When AI Coding Assistants Leak Training Data: A Study of LLM Memorization in Code Generation

Xiaoyu Cheng
xiaoyu.cheng@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Kundi Yao*
kundi.yao@ontariotechu.ca
Ontario Tech University
Oshawa, ON, Canada

Pengyu Nie
pynie@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Weiyi Shang
wshang@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Abstract

Large Language Models (LLMs) for code generation risk memorizing and reproducing sensitive training data, including licensed code and proprietary information. We investigate memorization behavior in recent open-weight LLMs in code generation using a two-stage memorization evaluation pipeline, which combines a similarity-based extractability filter with a targeted data extraction attack. We evaluate four models (StarCoder2-3B, StarCoder2-7B, Llama3-8B, and DeepSeek-R1-distilled-Llama-8B) on a custom dataset of 30,000+ Python files. Our results reveal memorization rates of 42-64%, with code-specialized models exhibiting higher rates than general-purpose models. Categorical analysis shows that repetitive content (license headers, documentation) is memorized at rates up to 70%, while complex code exhibits lower susceptibility. Notably, realistic code completion scenarios trigger unintentional memorization in 13-14% of cases, posing practical risks for AI coding assistants. We demonstrate that knowledge distillation reduces extraction rates by approximately 19%, offering a cost-effective mitigation approach. Our findings confirm that memorization persists in modern LLMs and is influenced more by a complex interplay of training domain, dataset composition, architectural choices, and content characteristics, rather than parameter count alone.

CCS Concepts

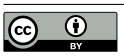
• **Security and privacy** → **Software and application security**; • **Software and its engineering** → *Software development techniques*; • **Computing methodologies** → **Artificial intelligence**.

Keywords

Large Language Models (LLMs), Code Memorization, Code Generation, Memorization Evaluation, Software Security and Privacy

ACM Reference Format:
Xiaoyu Cheng, Kundi Yao, Pengyu Nie, and Weiyi Shang. 2026. When AI Coding Assistants Leak Training Data: A Study of LLM Memorization in Code Generation. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware '26), July 6–7, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3805760.3814902

*Corresponding author



AIware '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2601-9/2026/07
https://doi.org/10.1145/3805760.3814902

1 Introduction

Large language models (LLMs) demonstrate remarkable proficiency in code-related tasks, including generation, summarization, and completion [17]. These models are typically trained on vast datasets from publicly available sources, including library databases, discussion forums, and GitHub repositories [9]. However, this extensive training can lead to unintended memorization, where models reproduce verbatim segments from their training data [4]. This phenomenon poses serious risks, including privacy breaches through exposure of personal identifiable information (PII), intellectual property violations when licensed code is reproduced without attribution [28], and performance degradation due to overfitting [7].

The severity of memorization in LLMs applied to coding tasks is particularly concerning. Recent studies have shown that models can extract sensitive information such as API keys, authentication tokens, and proprietary algorithms [28]. More alarmingly, even with benign prompts in realistic code completion scenarios, models may inadvertently reproduce memorized training sequences [21]. These risks are amplified as LLMs become increasingly integrated into developer workflows through tools like GitHub Copilot and other AI-assisted coding platforms.

Previous research has established pipelines combining membership inference attacks (MIA) and data extraction attacks to reveal and quantify memorization [2]. However, these studies have primarily focused on older models and have not systematically examined how memorization varies across different code categories or whether recent models exhibit similar vulnerabilities. Furthermore, while various mitigation strategies have been proposed, including dataset deduplication [16], differential privacy, and goldfish loss [12], each involves significant trade-offs in cost or performance. Knowledge distillation has emerged as a promising alternative that may reduce memorization while preserving model capabilities [6], but its effectiveness for LLMs applied to coding tasks remains underexplored.

In this work, we investigate memorization behavior in recent open-weight LLMs in code generation and evaluate knowledge distillation as a mitigation approach. We address three research questions: (1) Does the existing memorization-revealing pipeline remain

effective on LLMs in code generation? (2) How do content categories and prompt characteristics affect memorization during code generation? (3) Can knowledge distillation reduce memorization iin these models? We evaluate StarCoder2-3B, StarCoder2-7B, and Llama3-8B using a dataset of over 30,000 Python code sequences, categorizing them into five types (license, documentation, dictionaries, code logic, and testing). Our findings show memorization rates ranging from 42% to 64%, with license and documentation categories most susceptible. Realistic code completion prompts can trigger memorization at rates of 13-14%. Notably, a distilled version of Llama3-8B reduces extraction rates by approximately 20%, demonstrating knowledge distillation as a viable mitigation strategy.

The main contributions of this paper are:

- A systematic evaluation of memorization in recent open-weight LLMs in code generation using an established memorization evaluation pipeline.
- An analysis of memorization patterns across five code categories and realistic usage scenarios.
- Empirical evidence that knowledge distillation can reduce memorization rates while maintaining model performance.
- A curated dataset and evaluation methodology for future memorization research on LLMs applied to code generation.

Paper organization. The remainder of this paper is organized as follows: Section 2 reviews background on memorization and related work. Section 3 describes our research methodology and experimental setup. Section 4 presents our findings. Section 5 discusses threats to validity, and Section 6 concludes this study.

2 Background and Related Works

Memorization and Risks Generative models often memorize verbatim segments of training data instead of generalizing patterns [3]. This behavior persists across architectures and is particularly severe in code models; for instance, thousands of memorized snippets have been extracted from recent outputs [28]. Such memorization poses critical risks: *privacy leakage* of PII and API keys [4, 25], *intellectual property violations* of licensed code [26, 28], and *performance degradation* due to overfitting [7, 11]. Attackers can exploit this via Membership Inference Attacks (MIA) [24] and data extraction pipelines [1, 2], which have also recently been adapted to evaluate memorization dynamics across model fine-tuning stages [22]. While contemporary investigations have focused heavily on token-level secret memorization [20] or membership inference through targeted adversarial prompting [15], code also exhibits distinct category-specific memorization patterns due to its structural predictability [13]. In contrast to purely adversarial setups, we demonstrate that even benign prompts can trigger unintentional data leakage in realistic scenarios [21].

Mitigation and Distillation Existing defenses against memorization involve significant trade-offs. *Deduplication* reduces near-duplicates but cannot eliminate unique memorized sequences [3, 16]; *Differential privacy* often degrades model utility [23]; and *Goldfish loss* incurs training overhead [12]. Knowledge distillation, which transfers behavior from a teacher to a smaller student model [14], has emerged as a promising mitigation strategy [6, 27]. While recent work by Dong et al. [6] explores self-distillation, where a model acts as its own teacher through logit adjustment, our evaluation distinctly assesses a reasoning-distilled student model. This approach allows us to observe how properties inherited from a distinctly different teacher distribution and reinforcement learning objective impact similarity-based extractability. However, the broader effectiveness of distillation paradigms specifically for code-generating LLMs remains underexplored, which motivates our systematic evaluation.

3 Approach

To systematically investigate memorization, we adopt a two-stage memorization evaluation pipeline, which combines a Similarity-based Extractability Filter followed by a targeted Data Extraction Attack [2], as illustrated in Figure 1.

Dataset and Models. We constructed a diverse evaluation corpus by sampling over 30,000 Python files from BigQuery's GitHub dataset [8]. Full sampling, processing, and categorization scripts are documented in our replication package. From these files, we extracted random 150-token sequences, each split into equal 50-token parts: *pre-prefix*, *prefix*, and *suffix* (ground truth). We utilize a 50-token prefix threshold to align with prior benchmarking standards [2]. This length provides sufficient syntactic context for the model to orient itself while remaining short enough to rigorously test for exact memorized recall rather than generalized task completion. We acknowledge that extraction rates may be sensitive to this threshold, as longer prefixes generally increase the probability of verbatim generation. We evaluate three base models to assess the impact of architecture and training data: StarCoder2-3B, StarCoder2-7B (code-specialized) [18], and Llama3-8B (general-purpose) [10]. Furthermore, to evaluate mitigation strategies, we include a distilled variant: DeepSeek-R1-distilled-Llama-8B [5].

The two-stage memorization evaluation pipeline. In *Stage 1 (Similarity-based Extractability Filter)*, we prompt each model with a 100-token context (pre-prefix + prefix). Sequences whose generated output exhibits high similarity to the original suffix are flagged as likely candidates for training data. In *Stage 2 (Data Extraction Attack)*, we perform a guided extraction on these candidates using only the 50-token *prefix* to test the model's ability to reproduce training data with minimal context. While our pipeline utilizes a similarity-based filter rather than a verifiable membership classifier, we follow established literature [2] in using the term memorization rate to quantify the proportion of successful, highly similar extractions, evaluated via Exact Match (EM) and BLEU-4 scores.

4 Results

RQ1: Does the existing memorization-revealing pipeline remain effective on LLMs in code generation?

To establish baseline memorization behaviors, we apply the complete two-stage pipeline to both code-specialized (StarCoder2-3B/7B) and general-purpose (Llama3-8B) models. As defined in our methodology, we measure successful extractions using Exact Match (EM) and BLEU-4 scores between the generated completions and the ground-truth suffixes. Table 1 (left columns) summarizes the overall extraction rates.

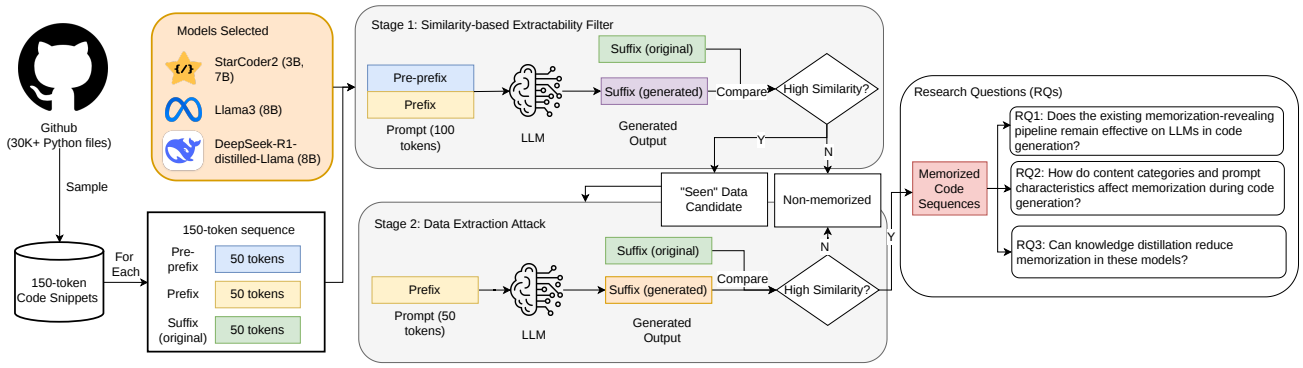


Figure 1: Overview of this research.

Table 1: Overall and Categorical Extraction Rates Across LLMs

Model	Overall		Categorical Extraction Rate (%)				
	EM (%)	BLEU-4	License	Docs	Dicts	Code	Testing
StarCoder2-3B	64.16	0.848	72	64	62	39	22
StarCoder2-7B	62.00	0.836	70	63	63	39	21
LLaMA 3-8B	42.24	0.821	45	20	55	26	11

Memorization persists across modern LLMs. All three models exhibited substantial memorization, with EM rates ranging from 42.24% to 64.16%. This confirms that the pipeline designed for earlier models [2] remains effective for recent open-weight LLMs, and that unintended memorization continues to be an intrinsic behavior of these systems.

Code-specialized models memorize more training data. The StarCoder2 variants showed significantly higher extraction rates (62-64%) compared to Llama3-8B (42%) ($p < 0.001$, Chi-square test). While this difference strongly correlates with their training domains, we need to acknowledge that varying architectures, distinct dataset compositions, and different training procedures between the StarCoder and Llama families all contribute to the observed differences in extractability. This specialization appears to amplify memorization of domain-specific content, as the models encounter repeated code patterns more frequently during training.

Parameter size has limited impact at this scale. Interestingly, increasing parameters from 3B to 7B in StarCoder2 did not correspond to higher memorization rates, with both models demonstrating comparable extraction rates. This suggests that at this scale, architectural choices and training data characteristics may dominate over parameter count in determining memorization behavior. This finding contrasts with earlier work showing monotonic increases in memorization with model size [4], suggesting that the relationship between scale and memorization may be more nuanced in modern architectures.

> **RQ1 Findings:** Memorization persists in recent open-weight LLMs, with memorization rates varying due to a complex interplay of training domain, dataset composition, and architectural choices, rather than parameter count alone. Furthermore, our two-stage similarity-based evaluation pipeline remains effective for revealing these extraction vulnerabilities in recent models.

RQ2: How do content categories and prompt characteristics affect memorization during code generation?

We investigate how memorization varies across different types of code content and whether realistic developer workflows can trigger unintentional data leakage.

1) Categorical Memorization. Following prior work [2], we categorized memorized sequences into five functional types: license headers, documentation, data structures (dicts), code logic, and testing code. Table 1 details these categorical extraction rates.

Repetitive boilerplate content exhibits the highest memorization rates. Across both StarCoder2 models, license headers exhibited the highest memorization rates (~70%), closely followed by documentation. This aligns with the mechanics of LLM training: legal boilerplates and standardized docstrings act as highly predictable anchors across thousands of GitHub repositories, causing models to heavily overfit these sequences.

Data structures pose a potential vector for sensitive data exposure due to high extraction rates. An alarming finding is the high extraction rate of data structures (Dicts), particularly in Llama3-8B (55%). Since dictionaries frequently store configuration mappings, hardcoded credentials, or lookup tables, this high memorization rate exposes a severe attack vector for extracting sensitive project-specific data or Personally Identifiable Information (PII).

Complex testing code resists memorization despite its repetitive framework syntax. Interestingly, testing code exhibited the absolute lowest memorization rates (11-22%). While framework syntax (e.g., `pytest`, `JUnit`) is repetitive, actual assertions, mock data, and unit logic are hyper-specific to individual repositories. This high structural variability prevents models from effectively memorizing exact test implementations.

2) Unintentional Memorization in Realistic Scenarios. Standard data extraction pipelines rely on arbitrary fixed-length prefixes, which do not reflect real-world usage. To assess the true threat to developers, we investigated whether benign, realistic prompts could trigger similar extraction behaviors. To simulate this, we replaced the arbitrary prefix with natural code-completion contexts. Specifically, we sampled sequences exclusively from the "code logic" category of our evaluation corpus. For each sequence, we concatenated up to 300 characters of preceding file context with the target

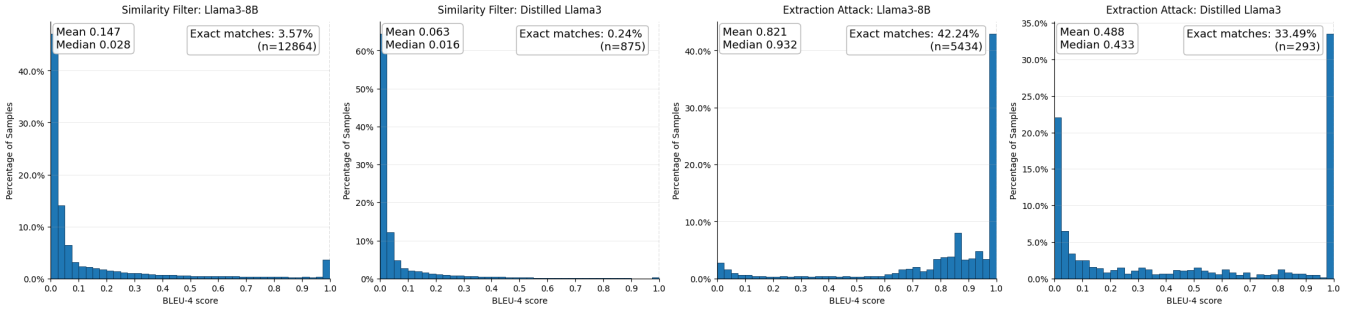


Figure 2: Similarity-based extractability filtering (left) and data extraction attack results (right) for the base Llama3-8B model compared to its DeepSeek-R1 distilled variant.

method's signature and its docstring. To simulate a standard AI coding assistant context window, we truncated this string to use the last 150 tokens as our natural prompt. We evaluated 3,158 such sequences on StarCoder2-7B and 1,042 sequences on Llama3-8B. The models were allowed to generate up to 50 new tokens (capped at the true method body length), and the outputs were evaluated against the first 50 tokens of the original method body via BLEU-4. This approach rigorously tests whether providing standard, realistic semantic context triggers unintentional data leakage.

Realistic prompts trigger substantial unintentional memorization. Without any adversarial intent or malicious prompting, StarCoder2-7B unintentionally reproduced highly similar output (BLEU > 0.85) in 13.8% of its completions. Llama3-8B exhibited similar behavior, leaking exact data in 12.7% of cases.

AI coding assistants inadvertently increase potential exposure to licensing compliance risks. These rates are highly concerning because they represent an invisible threat in modern software engineering workflows. The prevalence of unintentional memorization suggests that developers using AI coding assistants (e.g., GitHub Copilot) could inadvertently insert licensed, copyrighted, or proprietary code snippets into their own codebases simply by writing standard function descriptions. This exposes both individual developers and organizations to significant intellectual property violations and licensing compliance risks without their knowledge.

> **RQ2 Findings:** Memorization is strongly dictated by content structure: boilerplates (licenses) and data structures (dicts) are highly vulnerable, whereas highly coupled logic (tests) resists memorization. Alarmingly, normal developer workflows trigger unintentional verbatim leakage in 13-14% of cases, posing a severe compliance trap.

RQ3: Can knowledge distillation reduce memorization in these models?

To evaluate knowledge distillation as a potential mitigation strategy, we subjected Llama3-8B and its distilled variant, DeepSeek-R1-distilled-Llama-8B, to identical two-stage similarity-based evaluation pipelines. Because DeepSeek-R1-distilled-Llama-8B was distilled from a reasoning-oriented teacher using reinforcement learning, this setup does not isolate standard knowledge distillation as a singular variable. Rather, it allows us to evaluate how the properties inherited from this distinct training objective and teacher distribution impact similarity-based extractability compared to the base model.

Distillation significantly reduces extraction rates. The distilled model exhibited a notably lower extraction rate (34.17%) compared to the original Llama3-8B (42.24%), representing a statistically significant reduction of approximately 19% in similarity-based extraction ($p < 0.001$, Chi-square test). Rather than proving that standard knowledge distillation universally acts as a standalone unlearning mechanism, this represents an empirical observation that this specific reasoning-oriented distilled model exhibits lower extraction rates under our pipeline.

The distilled model exhibits reduced similarity-based extractability. As we observe in Figure 2, the distilled model generated far fewer positive matches during the similarity-based extractability filtering stage. This indicates that training sequences are less likely to be flagged by our similarity heuristic. However, rather than proving that standard knowledge distillation universally acts as a mechanism that forces a model to "unlearn" specific training instances, this finding represents an empirical observation of this specific reasoning-oriented distilled model. The lower extraction rate observed under our evaluation pipeline is likely influenced by properties inherited from the teacher model's distinct distribution and its reinforcement learning objectives, rather than distillation acting alone.

Distillation is cost-effective but incomplete. Unlike resource-intensive defenses such as differential privacy or dataset deduplication, knowledge distillation is straightforward to implement and often yields faster, more efficient models. However, the distilled model still exhibited a 34% extraction rate, indicating that distillation does not eliminate memorization entirely. This suggests that distillation should be considered as one component in a multi-layered mitigation strategy rather than a standalone solution.

> **RQ3 Findings:** The reasoning-oriented DeepSeek-R1-distilled-Llama-8B exhibits an approximate 19% reduction in similarity-based extraction compared to its base model. However, this reduction is likely influenced by the specific dynamics of the teacher model rather than standard distillation universally acting as an unlearning mechanism. Furthermore, this approach does not eliminate memorization entirely and should be combined with other defenses for comprehensive protection.

Implications

Our findings carry important implications for both researchers and practitioners working with code LLMs.

For researchers: Our results confirm that memorization remains a persistent challenge in modern LLMs and is influenced by multiple factors beyond parameter count, including training domain and data characteristics. Future work should explore categorical memorization across diverse datasets and develop quantitative metrics for assessing memorization risk at different granularities. Additionally, investigating the mechanisms by which distillation reduces memorization could inform the design of more effective mitigation strategies.

For practitioners: The prevalence of unintentional memorization poses practical risks for AI-assisted coding tools. Developers and tool providers should implement stricter review processes for model outputs, particularly when generated code may contain licensed or proprietary content. Organizations deploying code LLMs should consider adopting the memorization-revealing pipeline as part of their validation and auditing workflows. Furthermore, incorporating knowledge distillation into model deployment pipelines could reduce memorization risks while maintaining performance.

For policymakers: The potential for LLMs to inadvertently leak licensed code or sensitive information raises legal and ethical concerns. Clear guidelines are needed regarding the use of memorized training data in AI-generated outputs, particularly in commercial settings. Providers should be transparent about memorization risks and implement technical safeguards to minimize unintended data exposure.

5 Threats to validity

Internal validity. While we compared models of different sizes (3B, 7B, 8B), these models also differ in architecture and training procedures. StarCoder2 models are trained exclusively on code, while Llama3-8B is trained on diverse data. We cannot fully disentangle the effects of parameter count, training data composition, and architectural design. Our knowledge distillation evaluation compared only one teacher-student pair (DeepSeek-R1 and Llama3-8B), and the observed reduction may be specific to this configuration.

External validity. We evaluated models up to 8 billion parameters, treating this ≤8B scale as a specific scoping constraint for this study. While recent literature suggests that memorization trends scale predictably with parameter count, our observations may not perfectly mirror the behaviors of significantly larger frontier models (e.g., GPT-4, Claude 3). Empirically evaluating this pipeline on larger open-weight models (e.g., CodeLlama-13B) remains a priority for our future extension work. Furthermore, we focused exclusively on accessible open-weight models and Python code from public GitHub repositories. Memorization patterns may differ for proprietary LLMs, other programming languages, or codebases with different structural characteristics.

Construct validity. There is no universally accepted definition of memorization in language models. We operationalize similarity-based extractability through exact match and BLEU-4 scores, but this may not capture semantic patterns or paraphrased content. Furthermore, our similarity-based filter is a heuristic approach lacking cryptographic ground-truth verification (e.g., querying Data Portraits [19]). This introduces risks of false positives (flagging highly predictable boilerplate as memorized) and false negatives (failing to trigger memorized sequences due to suboptimal prompting). Additionally, our choice of a 50-token prefix is a fixed parameter; extraction rates are inherently sensitive to this threshold, as providing longer contexts generally increases the probability of verbatim generation. Finally, our categorization of code relies on heuristic rules that may introduce classification errors.

6 Conclusion

This study investigates memorization behavior in recent open-weight LLMs applied to coding tasks through a systematic two-stage memorization evaluation pipeline. We find that memorization persists across modern models, with extraction rates ranging from 42% to 64%, confirming that the existing pipeline remains effective for recent LLMs. Our categorical analysis reveals that repetitive content (e.g., license headers and documentation) exhibits the highest memorization rates, while complex code such as testing logic is less susceptible. Realistic code completion scenarios can trigger unintentional memorization at rates of 13-14%, raising practical concerns for AI-assisted development tools. Our investigation of knowledge distillation demonstrates promising results, with the distilled Llama3-8B showing an approximate 19% reduction in extraction rates. These findings establish that memorization is influenced by a complex interplay of training domain, dataset composition, architectural choices, and content characteristics, rather than parameter count alone, and that distillation offers a cost-effective mitigation approach, though incomplete as a standalone solution.

7 Data Availability

The experimental data and scripts are available at https://github.com/senseuwaterloo/LLM_Memorization.

References

[1] Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. 2023. Targeted Attack on GPT-Neo for the SATML Language Model Data Extraction Challenge. doi:10.48550/arXiv.2302.07735 arXiv:2302.07735 [cs].

[2] Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. 2024. Traces of Memorisation in Large Language Models for Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3597503.3639133

[3] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. 2019. The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks. doi:10.48550/arXiv.1802.08232 arXiv:1802.08232 [cs].

[4] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting Training Data from Large Language Models. doi:10.48550/arXiv.2012.07805 arXiv:2012.07805 [cs].

[5] DeepSeek-AI, Daya Guo, Dejian Yang, and et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. doi:10.48550/arXiv.2501.12948 arXiv:2501.12948 [cs].

[6] Yijiang River Dong, Hongzhou Lin, Mikhail Belkin, Ramon Huerta, and Ivan Vulić. 2024. Unmemorization in Large Language Models via Self-Distillation and Deliberate Imagination. doi:10.48550/arXiv.2402.10052 arXiv:2402.10052 [cs] version: 1.

[7] Vitaly Feldman. 2021. Does Learning Require Memorization? A Short Tale about a Long Tail. doi:10.48550/arXiv.1906.05271 arXiv:1906.05271 [cs].

[8] Sérgio Fernandes and Jorge Bernardino. 2015. What is BigQuery?. In *Proceedings of the 19th International Database Engineering & Applications Symposium (IDEAS '15)*. Association for Computing Machinery, New York, NY, USA, 202–203. doi:10.1145/2790755.2790797

[9] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and

Connor Leahy. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. doi:10.48550/arXiv.2101.00027 arXiv:2101.00027 [cs].

[10] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, and et al. 2024. The Llama 3 Herd of Models. doi:10.48550/arXiv.2407.21783 arXiv:2407.21783 [cs].

[11] Yufei Guo, Muzhe Guo, Juntao Su, Zhou Yang, Mengqiu Zhu, Hongfei Li, Mengyang Qiu, and Shuo Shuo Liu. 2024. Bias in Large Language Models: Origin, Evaluation, and Mitigation. doi:10.48550/arXiv.2411.10915 arXiv:2411.10915 [cs].

[12] Abhimanyu Hans, Yuxin Wen, Neel Jain, John Kirchenbauer, Hamid Kazemi, Prajwal Singhania, Siddharth Singh, Gowthami Somepalli, Jonas Geiping, Abhinav Bhatele, and Tom Goldstein. 2024. Be like a Goldfish, Don't Memorize! Mitigating Memorization in Generative LLMs. doi:10.48550/arXiv.2406.10209 arXiv:2406.10209 [cs].

[13] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (April 2016), 122–131. doi:10.1145/2902362

[14] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. doi:10.48550/arXiv.1503.02531 arXiv:1503.02531 [stat].

[15] Yuan Jiang, Zehao Li, Shan Huang, Christoph Treude, Xiaohong Su, and Tiantian Wang. 2025. Effective code membership inference for code completion models via adversarial prompts. *arXiv preprint arXiv:2511.15107* (2025).

[16] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2022. Deduplicating Training Data Makes Language Models Better. doi:10.48550/arXiv.2107.06499 arXiv:2107.06499 [cs].

[17] Yujia Li, David Choi, Junyoung Chung, and et al. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. doi:10.1126/science.abq1158 arXiv:2203.07814 [cs].

[18] Anton Lozhkov, Raymond Li, Loubna Ben Allal, and et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. doi:10.48550/arXiv.2402.19173 arXiv:2402.19173 [cs].

[19] Marc Marone and Benjamin Van Durme. 2023. Data portraits: Recording foundation model training data. *Advances in Neural Information Processing Systems* 36 (2023), 15121–15135.

[20] Yuqing Nie, Chong Wang, Kailong Wang, Guoai Xu, Guosheng Xu, and Haoyu Wang. 2025. Decoding secret memorization in code llms through token-level characterization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2880–2892.

[21] Rafiqul Rabin, Sean McGregor, and Nick Judd. 2025. Malicious and Unintentional Disclosure Risks in Large Language Models for Code Generation. doi:10.48550/arXiv.2503.22760 arXiv:2503.22760 [cs].

[22] Fabio Salerno, Ali Al-Kaswan, and Maliheh Izadi. 2025. How Much Do Code Language Models Remember? An Investigation on Data Extraction Attacks before and after Fine-tuning. doi:10.48550/arXiv.2501.17501 arXiv:2501.17501 [cs].

[23] Yashothara Shanmugarasa, Ming Ding, M. A. P. Chamikara, and Thierry Rakotoarivelo. 2025. SoK: The Privacy Paradox of Large Language Models: Advancements, Privacy Risks, and Mitigation. doi:10.1145/3708821.3733888 arXiv:2506.12699 [cs].

[24] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership Inference Attacks Against Machine Learning Models. In *2017 IEEE Symposium on Security and Privacy (SP)*. 3–18. doi:10.1109/SP.2017.41 ISSN: 2375-1207.

[25] Victoria Smith, Ali Shahin Shamsabadi, Carolyn Ashurst, and Adrian Weller. 2024. Identifying and Mitigating Privacy Risks Stemming from Language Models: A Survey. doi:10.48550/arXiv.2310.01424 arXiv:2310.01424 [cs].

[26] Weiwei Xu, Kai Gao, Hao He, and Minghui Zhou. 2025. LiCoEval: Evaluating LLMs on License Compliance in Code Generation. doi:10.48550/arXiv.2408.02487 arXiv:2408.02487 [cs] version: 3.

[27] Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. 2024. A Survey on Knowledge Distillation of Large Language Models. doi:10.48550/arXiv.2402.13116 arXiv:2402.13116 [cs].

[28] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2024. Unveiling Memorization in Code Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639074 arXiv:2308.09932 [cs].

Received 2026-02-15; accepted 2026-03-28