Large Language Model Unlearning for Source Code

Xue Jiang[1,2*], Yihong Dong[1,2,*], Huangzhao Zhang[3], Tangxinyu Wang[1], Zheng Fang[1], Yingwei Ma[2], Rongyu Cao[2], Binhua Li[2], Zhi Jin[1], Wenpin Jiao[1], Yongbin Li[2], Ge Li[1†]

[1]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China
[2]Tongyi Lab, Alibaba Group
[3]Verdent AI
{jiangxue, dongyh}@stu.pku.edu.cn, lige@pku.edu.cn

Abstract

While Large Language Models (LLMs) excel at code generation, their inherent tendency toward verbatim memorization of training data introduces critical risks like copyright infringement, insecure emission, and deprecated API utilization, *etc*. A straightforward yet promising defense is unlearning, *i.e*., erasing or down-weighting the offending snippets through post-training. However, we find its application to source code often tends to spill over, damaging the basic knowledge of programming languages learned by the LLM and degrading the overall capability. To ease this challenge, we propose PROD for precise source code unlearning. PROD surgically zeroes out the prediction probability of the prohibited tokens, and renormalizes the remaining distribution so that the generated code stays correct. By excising only the targeted snippets, PROD achieves precise forgetting without much degradation of the LLM's overall capability. To facilitate in-depth evaluation against PROD, we establish an unlearning benchmark consisting of three downstream tasks (*i.e*., unlearning of copyrighted code, insecure code, and deprecated APIs), and introduce Pareto Dominance Ratio (PDR) metric, which indicates both the forget quality and the LLM utility. Our comprehensive evaluation demonstrates that PROD achieves superior overall performance between forget quality and model utility compared to existing unlearning approaches across three downstream tasks, while consistently exhibiting improvements when applied to LLMs of varying series. PROD also exhibits superior robustness against adversarial attacks without generating or exposing the data to be forgotten. These results underscore that our approach not only successfully extends the application boundary of unlearning techniques to source code, but also holds significant implications for advancing reliable code generation.

Introduction

Large Language Models (LLMs) have revolutionized the field of software engineering, automatically converting requirements into executable code with efficiency exceeding human programmers (Li et al. 2022; GitHub 2022; Anthropic 2025). Such a leap is powered by the scaling law, *i.e*., ever-larger models and ever-growing corpora yield ever-better performance (Kaplan et al. 2020; Hoffmann et al. 2022). However, the training corpora themselves are usually scraped from the open web, almost inevitably containing copyrighted fragments, vulnerable snippets, and deprecated Application Programming Interfaces (APIs), *etc*. Because LLMs are trained autoregressively, they can memorize and later generate these undesirable snippets (Dong et al. 2024), exposing users to legal risk, security breaches, fragile legacy code, and so on. Empirical studies show that up to 40% of LLM-generated code contains exploitable vulnerabilities (He et al. 2024), while a wave of high-profile lawsuits accuses leading LLMs of violating the copyright of the developers (Zhang et al. 2024a; Butterick 2022).

Purging every problematic code snippet from training corpora and retraining the LLMs from scratch would be straightforward and ideal, yet the cost is obviously unacceptable. A practical alternative is to force the LLMs to selectively forget what it was never meant to know, which is the objective known as unlearning (Yao, Xu, and Liu 2024; Liu et al. 2025). Given a functional LLM, unlearning algorithms behave like erasers: they excise the influence of the undesirable samples from the target LLM, while leaving all other knowledge and capabilities intact. Existing LLM unlearning methods (Yao, Xu, and Liu 2024; Zhang et al. 2024b; Rafailov et al. 2023; Wang et al. 2025) typically optimize the model to suppress the likelihood of undesirable outputs by reversing the gradient descent process upon undesirable contents or setting them as negative samples (Liu et al. 2025; Yao, Xu, and Liu 2024; Zhang et al. 2024b; Rafailov et al. 2023; Wang et al. 2025). *e.g*., Gradient Ascent (GA) (Yao, Xu, and Liu 2024) tunes parameters uphill on samples to be forgotten. For natural language tasks, such methods have been demonstrated effective in excising toxic responses, secret leakage, or policy-violating answers without impairing general capabilities significantly.

Programming languages, however, are bound by rigid syntax, along with strict semantics. When existing unlearning methods directly down-weighting undesirable code snippets, they also strip the language scaffolding that keeps the

[*]Work done during Xue Jiang and Yihong Dong's internship at Tongyi Lab.
[†]Corresponding author.

surrounding code valid. It results in direct utility loss: syntax errors and grammar errors occur, high-frequency tokens repeat, or even the LLMs fall mute. Our pilot empirical study confirms such an issue (see Figure 1): after manipulated by existing unlearning techniques, the target LLMs frequently violate programming language rules or emit nonsensical code snippets. We argue that the root cause here is granularity, *i.e.*, current techniques erase entire slices of knowledge (*i.e.*, the undesirable snippets), damaging the foundational grammar and universal coding patterns the model once mastered through pre-training. Therefore, the open challenge is to eliminate only the offending snippets with surgical precision (maximize forget quality, *i.e.*, the degree that the undesirable contents are eliminated) while leaving model utility (*i.e.*, the general performance of the target LLM) intact.

We introduce **P**robabilistic **R**edistribution for **O**utput **D**istribution (abbreviated as PROD), a surgical, code-oriented unlearning method that forgets code snippets that must be forgotten while leaving all other knowledge of programming languages intact. Unlike existing unlearning approaches, which operate at the granularity of samples, PROD manipulates distributions over individual tokens in a fine-grained granularity. PROD first captures the full distribution over vocabulary predicted by the target LLM, then surgically zeroes out every probability mass assigned to tokens in the undesirable snippet. After pruning incidental noises, it reallocates the probability distribution across the remaining vocabulary in a manner that faithfully preserves statistics of programming languages learned by the target LLM. By optimizing the model to match this carefully sculpted target distribution, PROD attains near-perfect forgetting of the targeted code while effectively maintaining its utility unchanged. We also propose a benchmark for source code unlearning along with a metric, Pareto Dominance Ratio (PDR). The benchmark covers three tasks: 1) copyrighted code unlearning, where the goal is to remove copyrighted code snippets from LLMs; 2) insecure code unlearning, which aims to avoid security vulnerabilities produced by LLMs; 3) deprecated API unlearning, which prevents LLMs from generating APIs from outdated libraries or packages. PDR metric evaluates unlearning approaches by considering both forget quality and model utility.

Our in-depth evaluation over the three unlearning tasks shows that PROD achieves the best PDR score, suggesting its best overall performance between forget quality and model utility. Experiments on four distinct LLMs (*i.e.*, CodeLlama-7B, Qwen2.5-Coder-7B, Deepseek-coder-6.7, Starcoder-7B) demonstrate the broad applicability of PROD. Adversarial attack experiment further reveals that PROD is the only unlearning method among existing LLM unlearning methods resilient against targeted attacks without reproducing forgotten codes.

Our main contributions are highlighted as follows:

- We conduct an investigation of existing LLM unlearning approaches on code-related tasks, revealing that they lead to severe model utility degradation, making models practically unusable for code generation.
- We propose a novel unlearning approach PROD, that can precisely forget undesired code snippets and preserve knowledge of programming languages intact in LLMs.
- We identify three critical unlearning applications in the code generation domain (including copyrighted code unlearning, insecure code unlearning, and deprecated API unlearning) and establish a benchmark for evaluating LLM unlearning approaches for code.
- Extensive experimental results demonstrate that PROD significantly outperforms existing LLM unlearning approaches, achieving successful forgetting of specific code while effectively preserving code generation capabilities. Our source code and data are available at https://github.com/jiangxxxue/PROD.

Related Work

In this section, we outline the most relevant work of PROD and detail the extended related work section in Appendix.

Code Generation with LLMs

Code generation has been significantly advanced by LLMs, from general-purpose LLMs like GPT-4 (Achiam et al. 2023) to a variety of specialized code LLMs (Du et al. 2024; Rozière et al. 2023; Zhu et al. 2024; Hui et al. 2024; Team 2024). However, their training on vast, public code repositories introduces critical challenges regarding security, reliability, and legality. A primary concern is that LLMs may replicate vulnerabilities. To mitigate this, researchers have explored security-focused fine-tuning with curated datasets (SaferCode (He et al. 2024)) and in-context learning with secure examples (Mohsin et al. 2024). Beyond security, model reliability is undermined by poor handling of software versions. Studies show that LLMs often suggest deprecated APIs (SecuCoGen (Wang et al. 2024)) and struggle with API changes across different library releases, a challenge highlighted by the VersiCode benchmark (Wu et al. 2024). Finally, legal issues arise from models generating code that may violate software licenses, as systematically evaluated by LiCoEval (Xu et al. 2025).

These works highlight that despite remarkable progress, LLM-based code generation still falls short of developers' expectations for producing secure, legally compliant, and up-to-date code. Our work aims to address this gap by proposing a general unlearning approach for suppressing undesired code output from LLMs.

LLM Unlearning

Machine unlearning aims to remove the influence of specific data from a trained model (Cao and Yang 2015). Initial research primarily focused on classification models (Golatkar, Achille, and Soatto 2020; Izzo et al. 2021; Bourtoule et al. 2021), with methods broadly categorized into data-reversed training (Tarun et al. 2024; Chundawat et al. 2023), influence function-based approaches (Izzo et al. 2021), and optimization-based unlearning (Guo et al. 2020; Neel, Roth, and Sharifi-Malvajerdi 2021).

The vast scale and generative nature of LLMs present unique challenges, rendering many traditional methods impractical (Liu et al. 2025). Specifically, the computational

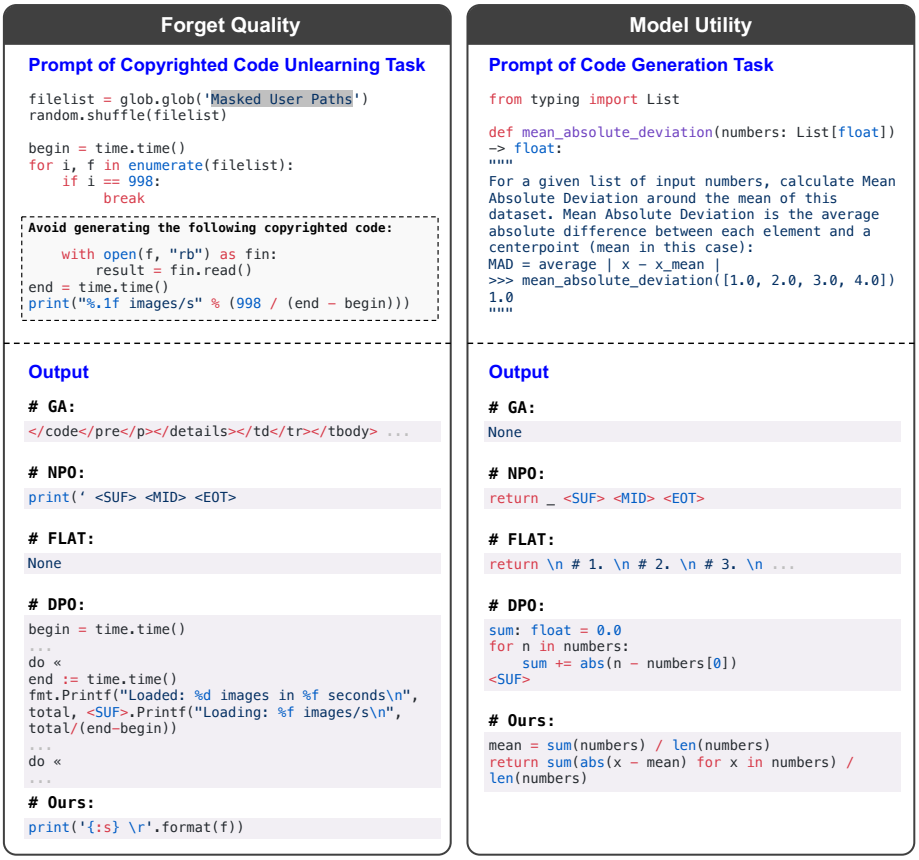


Figure 1: A case on the forget quality and model utility of existing method in unlearning code. Existing methods (GA, NPO, FLAT, DPO) exhibit severe utility degradation, such as mute refusal, token collapse, or syntactic incoherence. PROD successfully unlearns copyrighted content while maintaining utility for general code generation tasks.

cost of Hessian inversion makes influence function-based methods prohibitive for LLMs. Consequently, recent work has shifted towards optimization-based techniques tailored for LLMs in the NLP domain. Methods such as GA (Yao, Xu, and Liu 2024), DPO-based unlearning (Rafailov et al. 2023), NPO (Zhang et al. 2024b), and FLAT (Wang et al. 2025) have been proposed to erase private data, copyrighted material, and harmful content. The evaluation of these techniques relies on specialized benchmarks (Maini et al. 2024; Yao, Xu, and Liu 2023) and is often tested for robustness against adversarial attacks like prefix injections (Wei, Haghtalab, and Steinhardt 2023; Yuan et al. 2024; Qi et al. 2025).

Given that source code possesses unique characteristics compared to natural language, although the exploration of LLM unlearning in NLP has proven its value, its application in code generation remains largely unexplored.

Preliminary of LLM Unlearning

In this section, we first formalize the LLM unlearning problem and establish the evaluation criteria. Next, we briefly survey four representative baselines that span current LLM unlearning paradigms. Finally, we conduct an empirical pilot study revealing that, on source code, existing baselines cause great losses in model utility, motivating the need for a more surgical solution.

Objective of Unlearning. Within the broad landscape of machine unlearning, we focus on optimization-based methods. Here, unlearning is a post-training procedure that removes the LLM's ability to produce specific, undesirable code snippets. Let $\mathcal{D}_f = \{x_f^{(i)}, y_f^{(i)}\}_{i=1}^{N_f}$, denote the set of such snippets, where each pair consists of a prompt $x_f^{(i)}$, and its undesirable completion $y_f^{(i)}$ (note that the prompt $x_f^{(i)}$ can be empty in cases such as the entire snippet $y_f^{(i)}$ must be forgotten). After unlearning, the target LLM π_Θ is supposed to assign zero probability to every undesirable $y_f^{(i)}$ given $x_f^{(i)}$. Formally, we quantify this objective with the loss $\mathcal{L}_f\left(\pi_\Theta(x_f), y_f\right)$ that measures the divergence between the LLM's output $\pi_\Theta(x_f^{(i)})$ and the undesirable output $y_f^{(i)}$. Unlearning is then cast as the following optimization:

$$\min_{\Theta} \sum_{i=1}^{N_f} -\mathcal{L}_f\left(\pi_\Theta(x_f^{(i)}), y_f^{(i)}\right), \tag{1}$$

where maximizing $\mathcal{L}_f$ forces π_Θ to forget every undesirable $y_f^{(i)}$ in $\mathcal{D}_f$, including copyrighted code, insecure fragment,

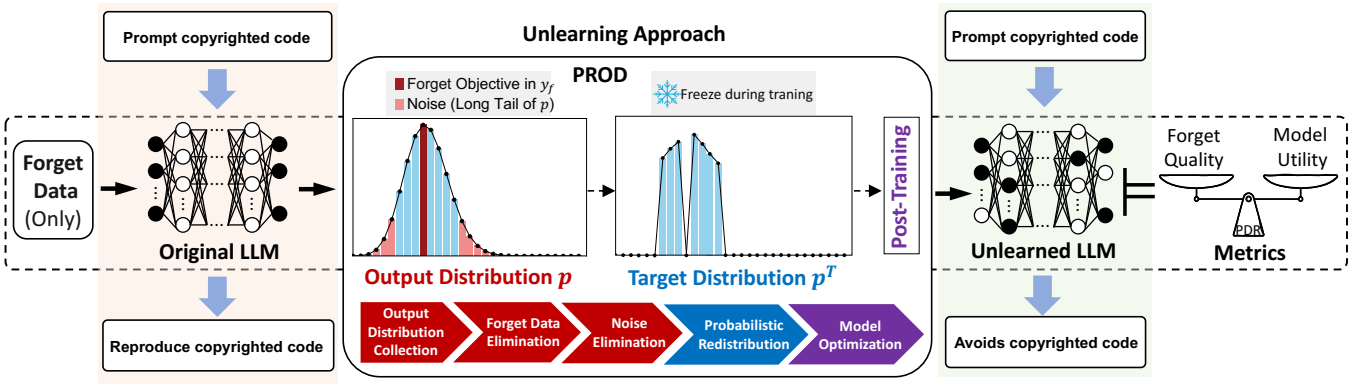


Figure 2: The PROD unlearning pipeline. The pipeline contains three key steps: 1) Suppress the probabilities of undesirable code snippets to zero. 2) Redistribute the probabilities across the remaining vocabulary. 3) Optimize the model to match the surgically sculpted target distribution.

or deprecated API invocation, *etc*. Notably, $\mathcal{D}_f$ merely captures behaviors we wish to eliminate, and samples in it do not have to appear verbatim in the original training data.

Metrics. As aforementioned, unlearning requires to erase only what is undesirable or even harmful, and retain everything else. We therefore assess unlearning approaches along two axes:

1. Forget quality: the thoroughness with which π_Θ eliminates the influence of $\mathcal{D}_f$, measured by the degree of dissimilarity between the output of π_Θ and $y_f^{(i)}$ when prompted with $x_f^{(i)}$.
2. Model utility: the collateral side damage caused by the procedure, gauged by π_Θ's retained performance on other general tasks and datasets. Assessing the balance between forget quality and model utility is essential due to their trade-off in unlearning methods.

LLM Unlearning Solutions. Existing methods are almost exclusively tuned to optimize the forget objective in Equation (1), and treat downstream utility as an after-thought, resulting in plausible damage. Gradient Ascent (GA) (Yao, Xu, and Liu 2024) carries out a "reverse" of training by ascending the loss on $\mathcal{D}_f$, pushing π_Θ away from the parameter region that supports undesirable generations. Negative Preference Optimization (NPO) (Zhang et al. 2024b), which is a variant of Direct Preference Optimization (DPO) (Rafailov et al. 2023), supplies only negative examples (*i.e.*, $(x_f^{(i)}, y_f^{(i)})$) to repel π_Θ from the unwanted distribution. DPO is a general preference optimization method that, while not specifically designed for unlearning, can be adapted for this purpose by treating the content-to-be-forgotten as the "dispreferred" response. Forget Data Only Loss Adjustment (FLAT) (Wang et al. 2025) formulates $\mathcal{L}_f$ with f-divergence, enforcing π_Θ to assign maximal probability to a hand-crafted refusal template (*e.g.*, "I do not know") while minimizing probability on each $(x_f^{(i)}, y_f^{(i)})$.

A Motivating Example. To vividly show the collateral damage caused by existing unlearning methods on source code tasks, we conduct an empirical study. We benchmark the above-mentioned baselines on a copyrighted code unlearning task and then measure their retained utility with a standard code generation benchmark (detailed setups in the experiment section). As Figure 3 indicates, all baseline approaches achieve $\approx 90\%$ forgetting ratio, yet their success ratio on code generation degrades to near-zero when they reach best forget quality. In other words, these approaches successfully erase the undesirable snippets (*i.e.*, copyrighted code) but simultaneously cause LLMs to forget how to program, *i.e.*, such an outcome makes the target LLMs almost useless. Our motivation in this paper is precisely the opposite, as we argue to excise the undesirable code snippets while leaving the target LLMs fully operational.

A closer look at the baselines' outputs (a case is presented in Figure1) reveals three dominant failure modes. 1) Mute Refusal: the model declines to produce code, 2) Token Collapse: the output degenerates into repetitive high-frequency tokens such as "\n" or training artifacts like "<SUF>", and 3) Syntactic Incoherence: the generated code violates fundamental language rules and fails to compile or execute. These breakdowns arise from the baselines' coarse granularity. By bluntly erasing every trace of $y_f^{(i)}$ from π_Θ, they eliminate not only the undesirable snippet but also the surrounding "scaffolding". Natural languages tolerate such collateral damage as they are much more flexible, yet programming languages, which are governed by rigid syntax and precise dependencies, do not. Once the supporting knowledge and structure are disturbed, the entire code generation capability collapses. To preserve utility while still forgetting, we introduce PROD, a surgical unlearning solution that manipulates the output distribution at token-level granularity instead of bluntly excising entire snippet sequences.

PROD

To protect model utility while guaranteeing forgetting, we introduce PROD, a surgically precise unlearning method that manipulates the LLM's token-level output distributions. As illustrated in Figure 2 and formalized in Algorithm 1, the working pipeline of PROD consists of three steps. 1)

Suppress the probabilities predicted by the target LLM of tokens that constitute the undesirable snippet to zero (*i.e.*, remove them from the output). 2) Redistribute the probabilities across the remaining vocabulary so that the resulting distribution mirrors the original statistical patterns and preserves knowledge that the LLM learn from the code corpora. 3) Optimize the target LLM to match this surgically sculpted target distribution. As a result, PROD achieves complete erasure of undesirable or even harmful code with virtually no collateral damage to any other programming competency.

Target Distribution Sculpting

We begin by sculpting the target LLM's output distribution into a precise supervisory signal that will steer the entire unlearning process of PROD.

Output Distribution Collection. The initial step of PROD retrieves the LLM's raw next-token distributions for each (x_f, y_f) pair [1]. Given a forget pair (x_f, y_f) with $y_f = (y_{f,1}, \cdots, y_{f,L})$, at time-step t, we feed the target LLM (π_Θ) the concatenation of the prompt x_f and the preceding ground-truth prefix $y_{f,<t} = (y_{f,1}, \cdots, y_{f,t-1})$ to obtain the logits $h_t \in \mathbb{R}^{|\mathcal{V}|}$, where $\mathcal{V}$ refers to the vocabulary. A softmax converts the logits into the original distribution $p_t(\cdot|x_f, y_{f,<t})$. p_t then serves as the canvas where the subsequent manipulations will be carried out.

Forget Data Elimination. To prohibit any generation of the target sequence, PROD performs a single, decisive edit on h_t at every t. The logit entry corresponding to the target token $y_{f,t}$ is suppressed to $-\infty$ as below:

$$\hat{h}_t[j] = \begin{cases} -\infty, & \text{If } \mathcal{V}[j] = y_{f,t}, \\ h_t[j], & \text{Otherwise}, \end{cases} \tag{2}$$

where $[\cdot]$ denotes vector indexing, and $\mathcal{V}[j]$ refers to the j-th token in the whole vocabulary $\mathcal{V}$. A softmax normalization of $\hat{h}_t$ would yield the eliminated distribution $\hat{p}_t(\cdot|x_f, y_{f,<t})$, which assigns zero probability to $y_{f,t}$ and redistributes its entire mass across the remaining vocabulary in exact proportion to their original likelihoods.

Noise Elimination. Suppressing the target token $y_{f,t}$ can inadvertently amplify less probable, non-target tokens, injecting noise into $\hat{p}_t(\cdot|x_f, y_{f,<t})$. We therefore prune the tail (*i.e.*, the noise) via nucleus sampling (Holtzman et al. 2020) with the threshold of p. Let $\mathcal{S}_p$ denote the smallest set of tokens whose cumulative probability under $\hat{p}_t$ reaches p. All logits outside $\mathcal{S}_p$ are then set to $-\infty$:

$$\widetilde{h}_t[j] = \begin{cases} \hat{h}_t[j], & \text{If } \mathcal{V}[j] \in \mathcal{S}_p, \\ -\infty, & \text{Otherwise}. \end{cases} \tag{3}$$

Only tokens within $\mathcal{S}_p$ receive redistributed probability, and the rest are eliminated as noise. The softmax normalization upon $\widetilde{h}_t$ results in noise-trimmed distribution $\widetilde{p}_t$.

Crucially, forget data elimination must precede noise elimination. Reversing the order would either retain target tokens or produce an overly sparse distribution. This two-step sequence ensures a clean unlearning process while preserving a robust, information-rich distribution over the surviving vocabulary.

[1]For simplicity, the superscripts in $x_f^{(i)}$ and $y_f^{(i)}$ are omitted henceforth in this section.

Algorithm 1: Pseudocode for PROD.

Require: LLM π_Θ; forget set $\mathcal{D}_f$; learning rate η; training epochs N; hyperparameters p, α.
Ensure: Unlearned LLM π_Θ^*.

```
for epoch k = 1 to N do
    for each (x_f, y_f) ∈ D_f do
        Collect output distribution p(w|x_f, y_{f,<t}).
        Compute ĥ_t to apply forget data elimination.
        Compute h̃_t to perform noise elimination.
        Compute probabilistic redistribution p_T.
        Calculate loss function L_PROD.
        Θ_{k+1} = Θ_k − η∇_Θ L_PROD.
        k ← k + 1 and Θ* ← Θ_k
    end for
end for
return π*_Θ
```

Probabilistic Redistribution. After excising both the target token $y_{f,t}$ and the noisy tail outside S_p, $\widetilde{h}_t$ holds the remaining probability mass. PROD redistributes this mass across the surviving safe vocabulary (*i.e.*, $\mathcal{S}_p$) through softmax, and therefore adopts $\widetilde{p}_t$ as the supervisory signal p_T.

Post-Training

With the sculpted supervisory distribution $p_T = \widetilde{p}_t$, PROD optimizes the target LLM π_Θ via standard cross-entropy loss, nudging its outputs to align with this sculpted distribution while preserving every other learned competency.

Objective. The optimization objective of PROD is to minimize the loss $\mathcal{L}_{\text{PROD}}$, finalized as following:

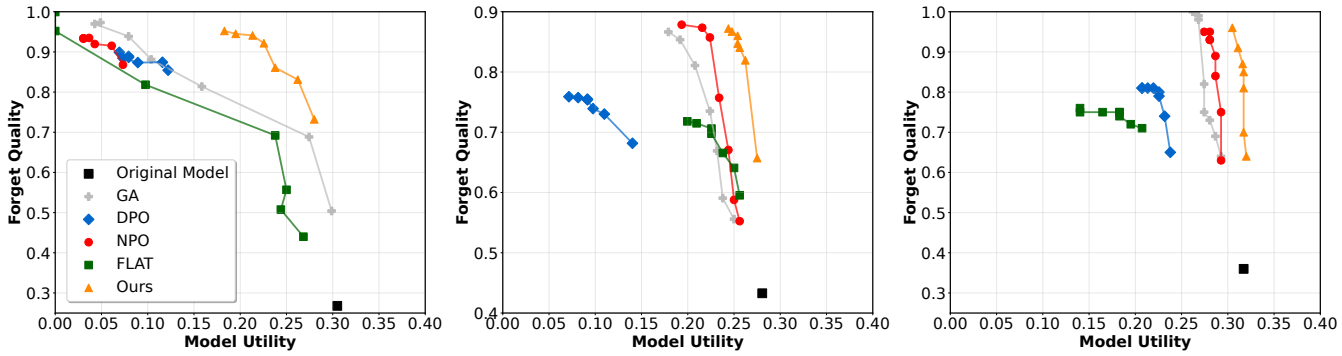
$$\mathcal{L}_{\text{PROD}}(x_f, y_f; \pi_\Theta) = -\sum_{t=1}^{L} \sum_{w \in \mathcal{V}} p_T(w|\cdot) \log \pi_\Theta(w|\cdot), \tag{4}$$

where $\pi_\Theta(\cdot)$ refers to the actual output distribution of π_Θ. The conditions (*i.e.*, $x_f, y_{f,<t}$) are neglected for simplicity. In default design, PROD adopts cross-entropy. The framework is agnostic, *i.e.*, KL (Kullback and Leibler 1951), JS (Lin 1991), or any other distributional divergence can be substituted seamlessly.

Optimization. Any compatible optimizer, with gradient descent included, can be employed in PROD.

Key Know-How of PROD

In general, PROD provides two strengths. 1) Model utility preservation: the supervisory p_T is a surgically sculpted clone of the original p_t, excising only the undesirable or forbidden snippets while leaving every other competency untouched. 2) Efficient convergence: since p_T remains close to the π_Θ's initial output, the optimization process is fast, stable, and light on compute. Furthermore, we elaborate on two implementation details.



(a) Copyrighted Code Unlearning (b) Insecure Code Unlearning (c) Deprecated API Unlearning

Figure 3: Forget quality versus model utility across different unlearning tasks. Performance curves positioned closer to the upper-right corner indicate superior approach effectiveness.

α-suppression. Although PROD excels in unlearning long code, early experiments revealed that short forget sequences y_f often require more training steps since the supervisory signal is weak. To modulate the "forget strength", we introduce the α-suppression trick and modify the supervisory signal p_T as follows:

$$p_{\mathrm{T}}(w|x_f, y_{f,<t}) = \begin{cases} -\alpha \cdot p_o(w|\cdot), & \text{If } w = y_{f,t}, \\ \widetilde{p}_t(w|\cdot), & \text{Otherwise,} \end{cases} \quad (5)$$

where p_o and $\widetilde{p}_t$ refers to the original and noise-trimmed distributions, respectively. Still, some conditions are neglected for better presentation. α is a hyperparameter to control the degree of forget, *i.e.*, setting $\alpha = 0$ disables the trick ($p_T = \widetilde{p}_t$); increasing α amplifies the negative pressure on y_f, yielding a stronger unlearning signal. Note that while α-suppression introduces negative probability, it is harmless, or even beneficial, under the setting of cross-entropy as defined in Equation 4: the remaining terms maintain the loss magnitude and gradient direction; and the negative entry amplifies the penalty on the forget token without computational issues.

Training Stability. As training proceeds, π_Θ's distribution drifts with every gradient step. If we recollected p_o and recomputed p_T on-the-fly, the supervisory signal would swing constantly, destabilizing the training process and risking collapse. To avoid such an issue, we freeze p_o at its initial state, *i.e.*, the distributions produced by the target LLM before unlearning, thereby guaranteeing a steady and reliable target throughout optimization.

Evaluation

We introduce a dedicated benchmark for code unlearning and report the main experimental results in this section. Please refer to the appendix for full implementation details and a perceptual quality study.

Code Unlearning Benchmark

Current unlearning benchmarks focus almost exclusively on natural language, leaving code generation unexamined. To fill this gap, we curate and adapt datasets into a code unlearning benchmark, paired with custom metrics. The benchmark contains three unlearning tasks (*i.e.*, copyrighted code, insecure code, and deprecated APIs) to measure forget quality and a general code generation set to assess model utility.

Copyrighted Code Unlearning. We randomly draw 100 files from the high-quality, deduplicated Stack corpus (Lozhkov et al. 2024) to serve as the forget set $\mathcal{D}_f$, simulating scenarios where users wish to purge copyrighted code files from an LLM. To mimic copyright protection, test files, toy examples, and standard templates are filtered and discarded, retaining those with expressive form, such as substantive implementations, business logic, and non-trivial algorithms. During unlearning, x_f is empty and y_f contains the full content of the retrieved file; when assessing forget quality, we prompt the target LLM with the first half of each file and evaluate against the other half. As the law of copyright protects expression instead of ideas (Autry 2002; U.S. Copyright Office 2024), textual similarity is employed as an indicator of potential copyright infringement. Thus, the forget quality score is measured by the complement of BLEU (Papineni et al. 2002) (*i.e.*, $1 - \text{BLEU}$), where higher values indicate stronger erasure of the protected code snippets.

Insecure Code Unlearning. CyberSecEval from Purple Llama (Bhatt et al. 2023) is a cybersecurity benchmark, containing 1,916 code snippets collected from open sources that exhibit 50 distinct CWE vulnerabilities (The MITRE Corporation 2025) across eight programming languages. We adopt these snippets as our forget set $\mathcal{D}_f$. Evaluation proceeds in an autocomplete setup, *i.e.*, given the code that precedes a known vulnerable segment (*i.e.*, x_f), we prompt the model to continue and then inspect the continuation for security flaws (*i.e.*, y_f). Forget quality is assessed by the average of two complementary indicators: $1 - \text{BLEU}$ between the LLM's generation and the vulnerable snippet, and the pass rate under CyberSecEval's built-in static analyzer, which detects any insecure lines.

Deprecated API Unlearning. This task is curated atop Versicode (Wu et al. 2024), which is a version-aware code-generation dataset. With the original dataset, we apply three

Task	GA	DPO	NPO	FLAT	Ours
Copyright	15.3	3.6	0.6	0.0	**41.8**
Insecurity	13.3	0.0	24.5	6.1	**70.4**
Deprecation	31.5	20.6	51.6	0.0	**58.0**

Table 1: PDR (in %) among unlearning approaches.

additional preprocessing steps. 1) Version filtration discards packages with ambiguous version constraints (*e.g.*, `>=2.1.0`), retaining those with explicit version requirements (*e.g.*, `==2.1.0`). 2) File filtration removes packages that have less than 3 snippets. 3) Temporal split designates a middle release as the deprecation boundary, labeling all invocations of APIs prior to that release as "deprecated" and the remainder as "still-valid." The final dataset consists of 252 packages, 3,449 snippets in total (≈14 snippets and 10 distinct APIs per package). Unlearning is carried out on the deprecated APIs, and evaluated on the remaining still-valid ones. Given the code context immediately preceding an API call, we measure forget quality by exact-match accuracy.

General Code Generation. Following prior work (He et al. 2024), we estimate model utility through HumanEval (Du et al. 2024), which is a widely used code generation benchmark. We employ functional correctness verified against test cases as the indicator of model utility.

Overall Performance Indicator. Forget quality and model utility often pull in opposite directions; naively averaging them or even independently assessing them overlooks the real trade-off. To make this multi-objective comparison explicit, we introduce the Pareto Dominance Ratio (PDR). Let $\mathcal{M} = \{m_1, \cdots, m_n\}$ be the set of evaluated unlearning methods and $\mathcal{O} = \{\text{forget, utility}\}$ the two objectives. For any method $m_i \in \mathcal{M}$, PDR is defined as following:

$$\text{PDR}(m_i) = \frac{\left|m_j \in \mathcal{M} \setminus m_i \mid m_i \succ_{\mathcal{O}} m_j\right|}{|\mathcal{M}| - 1}, \qquad (6)$$

where $m_i \succ_{\mathcal{O}} m_j$ means that m_i Pareto-dominates m_j, *i.e.*, m_i is no worse on both objectives and strictly better on at least one than m_j. Geometrically, on a 2-D scatter of forget quality versus model utility, $m_i \succ_{\mathcal{O}} m_j$ places m_i strictly to the upper-right of m_j. PDR therefore reports the fraction of rival methods that m_i dominates, yielding a single, interpretable summary of overall performance.

Experimental Results

Unlearning Performance. We benchmark PROD against four representative baselines (*i.e.*, GA, DPO, NPO, and FLAT) on all three unlearning tasks. The target model is CodeLlama-7B (Rozière et al. 2023). Greedy decoding (zero temperature) is applied for every inference run. On each task, we compute PDR with forget quality (task-specific) and overall model utility (via HumanEval). Comprehensive results are presented in Table 1 and illustrated in Figure 3.

PROD consistently occupies the upper-right region of the scatter plot (Figure 3), simultaneously delivering stronger forget quality and higher model utility than the baselines. The PDR metric (Table 1) also quantifies this dominance –

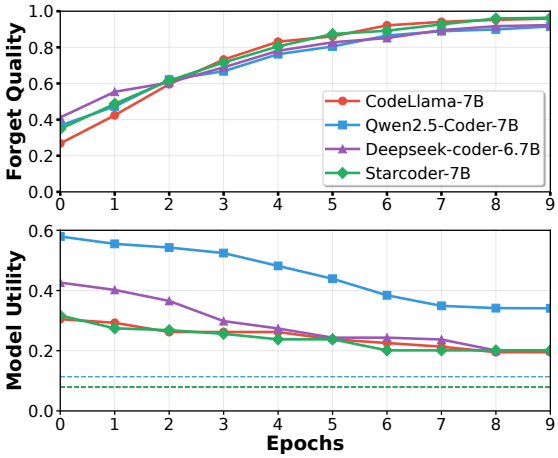


Figure 4: Performance of PROD on different LLMs, where the dotted line represents the best model utility of baselines in achieving comparable (*i.e.*, 90%) forget quality.

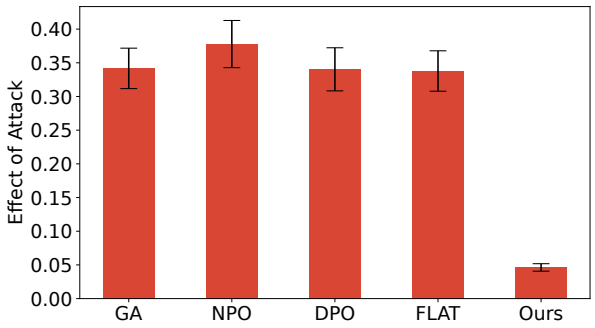
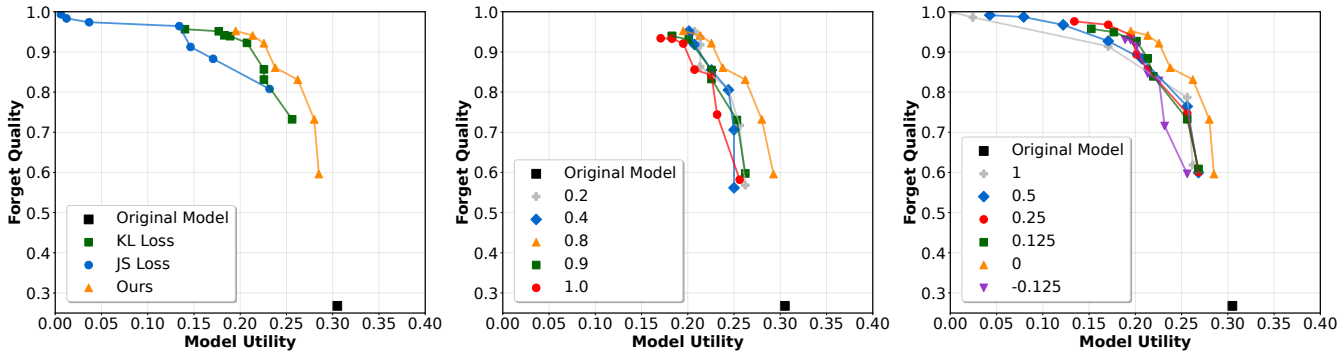


Figure 5: Comparison of adversarial attack results across different LLM unlearning approaches, showing mean attack effects with maximum and minimum ranges.

PROD tops every task, yielding an average relative gain of 124% over the strongest competitor. Training dynamics offer further insight. By analyzing the training log, we find that PROD forgets rapidly during early steps and then gently converges on both objectives, achieving high forgetting efficiency without triggering model collapse.

The three tasks pose different levels of challenges, as the curve slope in Figure 3 varies with the volume of undesirable content. Copyrighted code unlearning is the most challenging from this point of view, because it requires LLMs to forget entire files. Remarkably, PROD retains a notable advantage even on this most difficult task.

Application on Different LLMs. We extend our evaluation to four widely-used code LLMs (*i.e.*, CodeLlama-7B (Rozière et al. 2023), Qwen2.5-Coder-7B (Hui et al. 2024), Deepseek-coder-6.7B (Zhu et al. 2024), and Starcoder-7B (Li et al. 2023)) to demonstrate the versatility of PROD across diverse architectures and training recipes. While holding all other experimental settings fixed, we track both forget quality and model utility over successive epochs on



(a) Loss Function (b) Hyperparameter $\mathbf{p}$ (c) Hyperparameter α

Figure 6: Ablation results on alternative loss function, and the impact of hyperparameters p and α in PROD.

the copyrighted-code unlearning task.

Figure 4 reveals a consistent pattern across all four LLMs. As training proceeds, each LLM attains near-perfect forget quality while its utility curve stabilizes, which significantly outperforms all baseline approaches in model utility. This uniformity underscores both the strength and robustness of PROD. Among the evaluated models, CodeLlama-7B and StarCoder-7B exhibit the mildest utility degradation, implying that an LLM's intrinsic stability can modulate the effectiveness of unlearning.

Adversarial Attacks Recent studies have exposed the brittleness of unlearning: forgotten content can resurface when an adversary manipulates the prompt (Yuan et al. 2024; Qi et al. 2025). We therefore assess PROD against prefix injection attack, following previous work (Qi et al. 2025). In such an attack, an adversary prepends an increasing number of tokens from the copyrighted snippet to the prompt in an attempt to recover the erased snippets. We conduct this evaluation on the copyrighted code task against CodeLlama-7B and measure robustness with the drop in forget quality. To ensure validity of the experiment (*i.e.*, the model remains practically useful), we only evaluate checkpoints that retain at least 60% of the original LLM's HumanEval performance.

Figure 5 shows that PROD is significantly more robust under prefix injection attack than other baselines. Specifically, the similarity between the generation and the undesirable copyrighted code stays below 0.05, while others exceed 0.3. Moreover, PROD shows the smallest variance among baselines, underscoring its stable resilience against prefix inject attack. These results provide clear evidence that PROD delivers stronger and more reliable forgetting than baselines.

Ablation Study. Finally, we perform some ablations. We replace the default cross-entropy loss with two alternative divergence measures (*i.e.*, KL and JS). We also systematically vary the key hyperparameters p, which governs the strength of noise elimination, and α, which scales the amplified suppression on forget data samples.

From Figure 6a, we observe that Cross-entropy outperforms both KL and JS divergence, delivering the best trade-off between forget quality and model utility. Figure 6b shows that the moderate noise elimination ($p = 0.8$) achieves optimal results, while no noise elimination yields the worst. Figure 6c shows that α has a large impact on unlearning performance. When $\alpha = 0$, which prohibits the generation of forget data by setting its probability to zero in the target distribution, the model achieves the most balanced performance. Positive α values are beneficial for deepening the degree of forgetting, while small negative α values delay forgetting to maintain usability.

Conclusion

In this paper, we have investigated existing LLM unlearning approaches on source code, identifying a significant utility degradation problem. To this end, we propose PROD, a surgical, code-oriented unlearning method that forgets targeted code snippets while preserving other knowledge of programming languages intact. We also establish a benchmark covering copyrighted code, insecure code, and deprecated API unlearning tasks. Extensive experiments show PROD significantly outperforms existing methods in both forgetting performance and user experience, while also exhibiting broad applicability and robustness against unlearning attacks.

Acknowledgments

This research is supported by the National Key R&D Program under Grant No. 2023YFB4503801, the National Natural Science Foundation of China under Grant No. 62192733, 62192730, 62192731, and the Major Program (JD) of Hubei Province (No.2023BAA024).

References

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

Anthropic. 2025. Claude Code.

Autry, J. R. 2002. Toward a definition of striking similarity in infringement actions for copyrighted musical works. *J. Intell. Prop. L.*, 10: 113.

Bhatt, M.; Chennabasappa, S.; Nikolaidis, C.; Wan, S.; Evtimov, I.; Gabi, D.; Song, D.; Ahmad, F.; Aschermann, C.; Fontana, L.; Frolov, S.; Giri, R. P.; Kapil, D.; Kozyrakis, Y.; LeBlanc, D.; Milazzo, J.; Straumann, A.; Synnaeve, G.; Vontimitta, V.; Whitman, S.; and Saxe, J. 2023. Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models. *CoRR*, abs/2312.04724.

Bourtoule, L.; Chandrasekaran, V.; Choquette-Choo, C. A.; Jia, H.; Travers, A.; Zhang, B.; Lie, D.; and Papernot, N. 2021. Machine Unlearning. In *SP*, 141–159. IEEE.

Butterick, M. 2022. Butterick v. GitHub, Inc., Microsoft Corporation, and OpenAI, LP. U.S. District Court for the Northern District of California. Case No. 3:22-cv-07074, U.S. District Court for the Northern District of California.

Cao, Y.; and Yang, J. 2015. Towards Making Systems Forget with Machine Unlearning. In *IEEE Symposium on Security and Privacy*, 463–480. IEEE Computer Society.

Chundawat, V. S.; Tarun, A. K.; Mandal, M.; and Kankanhalli, M. S. 2023. Zero-Shot Machine Unlearning. *IEEE Trans. Inf. Forensics Secur.*, 18: 2345–2354.

Dong, Y.; Jiang, X.; Liu, H.; Jin, Z.; Gu, B.; Yang, M.; and Li, G. 2024. Generalization or Memorization: Data Contamination and Trustworthy Evaluation for Large Language Models. In *Findings of the Association for Computational Linguistics: ACL 2024*, 12039–12050. Association for Computational Linguistics.

Du, X.; Liu, M.; Wang, K.; Wang, H.; Liu, J.; Chen, Y.; Feng, J.; Sha, C.; Peng, X.; and Lou, Y. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*.

GitHub. 2022. Copilot.

Golatkar, A.; Achille, A.; and Soatto, S. 2020. Eternal Sunshine of the Spotless Net: Selective Forgetting in Deep Networks. In *CVPR*, 9301–9309. Computer Vision Foundation / IEEE.

Guo, C.; Goldstein, T.; Hannun, A. Y.; and van der Maaten, L. 2020. Certified Data Removal from Machine Learning Models. In *ICML*, volume 119, 3832–3842.

He, J.; Vero, M.; Krasnopolska, G.; and Vechev, M. 2024. Instruction tuning for secure code generation. In *Proceedings of the 41st International Conference on Machine Learning*. JMLR.org.

Hoffmann, J.; Borgeaud, S.; Mensch, A.; Buchatskaya, E.; Cai, T.; Rutherford, E.; Casas, D. d. L.; Hendricks, L. A.; Welbl, J.; Clark, A.; et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Holtzman, A.; Buys, J.; Du, L.; Forbes, M.; and Choi, Y. 2020. The Curious Case of Neural Text Degeneration. In *ICLR*. OpenReview.net.

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

Izzo, Z.; Smart, M. A.; Chaudhuri, K.; and Zou, J. 2021. Approximate Data Deletion from Machine Learning Models. In Banerjee, A.; and Fukumizu, K., eds., *The 24th International Conference on Artificial Intelligence and Statistics, AISTATS 2021, April 13-15, 2021, Virtual Event*, volume 130, 2008–2016. PMLR.

Kaplan, J.; McCandlish, S.; Henighan, T.; Brown, T. B.; Chess, B.; Child, R.; Gray, S.; Radford, A.; Wu, J.; and Amodei, D. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Kullback, S.; and Leibler, R. A. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1): 79–86.

Li, R.; allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; LI, J.; Chim, J.; Liu, Q.; Zheltonozhskii, E.; Zhuo, T. Y.; Wang, T.; Dehaene, O.; Lamy-Poirier, J.; Monteiro, J.; Gontier, N.; Yee, M.-H.; Umapathi, L. K.; Zhu, J.; Lipkin, B.; Oblokulov, M.; Wang, Z.; Murthy, R.; Stillerman, J. T.; Patel, S. S.; Abulkhanov, D.; Zocca, M.; Dey, M.; Zhang, Z.; Bhattacharyya, U.; Yu, W.; Luccioni, S.; Villegas, P.; Zhdanov, F.; Lee, T.; Timor, N.; Ding, J.; Schlesinger, C. S.; Schoelkopf, H.; Ebert, J.; Dao, T.; Mishra, M.; Gu, A.; Anderson, C. J.; Dolan-Gavitt, B.; Contractor, D.; Reddy, S.; Fried, D.; Bahdanau, D.; Jernite, Y.; Ferrandis, C. M.; Hughes, S.; Wolf, T.; Guha, A.; Werra, L. V.; and de Vries, H. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research*.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Lin, J. 1991. Divergence Measures Based on the Shannon Entropy. *IEEE Transactions on Information Theory*, 37(1): 145–151.

Liu, S.; Yao, Y.; Jia, J.; Casper, S.; Baracaldo, N.; Hase, P.; Yao, Y.; Liu, C. Y.; Xu, X.; Li, H.; Varshney, K. R.; Bansal, M.; Koyejo, S.; and Liu, Y. 2025. Rethinking machine unlearning for large language models. *Nat. Mac. Intell.*, 7(2): 181–194.

Lozhkov, A.; Li, R.; Allal, L. B.; Cassano, F.; Lamy-Poirier, J.; Tazi, N.; Tang, A.; Pykhtar, D.; Liu, J.; Wei, Y.; et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173*.

Maini, P.; Feng, Z.; Schwarzschild, A.; Lipton, Z. C.; and Kolter, J. Z. 2024. TOFU: A Task of Fictitious Unlearning for LLMs. *CoRR*, abs/2401.06121.

Mohsin, A.; Janicke, H.; Wood, A.; Sarker, I. H.; Maglaras, L.; and Janjua, N. 2024. Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms. *arXiv preprint arXiv:2406.12513*.

Neel, S.; Roth, A.; and Sharifi-Malvajerdi, S. 2021. Descent-to-Delete: Gradient-Based Methods for Machine Unlearning. In *ALT*, volume 132, 931–962. PMLR.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *ACL*, 311–318. ACL.

Qi, X.; Panda, A.; Lyu, K.; Ma, X.; Roy, S.; Beirami, A.; Mittal, P.; and Henderson, P. 2025. Safety Alignment Should

be Made More Than Just a Few Tokens Deep. In *The Thirteenth International Conference on Learning Representations*.

Rafailov, R.; Sharma, A.; Mitchell, E.; Manning, C. D.; Ermon, S.; and Finn, C. 2023. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; Kozhevnikov, A.; Evtimov, I.; Bitton, J.; Bhatt, M.; Canton-Ferrer, C.; Grattafiori, A.; Xiong, W.; Défossez, A.; Copet, J.; Azhar, F.; Touvron, H.; Martin, L.; Usunier, N.; Scialom, T.; and Synnaeve, G. 2023. Code Llama: Open Foundation Models for Code. *CoRR*, abs/2308.12950.

Tarun, A. K.; Chundawat, V. S.; Mandal, M.; and Kankanhalli, M. S. 2024. Fast Yet Effective Machine Unlearning. *IEEE Trans. Neural Networks Learn. Syst.*, 35(9): 13046–13055.

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

The MITRE Corporation. 2025. Common Weakness Enumeration (CWE). https://cwe.mitre.org/. Accessed: August 1, 2025.

U.S. Copyright Office. 2024. What does copyright protect?

Wang, C.; Huang, K.; Zhang, J.; Feng, Y.; Zhang, L.; Liu, Y.; and Peng, X. 2024. How and Why LLMs Use Deprecated APIs in Code Completion? An Empirical Study. *CoRR*, abs/2406.09834.

Wang, Y.; Wei, J.; Liu, C. Y.; Pang, J.; Liu, Q.; Shah, A.; Bao, Y.; Liu, Y.; and Wei, W. 2025. LLM Unlearning via Loss Adjustment with Only Forget Data. In *The Thirteenth International Conference on Learning Representations*.

Wei, A.; Haghtalab, N.; and Steinhardt, J. 2023. Jailbroken: How Does LLM Safety Training Fail? In *NeurIPS*.

Wu, T.; Wu, W.; Wang, X.; Xu, K.; Ma, S.; Jiang, B.; Yang, P.; Xing, Z.; Li, Y.-F.; and Haffari, G. 2024. Versicode: Towards version-controllable code generation. *arXiv preprint arXiv:2406.07411*.

Xu, W.; Gao, K.; He, H.; and Zhou, M. 2025. LiCoEval: Evaluating LLMs on License Compliance in Code Generation . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 589–589.

Yao, Y.; Xu, X.; and Liu, Y. 2023. Large Language Model Unlearning. *CoRR*, abs/2310.10683.

Yao, Y.; Xu, X.; and Liu, Y. 2024. Large Language Model Unlearning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Yuan, H.; Jin, Z.; Cao, P.; Chen, Y.; Liu, K.; and Zhao, J. 2024. Towards Robust Knowledge Unlearning: An Adversarial Framework for Assessing and Improving Unlearning Robustness in Large Language Models. *CoRR*, abs/2408.10682.

Zhang, D.; Finckenberg-Broman, P.; Hoang, T.; Pan, S.; Xing, Z.; Staples, M.; and Xu, X. 2024a. Right to be forgotten in the era of large language models: Implications, challenges, and solutions. *AI and Ethics*, 1–10.

Zhang, R.; Lin, L.; Bai, Y.; and Mei, S. 2024b. Negative Preference Optimization: From Catastrophic Collapse to Effective Unlearning. In *First Conference on Language Modeling*.

Zhu, Q.; Guo, D.; Shao, Z.; Yang, D.; Wang, P.; Xu, R.; Wu, Y.; Li, Y.; Gao, H.; Ma, S.; et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931*.