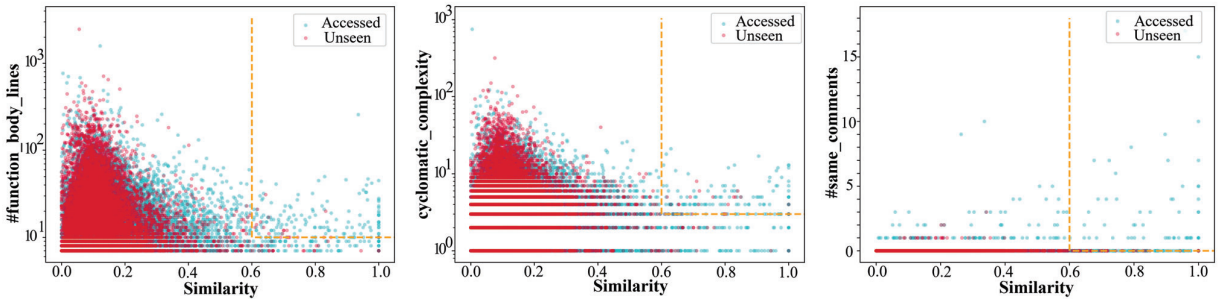
First, using the same methods employed in constructing ACCESSED and UNSEEN groups, we create two additional groups: ACCESSED_EVAL and UNSEEN_EVAL, each containing 10,000 samples that do not overlap with the original ACCESSED and UNSEEN groups. We then utilize `WizardCoder` and `Poro-34B-chat` to complete the functions for these 20,000 samples. We choose `Poro` because it is a general model, yet shares the same code-related training data with `WizardCoder`. This similarity in training data allows us to use the same two groups of samples we constructed, while also testing the standard's applicability to a more general model.

Among the outputs generated by these two models, 31 from `WizardCoder` and 2 from `Poro` meet our standard of *striking similarity*. All 33 samples are from the ACCESSED_EVAL group, yielding a precision of 100%.

To further validate the standard, we assemble a diverse panel: five developers with over six years of coding experience and three lawyers specializing in software intellectual property. This composition ensures both technical and legal perspectives in evaluating these 33 samples. The eight reviewers are tasked with determining whether independent creation can be ruled out based solely on the comparison between the code pairs, without knowledge of their origins. This approach ensures the standard's applicability to outputs from more advanced models, as the evaluation focuses purely on code characteristics. On average, each reviewer identifies 32 out of the 33 sample pairs as cases where independent creation can be excluded, implying a potential coping relationship.

The experts cited three categories of reasons for their judgments:

- Textual and structural similarities: This includes identical or similar variable names, overall textual resemblance, and similarities in code structure (such as indentation and blank lines).



(a) (b) (c)

Fig. 5. The distribution of similarity between the generated code and the corresponding open-source implementations in two groups, in relation to the number of function body lines, cyclomatic complexity, and the number of same comments. The similarity value is the maximum of the three text similarity metrics.

Fig. 6. Examples of cases above and below our striking similarity standard.

- Logical and functional similarities: This encompasses similar approaches to problem-solving, comparable use of specific programming constructs, and similarities in core logic.
- Comment and unique feature similarities: This involves comments similarity (including content and formatting), as well as any distinctive elements like special punctuation or unusual coding patterns.

An example of *striking similarity* is shown in Figure 6 (Example-1). In this example, the striking similarities are evident in multiple aspects: identical variable names (both in case and word choice), matching comment content and placement, and highly similar code structure. As noted by experts, such extensive similarities would be "hard to come by in standalone creations." We generally find that the characteristics the experts are looking for align well with our *striking similarity* standard.

Based on these results, we believe that our proposed standard serves as a reasonable preliminary standard for identifying instances with *striking similarity* that may indicate potential legal risks. While not intended to establish definitive legal guidelines, this standard offers insights for exploring the complex landscape of AI-generated code and associated copyright concerns.

Summary:
Through a comparative analysis of LLM-generated outputs derived from previously accessed and unseen code samples, we establish and validate a preliminary standard of *striking similarity* that effectively excludes the possibility of independent creation.

Implications: (1) Text similarity alone cannot determine non-independent creation in LLM-generated code. LLMs can produce highly similar code even for unseen simple functions. This necessitates using adequately complex code, as defined by our established standard, when constructing LLM evaluation benchmarks. (2) LLMs can memorize and reproduce comments from training data, guiding our development of evaluation frameworks for compliance capabilities. This finding informs our subsequent investigation into LLMs' ability to recall and generate license information in file headers.

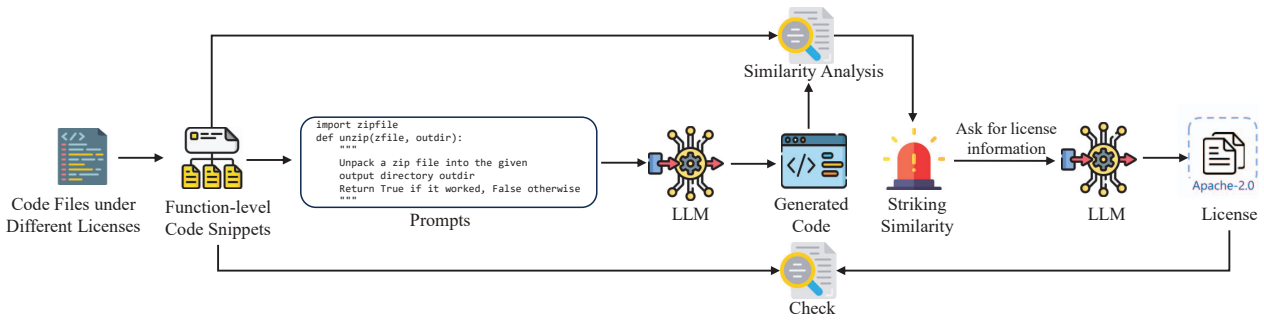


Fig. 7. Overview of the evaluation framework.

IV. Evaluation Framework and Benchmark for LLM License Compliance

A. Evaluation Framework

As illustrated in Figure 7, we propose a framework to evaluate LLMs on license compliance in code generation. This framework is grounded in the empirical findings that LLMs, in non-independent creation scenarios, may generate code strikingly similar to existing implementations, accompanied by identical comments. This observation leads to a crucial question: If LLMs can reproduce code and comments with high fidelity, can they also accurately output the associated license information typically found in file comments? Our framework builds upon this insight, positing that when an LLM generates code strikingly similar to existing code, it should also be capable of providing the corresponding license or copyright information. This principle forms the foundation of our evaluation methodology, linking the generation of strikingly similar code to the ethical and legal responsibility of proper attribution and license compliance.

The framework operates by first constructing a benchmark comprising functions from widely reused code files that have explicit copyright information in their file header comments. The detailed construction method is elaborated in SectionIV-B. The structure of the functions in this benchmark aligns with that as shown in Figure 2. Our prompt consists of the first four components, and we task the LLM to complete the function body. Subsequently, we conduct a similarity analysis between the LLM's output and the corresponding open-source code snippet. If the output meets our established standard for *striking similarity*, we prompt the LLM, in the form of a follow-up inquiry, to output the license information. Finally, we compare the license information provided by the LLM with the actual license of the open-source code snippet.

B. Benchmark LiCoEval

1) Construction Method: We construct our benchmark named LiCoEval by mining code files from the open-source ecosystem that have explicit license information and are widely reused. This process adheres to the standards we previously established. The specific steps are as follows.

Data Source—World of Code. We utilize World of Code (WoC) [76] as the source for constructing the benchmark. WoC is a comprehensive infrastructure for mining version control data across the entire open-source software ecosystem. It aggregates Git objects, including commits, trees, and blobs [77] on platforms like GitHub, Bitbucket, and GitLab. WoC provides several key-value databases that enable efficient querying of relationships between different entities [78], [79]. For instance, the blob-to-project (b2p) database maps each blob to all projects that contain it, enabling efficient tracking of code reuse across projects. For our benchmark, we use version U of WoC, released in October 2021. This version encompasses over 173 million Git repositories, 3.1 billion commits, 12.5 billion trees, and 12.4 billion blobs [80].

Collecting Python Code Files. Our first step utilizes WoC's c2fbb database, which maps each commit to its corresponding commit file name, new blob (post-commit), and old blob (pre-commit). We filter these commits to focus on Python files based on file extensions. By selecting the new blobs associated with these commits and removing duplicates, we obtain a dataset of 700,867,958 unique Python file blobs.

Collecting licensed function-level code snippets. We identify blobs containing explicit license information in their file header comments. Using the b2p database, we quantify each selected blob's occurrence across different projects. We then sort these license-containing blobs in descending order of project count, aiming to identify widely reused code files with clear license information. The presence of license information in file headers indicates clear copyright attribution, while widespread reuse suggests general acknowledgment and acceptance of this copyright information.

We iterate through the sorted blobs in descending order, segmenting each file into function-level code snippets. We then apply a filtering process to select snippets that not only adhere to the three principles described in Section III-D1 but also meet the preconditions of our *striking similarity* standard (function body $>$ 10 lines, complexity $>$ 3, and comment number $>$ 0). From this filtered set, we select the top 10,000 qualifying code snippets as candidate samples. We further refine these 10,000 candidate samples removing code snippets that are explicitly indicated as copied or derived from other sources (identified by keywords like "copied from" or "taken from") and excluding code snippets with dual licenses to ensure license clarity. We also perform deduplication based on

function signatures and docstrings, retaining only one instance where these elements are identical. This rigorous filtering process ultimately yields 4,187 unique function-level code snippets for our LICOEVAL benchmark.

2) Benchmark Characteristics: The detailed characteristics of LICOEVAL are as follows.

License Information Accuracy. We employ a keyword and rule-based method proposed by Xu et al. [19] to identify licenses. To evaluate the accuracy of license information in LICOEVAL, we randomly sample 352 samples (95% confidence level, 5% confidence interval [81]). We manually review the file header comments and function docstrings of these samples to verify the correctness of the license information extracted from the file headers and to check for any exception statements in the docstrings. We find that the license information for all 352 samples is correct. Therefore, we believe that the license information in LICOEVALis highly reliable and can serve as a robust foundation for evaluating the compliance capabilities of LLMs.

License Distribution. Licenses can be categorized into three types based on their level of permissiveness: Permissive, Weak Copyleft, and Strong Copyleft [19]. Copyleft licenses mandate that software which modifies or utilizes existing software must be licensed under the same terms, unless explicitly specified otherwise (e.g., *GPL-3.0*). It is crucial to note that permissive licenses also require adherence to their terms. A license is essentially a conditional authorization [42], and failure to comply with conditions stipulated in permissive licenses can also result in non-compliance issues [22], e.g., Apache-2.0 and CC-BY-4.0 licenses require users to provide a statement of changes made to the original software.

Therefore, in LICOEVAL, we retain a natural distribution of licenses, covering the three different types of licenses. As shown in Figure 8, out of 4,187 function-level code snippets, the number of licenses for Permissive, Weak Copyleft, and Strong Copyleft are 3,073 (73.4%), 369 (8.8%), and 745 (17.8%), respectively. Among permissive licenses, *Apache-2.0* is the most prevalent, accounting for 2,596 snippets (62.0% of the total). For copyleft licenses, the most common are *GPL-2.0-or-later* with 375 snippets (9.0%), *GPL-3.0-or-later* with 220 snippets (5.25%), and *MPL-2.0* with 163 snippets (3.9%).

Code Metrics. Table III presents statistics of LICOEVAL, including the number of lines in function bodies (#body_lines), cyclomatic complexity, the number of comments within function bodies (#comments), and other relevant metrics. Due to our application of preconditions for striking similarity during the selection process, all functions in the benchmark satisfy the criteria of #body_lines > 10, #comments > 0, and cyclomatic_complexity > 3. The metric #reuse_projects indicates the number of projects that have reused the blob containing the function. A higher value suggests more widespread adoption in the open-source ecosystem.

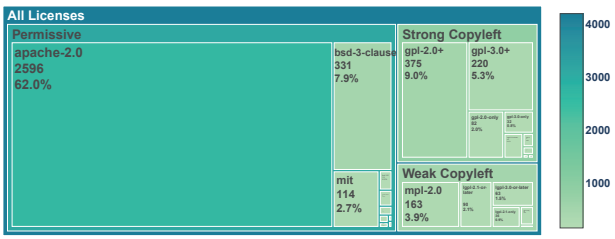


Fig. 8. The distribution of licenses in the benchmark.

TABLE III
STATISTICS OF LICOEVAL.

Metric	Min	Median	Mean	Max	Distribution*
#prompt_lines	2	25	28.5	119	
#prompt_tokens	25	265	310.3	964	
#body_lines	11	28	36.8	398	
#body_tokens	64	356	479.8	5418	
#project_reuse	16	54	245.0	42,476	
#comments	1	4	5.7	159	
cyclomatic_complexity	4	7	9.1	95	

* We increment all values by one to plot the distribution in log-scale.

As evident from Table III, all function-level code snippets in LICOEVAL generally exhibit high complexity and are extensively reused. This indicates that these code snippets are non-trivial while also being widely acknowledged by the open-source community in terms of their copyright information.

V. EVALUATING LLMS ON LICENSE COMPLIANCE

A. Experiment setup

We evaluate 14 popular LLMs that exhibit strong performance in code generation tasks (Pass@1 > 0.5 on HumanEval) using LICOEVAL. Table IV presents the complete list of evaluated LLMs. Throughout the evaluation process, we consistently employ a one-shot approach using greedy decoding (temperature $= 0$).

Effective assessment of license compliance requires a balanced consideration of two critical factors: a model's propensity to generate code strikingly similar to existing implementations, and its accuracy in providing licenses, with particular emphasis on copyleft licenses. This dual focus is essential, as it addresses both the risk of unintended code replication and the model's cognizance of licensing obligations. To quantify this balance, we introduce a novel metric, LICO (License Compliance), calculated as follows:

$$\text{LICO} = \frac{w_1 \cdot (1 - N) + w_2 \cdot Acc_p + w_3 \cdot Acc_c}{w_1 + w_2 + w_3} \quad (1)$$

Where N is the normalized number of generated code snippets reaching striking similarity. Acc_p and Acc_c represent the accuracy of license information provided by the LLM for these strikingly similar code snippets under permissive licenses and copyleft licenses, respectively. We set the weights as $w_1 = 1$, $w_2 = 2$, and $w_3 = 4$, emphasizing copyleft license compliance due to its associated legal risks. For missing accuracy metrics, we use a value of 1, assuming optimal performance in the absence of data. The LICO score ranges from 0 to 1, with higher scores indicating less compliance risks. It's crucial to note that a high LICO score, particularly

TABLE IV
PERFORMANCE OF 14 LLMS ON LICOEVAL. ✓MEANS PUBLICLY AVAILABLE WEIGHTS AND × MEANS UNAVAILABLE WEIGHTS.

	Model	HumanEval	Weights	#striking_sim	Acc	#permissive	Acc_p	#copyleft	Acc_c	LICO
General LLM	GPT-3.5-Turbo [82]	72.6	×	29 (0.69%)	0.72	26	0.81	3	0.0	0.373
	GPT-4-Turbo [82]	85.4	×	25 (0.60%)	0.72	22	0.82	3	0.0	0.376
	GPT-4o [82]	90.2	×	47 (1.12%)	0.74	41	0.85	6	0.0	0.385
	Gemini-1.5-Pro [83]	71.9	×	41 (0.98%)	0.59	39	0.62	2	0.0	0.317
	Claude-3.5-Sonnet [84]	92.0	×	84 (2.01%)	0.69	79	0.71	5	0.4	0.571
	Qwen2-7B-Instruct [85]	79.9	✓	20 (0.48%)	0.95	20	0.95	0	-	0.985
	GLM-4-9B-Chat [86]	71.8	✓	0 (0.0%)	-	-	-	-	-	1.0
	Llama-3-8B-Instruct [87]	62.2	✓	1 (0.02%)	0.0	1	0.0	0	-	0.714
Code LLM	DeepSeek-Coder-V2 [88]	90.2	✓	37 (0.88%)	0.0	36	0.0	1	0.0	0.142
	CodeQwen1.5-7B-Chat [89]	83.5	✓	17 (0.41%)	0.24	17	0.24	0	-	0.781
	StarCoder2-15B-Instruct [55]	72.6	✓	13 (0.31%)	0.23	13	0.23	0	-	0.780
	Codestral-22B-v0.1 [90]	61.5	✓	91 (2.17%)	0.73	87	0.77	4	0.0	0.360
	CodeGemma-7B-IT [91]	56.1	✓	3 (0.07%)	0.33	3	0.33	0	-	0.809
	WizardCoder-Python-13B [56]	64.0	✓	27 (0.64%)	0.04	26	0.04	1	0.0	0.153

a score of 1 in the absence of any strikingly similar cases, is not meaningful if the model's code generation performance is poor. Models producing erroneous or chaotic code may naturally avoid striking similarities, resulting in high LICO scores that lack practical significance.

B. Results

Table IV presents the performance of 14 LLMs on LICOEVAL. We first observe that the three LLMs currently performing best in code generation (`GPT-4o`, `Claude-3.5-Sonnet`, and `DeepSeek-Coder-V2`) show significant variations in their results. They produce 47 (1.12%), 84 (2.01%), and 37 (0.88%) strikingly similar cases, respectively, which are not insignificant proportions, indicating that compliance issues are not uncommon even among top-performing models. Regarding the accuracy of providing license information, `DeepSeek-Coder-V2` performs the poorest, unable to provide any license information, while `GPT-4o` performs the best with an accuracy of 0.74. For the higher-risk copyleft licenses, only `Claude-3.5-Sonnet` demonstrates good performance, which contributes to its highest LICO score. Among other models, `GLM-4-9B-Chat` stands out by not producing any strikingly similar cases, and `Qwen2-7B-Instruct` also demonstrates excellent compliance performance, achieving a license accuracy of 0.95 for its 20 strikingly similar cases, resulting in a LICO score of 0.985. Furthermore, `Codestral-22B-v0.1` generates the highest number of strikingly similar cases at 91, but maintains a relatively good accuracy of 0.73 in providing correct license information. Notably, `WizardCoder-Python-13B` exhibits poor compliance performance, with a LICO score of 0.153, and is almost incapable of providing any correct license information.

For strikingly similar code snippets under higher-risk copyleft licenses, we find that all LLMs perform poorly. Only `Claude-3.5-Sonnet` provides some copyleft license information, while other LLMs have an accuracy of zero. We speculate that some closed-source LLMs may have implemented post-processing steps to avoid acknowledging outputs derived from copyleft-licensed code snippets. It is worth noting that `StarCoder2`, in constructing its training set `The Stack v2`, employed file-level license detection to exclude copyleft-licensed files. This approach may be a significant factor in explaining why the number of strikingly similar cases for copyleft licenses is zero for this LLM. Moreover, we observe that among general LLMs, open-source LLMs demonstrate superior compliance performance compared to closed-source LLMs. This finding suggests a potential correlation between model transparency and license compliance capabilities.

VI. DISCUSSION

A. Implications

In this section, we discuss the implications of our results for LLM providers, LLM users, open-source communities, and legal professionals:

1) LLM providers: Our findings highlight the need for LLM providers to enhance their LLMs' license compliance capabilities. This involves several key areas:

Data Cleaning and License Detection: Our empirical study in Section III-D1 reveals that despite the provider's efforts to exclude code files from copyleft-licensed repositories during `Starcoderdata`'s construction, copyleft-licensed code still persists. This is often due to discrepancies between file-level and repository-level license information [21]. The success of `Starcoder2`'s file-level, fine-grained license detection strategy, which resulted in no strikingly similar cases under copyleft licenses in our evaluation, underscores the importance of meticulous data cleaning processes.

Enhancing License-Code Association: Despite generating non-negligible proportion of strikingly similar code, demonstrating impressive memorization capabilities, many LLMs fail to provide correct license information. This suggests ineffective learning of code-license associations during training. LLM providers should implement more sophisticated preprocessing techniques to strengthen these associations. Considering autoregressive nature of these LLMs, which predict subsequent

content based on preceding information, one potential area for exploration could be the positioning of license information during training. Placing license information after the code snippets might potentially enhance LLMs' ability to associate specific code with its corresponding license, presenting a promising direction for future research.

Addressing Copyleft Information Suppression: In Section V-B, we find that most models tend to avoid providing information about copyleft code usage, likely due to implemented output filters that suppress such information. This approach is problematic as it provides users with incomplete or potentially misleading information, exposing them to higher legal risks. Instead of suppressing copyleft-related outputs, providers should implement post-processing steps that ensure proper attribution and accurate license information are included with generated code, regardless of the license type.

2) LLM users: Our evaluation reveals that many LLMs exhibit poor compliance capabilities. Users, especially commercial entities, must be aware of the potential legal risks associated with AI-generated code. Before incorporating AI-assisted development into their workflows, users should carefully evaluate the compliance capabilities of LLMs. If opting to use LLMs, it is crucial to employ code review tools to verify license compliance of generated code, particularly for code potentially derived from copyleft-licensed sources.

3) Open-source communities: The difficulties LLMs face in accurately providing license information underscore the risk of open-source projects' intellectual property being infringed upon in AI-driven development. While not obligated to facilitate AI training, open-source projects might consider adopting more explicit license declarations to protect their own intellectual property rights effectively. This could include embedding license information in individual files and adopting more granular licensing practices. Furthermore, as AI-assisted coding becomes more prevalent, open-source communities may need to revise their practices and principles. This could include developing guidelines for incorporating and attributing AI-generated code in projects, and establishing clear policies on how their own code should be used in AI training and generation processes.

4) Legal professionals: We find even top-performing code generation LLMs produce a non-negligible proportion (0.88% to 2.01%) of strikingly similar cases. Their varying abilities to provide correct license information (with `GPT-4o` achieving 0.74 accuracy and `DeepSeek-Coder-V2` failing entirely) highlight the complexity of license compliance in AI-generated code. Our study underscores the need for clearer legal frameworks addressing AI-generated code and potential copyright infringements. We demonstrate that it is feasible to characterize non-independent creation in LLM outputs using specific features, opening new avenues for legal analysis. These insights could serve as a reference for establishing more concrete legal standards in this emerging field, aiding legal professionals in cases involving AI-generated code.

B. Threats to validity

1) Internal Validity: Our *striking similarity* standard focuses on precision, potentially overlooking cases where LLMs generate code derived from open-source code but fall below our threshold (e.g., Example-2 in Figure 6). There is no precise legal standard to exclude independent creation and any automated approach would be inherently ambiguous and would not predict precise decisions in a legal context. To mitigate this threat, we opt for a "minimum" standard that emphasizes precision and interpretability, identifying cases that are likely not independently created (e.g., Example-1). This minimum standard aligns well with the qualitative characteristics experts look for when the decide on *striking similarities*. If any of those standards are not met, as in Example-2, it would be unclear whether the two snippets would have been independently created. We must acknowledge that our standard may perform poorly on recall, which is hard to provide evidence in terms of recall due to such inherent ambiguity. Even with such a minimum standard, we are still able to obtain concerning results from state-of-the-art LLMs as shown in Table IV. Furthermore, LiCoEval, consisting of 4,187 samples (code snippets), is a constructed benchmark that may not fully represent the vast diversity of real-world code. This limits our ability to determine if LLMs generate code infringing on open-source software outside our benchmark. Despite this constraint, the substantial findings within our focused dataset indicate that the identified compliance issues are likely prevalent in wider contexts as well.

2) External Validity: our study primarily focused on Python code. While results may vary for other programming languages with different licensing practices, we believe that our evaluation framework and the methodology for constructing the benchmark are general and can be easily extended to other programming languages. Furthermore, We only addressed function-level code completion, overlooking potentially more severe compliance issues at class or project levels. However, our approach provides a foundation for investigating these broader contexts.

VII. Conclusion and Data Availability

In this paper, our main contribution is the development of LiCoEval, the first benchmark for assessing LLMs' license compliance capabilities. To construct this benchmark, we conduct an empirical study on *striking similarity* in LLM-generated code, establishing a preliminary standard for this concept. Using LiCoEval, we perform a evaluation of 14 LLMs, revealing significant license compliance shortcomings.

Although this study is only a preliminary attempt, and much more work is needed beyond the basic requirements of copyright laws, we believe our work could provide valuable insights for improving license compliance in AI-assisted software development. LiCoEval can be accessible at [92].

Acknowledgment

This work is sponsored by the National Natural Science Foundation of China 62332001.

REFERENCES

[1] Y. Dong, X. Jiang, H. Liu, Z. Jin, B. Gu, M. Yang, and G. Li, "Generalization or memorization: Data contamination and trustworthy evaluation for large language models," in *Findings of the Association for Computational Linguistics ACL 2024*. Association for Computational Linguistics, Aug. 2024, pp. 12 039–12 050.

[2] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[3] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[4] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 401–412.

[5] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.

[6] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1482–1494.

[7] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 172–184.

[8] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang *et al.*, "Repoagent: An llm-powered open-source framework for repository-level code documentation generation," *arXiv preprint arXiv:2402.16667*, 2024.

[9] (2023, June) Survey reveals ai's impact on the developer experience. [Online]. Available: https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/

[10] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[11] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2785–2799.

[12] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "Unveiling memorization in code models," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 856–856.

[13] A. Al-Kaswan, M. Izadi, and A. V. Deursen, "Traces of memorisation in large language models for code," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 862–862.

[14] A. Al-Kaswan and M. Izadi, "The (ab) use of open source code to train large language models," in *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*. IEEE, 2023, pp. 9–10.

[15] M. Z. Choksi and D. Goedicke, "Whose text is it anyway? exploring bigcode, intellectual property, and ethics," *arXiv preprint arXiv:2304.02839*, 2023.

[16] P. Henderson, X. Li, D. Jurafsky, T. Hashimoto, M. A. Lemley, and P. Liang, "Foundation models and fair use," *arXiv preprint arXiv:2303.15715*, 2023.

[17] Z. Yu, Y. Wu, N. Zhang, C. Wang, Y. Vorobeychik, and C. Xiao, "Codeipprompt: intellectual property infringement assessment of code language models," in *International Conference on Machine Learning (ICML)*, 2023, pp. 40 373–40 389.

[18] A. Karamolegkou, J. Li, L. Zhou, and A. Søgaard, "Copyright violations and large language models," *arXiv preprint arXiv:2310.13771*, 2023.

[19] W. Xu, H. He, K. Gao, and M. Zhou, "Understanding and remediating open-source license incompatibilities in the pypi ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 178–190.

[20] D. M. German and A. E. Hassan, "License integration patterns: Addressing license mismatches in component-based development," in *2009 IEEE 31st international conference on software engineering*. IEEE, 2009, pp. 188–198.

[21] T. Wolter, A. Barcomb, D. Riehle, and N. Harutyunyan, "Open source license inconsistencies on github," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–23, 2023.

[22] K. Huang, Y. Xia, B. Chen, Z. Zhou, J. Guo, and X. Peng, "Detecting and fixing violations of modification terms in open source licenses during forking," *arXiv preprint arXiv:2310.07991*, 2023.

[23] (2024, March) The Apache 2.0 license. [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0

[24] S. Karpinski. (2021, July) I don't want to say anything but that's not the right license mr copilot. [Online]. Available: https://twitter.com/stefankarpinski/status/1410971061181681674

[25] T. Davis. (2022, October) copilot, with "public code" blocked, emits large chunks of my copyrighted code, with no attribution, no lgpl license. [Online]. Available: https://twitter.com/DocSparse/status/1581461734665367554

[26] C. Green. (2022, June) Explored github copilot,a paid service, to see if it encodes code from repositories with restrictive licenses. [Online]. Available: https://twitter.com/ChrisGr93091552/status/1539731632931803137

[27] Eevee. (2021, June) I'm unclear on how it's not a form of laundering open source code into commercial works. [Online]. Available: https://twitter.com/eevee/status/1410037309848752128

[28] Github. (2022, August) Github copilot. [Online]. Available: https://github.com/features/copilot

[29] (2022, November) Github copilot litigation. [Online]. Available: https://githubcopilotlitigation.com/

[30] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[31] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[32] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[33] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.

[34] M. V. Pour, Z. Li, L. Ma, and H. Hemmati, "A search-based testing framework for deep neural networks of source code embedding," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 36–46.

[35] H. Aghakhani, W. Dai, A. Manoel, X. Fernandes, A. Kharkar, C. Kruegel, G. Vigna, D. Evans, B. Zorn, and R. Sim, "Trojan-puzzle: Covertly poisoning code-suggestion models," *arXiv preprint arXiv:2301.02344*, 2023.

[36] L. Niu, S. Mirza, Z. Maradni, and C. Pöpper, "{CodexLeaks}: Privacy leaks from code generation language models in {GitHub} copilot," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2133–2150.

[37] U. S. C. for the Ninth Circuit. (2022, December) Copying—access and substantial similarity. [Online]. Available: https://www.ce9.uscourts.gov/jury-instructions/node/274

[38] S. Scheffler, E. Tromer, and M. Varia, "Formalizing human ingenuity: A quantitative framework for copyright law's substantial similarity," in *Proceedings of the 2022 Symposium on Computer Science and Law*, 2022, pp. 37–49.

[39] U. C. of Appeals for the Second Circuit. (1946, February) Arnstein v. porter, 154 f.2d 464 (2d cir. 1946). [Online]. Available: https://law.justia.com/cases/federal/appellate-courts/F2/154/464/1478575/

[40] J. R. Autry, "Toward a definition of striking similarity in infringement actions for copyrighted musical works," *J. Intell. Prop. L.*, vol. 10, p. 113, 2002.

[41] S. R. Higgins, "Proving copyright infringement: Will striking similarity make your case," *Suffolk J. Trial & App. Advoc.*, vol. 8, p. 157, 2003.

[42] H. J. Meeker, *Open (Source) for Business: A Practical Guide to Open Source Software Licensing - Third Edition*. United States: Independently Published, 2020.

[43] L. Rosen, "Open source licensing," *Software Freedom and Intellectual Property Law*, 2005.

[44] J. Wu, L. Bao, X. Yang, X. Xia, and X. Hu, "A large-scale empirical study of open source license usage: Practices and challenges," in *2024 IEEE/ACM 21th International Conference on Mining Software Repositories (MSR)*, 2024.

[45] J. Reddy. (2015, Augst) The consequences of violating open source licenses. [Online]. Available: https://btlj.org/2015/11/consequences-violating-open-source-licenses/

[46] W. Xu, X. Wu, R. He, and M. Zhou, "Licenserec: Knowledge based open source license recommendation for oss projects," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2023, pp. 180–183.

[47] C. Zhang, D. Ippolito, K. Lee, M. Jagielski, F. Tramèr, and N. Carlini, "Counterfactual memorization in neural language models," *arXiv preprint arXiv:2112.12938*, 2021.

[48] N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramer, and C. Zhang, "Quantifying memorization across neural language models," *arXiv preprint arXiv:2202.07646*, 2022.

[49] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini, "Deduplicating training data makes language models better," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 8424–8445. [Online]. Available: https://aclanthology.org/2022.acl-long.577

[50] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges," *Journal of Systems and Software*, p. 111796, 2023.

[51] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[52] Z. Yang, Z. Sun, T. Z. Yue, P. Devanbu, and D. Lo, "Robustness, security, privacy, explainability, efficiency, and usability of large language models for code," *arXiv preprint arXiv:2403.07506*, 2024.

[53] R. Aleithan, "Explainable just-in-time bug prediction: Are we there yet?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021, pp. 129–131.

[54] R. Luukkonen, J. Burdge, E. Zosa, A. Talman, V. Komulainen, V. Hatanpää, P. Sarlin, and S. Pyysalo, "Poro 34b and the blessing of multilinguality," *arXiv preprint arXiv:2404.01856*, 2024.

[55] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[56] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," *arXiv preprint arXiv:2306.08568*, 2023.

[57] CodeParrot. (2024, January) Codeparrot. [Online]. Available: https://huggingface.co/codeparrot/codeparrot

[58] Bigcode. (2024, January) The stack. [Online]. Available: https://huggingface.co/datasets/bigcode/the-stack

[59] ——. (2024, January) The stack v2. [Online]. Available: https://huggingface.co/datasets/bigcode/the-stack-v2

[60] (2024, March) Codeparrot dataset cleaned. [Online]. Available: https://huggingface.co/datasets/codeparrot/codeparrot-clean

[61] S. Chaudhary, "Code alpaca: An instruction-following llama model for code generation," https://github.com/sahil280114/codealpaca, 2023.

[62] bigcode. (2024, March) starcoderdata. [Online]. Available: https://huggingface.co/datasets/bigcode/starcoderdata

[63] (2024, April) GNU general public license version 3. [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.html

[64] K. Srinath, "Python–the fastest growing programming language," *International Research Journal of Engineering and Technology*, vol. 4, no. 12, pp. 354–357, 2017.

[65] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *Annual symposium on combinatorial pattern matching*. Springer, 2000, pp. 1–10.

[66] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[67] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[68] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[69] K. Gao, R. He, B. Xie, and M. Zhou, "Characterizing deep learning package supply chains in pypi: Domains, clusters, and disengagement," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–27, 2024.

[70] W. Xiao, H. He, W. Xu, X. Tan, J. Dong, and M. Zhou, "Recommending good first issues in github oss projects," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1830–1842.

[71] W. Xu and X. Zhang, "Multi-granularity code smell detection using deep learning method based on abstract syntax tree." in *SEKE*, 2021, pp. 503–509.

[72] U. C. Office. (2024, March) What does copyright protect? [Online]. Available: https://www.copyright.gov/help/faq/faq-protect.html

[73] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[74] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[75] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*. PMLR, 2016, pp. 2091–2100.

[76] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus, "World of code: an infrastructure for mining the universe of open source vcs data," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 143–154.

[77] S. Chacon and B. Straub, "Git - git objects," https://git-scm.com/book/en/v2/Git-Internals-Git-Objects, 2024.

[78] K. Gao, W. Xu, W. Yang, and M. Zhou, "Pyradar: Towards automatically retrieving and validating source code repository information for pypi packages," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2608–2631, 2024.

[79] K. Gao, Z. Wang, A. Mockus, and M. Zhou, "On the variability of software engineering needs for deep learning: Stages, trends, and application types," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 760–776, 2022.

[80] swsc, "swsc / overview — bitbucket," https://bitbucket.org/swsc/overview/src/master/, 2024.

[81] (2024, March) Sample size calculator. [Online]. Available: https://www.calculator.net/sample-size-calculator.html

[82] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[83] M. Reid, N. Savinov, D. Teplyashin, D. Lepikhin, T. Lillicrap, J.-b. Alayrac, R. Soricut, A. Lazaridou, O. Firat, J. Schrittwieser *et al.*, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," *arXiv preprint arXiv:2403.05530*, 2024.

[84] Anthropic. (2024, June) Claude 3.5 sonnet. [Online]. Available: https://www.anthropic.com/news/claude-3-5-sonnet

[85] Q. Team. (2024, June) Hello qwen2. [Online]. Available: https://qwenlm.github.io/blog/qwen2/

[86] G. Team, A. Zeng, B. Xu, B. Wang, C. Zhang, D. Yin, D. Rojas, G. Feng, H. Zhao, H. Lai *et al.*, "Chatglm: A family of large language models from glm-130b to glm-4 all tools," *arXiv e-prints*, pp. arXiv–2406, 2024.

[87] AI@Meta, "Llama 3 model card," 2024. [Online]. Available: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[88] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma *et al.*, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024.

[89] Q. Team. (2024, April) Code with codeqwen1.5. [Online]. Available: https://qwenlm.github.io/blog/codeqwen1.5/

[90] M. A. team. (2024, May) Codestral: Hello, world! [Online]. Available: https://mistral.ai/news/codestral/

[91] C. Team, "Codegemma: Open code models based on gemma," *arXiv preprint arXiv:2406.11409*, 2024.

[92] (2024, Augest) Licoeval. [Online]. Available: https://github.com/osslab-pku/LiCoEval