Lightweight Model Editing for LLMs to Correct Deprecated API Recommendations

GUANCHENG LIN, Department of Computer Science, City University of Hong Kong, China
XIAO YU, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
JACKY KEUNG, Department of Computer Science, City University of Hong Kong, China
XING HU, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
XIN XIA, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
ALEX X. LIU, Midea Group, China

Pre-trained or fine-tuned on large code corpora, Large Language Models (LLMs) have demonstrated strong performance in code completion tasks. However, their embedded knowledge is constrained by the timeliness of training data, which often includes code using deprecated APIs. Consequently, LLMs frequently generate deprecated APIs that will no longer be supported in future versions of third-party libraries. While retraining LLMs on updated codebases could refresh their API knowledge, this approach is computationally expensive. Recently, lightweight model editing methods have emerged to efficiently correct specific knowledge in LLMs. However, it remains unclear whether these methods can effectively update deprecated API knowledge and enable edited models to generate up-to-date APIs. To address this gap, we conduct the first systematic study applying 10 state-of-the-art model editing techniques to update deprecated API knowledge in three LLMs: Qwen2.5-Coder, StarCoder2, and DeepSeek-Coder. We introduce EDAPIBench, a dedicated benchmark featuring over 70 deprecated APIs from 8 popular Python libraries, with more than 3,000 editing instances. Our results show that the parameter-efficient fine-tuning method AdaLoRA achieves the best performance in enabling edited models to generate correct, up-to-date APIs, but falls short in Specificity (i.e., the editing influences untargeted knowledge). To resolve this, we propose AdaLoRA-L, which defines "Common API Layers" (layers within the LLMs with high importance across all APIs, storing general knowledge and excluded from editing) and restricts edits exclusively to "Specific API Layers" (layers with high importance only for the target API, storing the API-specific knowledge). Experimental results demonstrate that AdaLoRA-L significantly improves Specificity while maintaining comparable performance across other evaluation metrics.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; **Software maintenance tools**.

Additional Key Words and Phrases: Large Language Model, Deprecated API, Model Editing

Authors' Contact Information: Guancheng Lin, guanchlin4-c@my.cityu.edu.hk, Department of Computer Science, City University of Hong Kong, Hong Kong, China; Xiao Yu, xiao.yu@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China; Jacky Keung, jacky.keung@cityu.edu.hk, Department of Computer Science, City University of Hong Kong, Hong Kong, China; Xing Hu, xinghu@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China; Xin Xia, xin.xia@acm.org, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China; Alex X. Liu, alexliu@midea.com, Midea Group, Foshan, China.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2026/07
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Guancheng Lin, Xiao Yu, Jacky Keung, Xing Hu, Xin Xia, and Alex X. Liu. 2026. Lightweight Model Editing for LLMs to Correct Deprecated API Recommendations. In *Proceedings of XXXXX (Conference acronym 'XX)*. ACM, New York, NY, USA, 22 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In modern software engineering, the need for rapid iteration and the complexity of systems necessitate the efficient reuse of existing solutions. A fundamental practice in this context is the reliance on third-party libraries, which provide reusable functionality through Application Programming Interfaces (APIs) [54, 59]. These libraries accelerate development but exist in a state of constant evolution—they undergo refactorings [21], bug fixes [16], security patches [51], and feature enhancements. This rapid pace of change results in frequent API updates, where older APIs are deprecated and replaced by newer versions. Taking PyTorch [4], a popular deep learning library for instance, the API `torch.svd()` is deprecated in favor of `torch.linalg.svd()`, and will be removed in a future PyTorch release. Consequently, newly developed code should avoid the deprecated `torch.svd()`, as such APIs typically lack compatibility with new features or data formats and are eventually removed in subsequent library releases [1].

This dynamic nature of APIs poses significant challenges for code-assistance tools powered by Large Language Models (LLMs). Trained on vast corpora of natural language and code, LLMs have demonstrated remarkable capabilities in understanding and generating code, enabling their widespread adoption in tasks such as code completion to accelerate software development [20, 48, 50]. However, the knowledge embedded in these models is inherently limited by the timeliness of their training data: many LLMs were trained on codebases using deprecated API versions before up-to-date iterations, leaving them with inherently outdated knowledge. Alternatively, some LLMs may have been trained on mixed datasets containing both deprecated and up-to-date API usages, but their understanding of deprecated APIs may not be fully corrected. Therefore, during code completion, LLMs frequently suggest deprecated API invocations, resulting in suboptimal or erroneous code recommendations. For instance, Wang et al. [49] report that 37.4% of GPT-3.5's API predictions correspond to deprecated APIs, underscoring the practical implications of this problem.

Motivations. To address this problem, Wang et al. [49] proposed two methods, among which `REPLACEAPI` exhibited superior performance. This method operates by detecting deprecated APIs in the LLM's code completion, removing the deprecated APIs and all subsequent tokens, then appending the corresponding up-to-date API to form a new prefix. This prefix is concatenated with the original prompt and fed back to the LLM to generate a suffix, which is finally combined with the prefix to produce a corrected completion. However, in practical scenarios, developers predominantly rely on black-box LLMs whose decoding workflows are generally non-interceptable. Further, since `REPLACEAPI` requires direct manipulation of the decoding process (e.g., token removal and prefix reconstruction), which increases inference latency, these two factors together make it less compatible with real-world LLM deployment. A more practical and straightforward solution to this problem is to retrain LLMs on the latest codebases that predominantly use up-to-date APIs, thereby overwriting outdated knowledge. However, this approach—akin to "killing a fly with a cannon"—is computationally expensive and time-consuming, rendering it impractical for frequent API updates. Recently, researchers have proposed model editing as a promising lightweight alternative that avoids full retraining. Model editing aims to update specific pieces of knowledge inside an LLM efficiently and effectively while leaving unrelated knowledge intact [53]. This technique has been primarily studied in the Natural Language Processing (NLP) domain [7, 11, 23, 31, 32, 45]. For example, in factual knowledge editing, a model might be updated to reflect a change in real-world facts—such as correcting the statement "Joe Biden is the President of the United States" to "Donald

Trump is the President"—without degrading the model's performance on other unrelated tasks. However, whether existing model editing methods can effectively update deprecated API knowledge within LLMs, and enable the edited models to correctly replace deprecated APIs with up-to-date ones, remains unexplored.

Approaches. To bridge this gap, we conduct the first systematic study on applying 10 state-of-the-art model editing techniques for updating deprecated API knowledge in three LLMs (Qwen2.5-Coder (3B), StarCoder2 (3B), and DeepSeek-Coder (1.3B)). To facilitate this, we first propose a dedicated Editing Deprecated API Benchmark, `EDAPIBench`, which can be fully automatically constructed. We begin with 145 verified API mappings (deprecated → up-to-date) from Wang et al. [49], covering eight popular Python libraries such as PyTorch and TensorFlow. Using these mappings, we extract 65,596 real-world functions from GitHub that call the up-to-date APIs. For each function, we extract lines preceding the API invocation as candidate editing inputs (to prompt LLM API completion) and the invocation line as the target API (ground truth). We then filter these inputs per LLM, retaining only inputs where the original model outputs the deprecated API—ensuring each case genuinely requires editing. This process yields over 1,000 unique LLM-specific editing instances per model. Next, we extend this core data to build four evaluation subsets that comprehensively assess model editing performance across four key dimensions: *Effectiveness*: Measuring whether the edited model generates the up-to-date API for the original inputs; *Generalization*: Measuring whether it generates the up-to-date API on semantically equivalent but syntactically varied inputs; *Portability*: Measuring whether it generates the up-to-date API across different inputs (with both syntactic and semantic differences) that involve the same deprecated API in their original completions; *Specificity*: Measuring whether it preserves consistent pre-editing behavior on inputs unrelated to the editing task. Among all evaluated model editing methods, the parameter-efficient fine-tuning approach AdaLoRA achieves the highest performance in Effectiveness, Generalization, and Portability. However, it exhibits poor Specificity, primarily because its edits tend to inadvertently modify parameters that encode general knowledge, leading to unintended changes in the model's behavior on inputs unrelated to the editing task.

To address this limitation, we propose an improved variant, AdaLoRA-L, to enhance Specificity while preserving other strengths. Its core design involves: (1) computing gradients of editable parameters for each editing instance to quantify their relevance to the target API; (2) calculating layer importance scores (average squared gradient magnitude of layer parameters) to distinguish "Common API Layers" (high importance across all APIs, storing general knowledge, excluded from editing) and "Specific API Layers" (high importance only for the target API, storing API-specific knowledge, set as editing targets); (3) restricting edits exclusively to Specific API Layers to avoid interfering with general knowledge. Results show AdaLoRA-L significantly improves Specificity by 836.2%, 33.5%, 310.2% across the three target edited models, while maintaining AdaLoRA's strong performance in Effectiveness, Generalization, and Portability.

In summary, our paper makes the following contributions:

- **Benchmark:** We construct `EDAPIBench`—the first dedicated benchmark for evaluating deprecated API knowledge editing in LLMs, with fully automated construction, serving as a standardized, rigorous platform that benefits both software engineering and broader NLP communities.
- **Evaluation:** We evaluate 10 state-of-the-art model editing techniques on three LLMs, and find AdaLoRA excels in three key dimensions but lacks Specificity due to inadvertently modifying parameters encoding general knowledge.
- **Method:** We propose AdaLoRA-L, an improved model editing approach that isolates API-specific layers via gradient-based importance scoring, addressing AdaLoRA's Specificity limitation while retaining its strengths in Effectiveness, Generalization, and Portability.

2 Task Definition

Our study frames the model editing scenario as a code completion task, with the primary goal of updating deprecated API knowledge within LLMs. In this context, we denote the pre-editing LLM as f and define an editing instance as a tuple $M = (x, y)$, where x represents the editing input (a code snippet to be completed) and y is the editing target (the specific correct, up-to-date API that should replace the deprecated one). For a given input x, the current API completion result produced by the original LLM f is denoted as y_d, which corresponds to the deprecated API call. The goal of model editing is to enable the edited LLM (denoted f_e) to generate the desired target y for the input x, effectively correcting the deprecated API call y_d. Following existing works [17, 24, 53], we define that an effective model edit should meet the following four key criteria.

Effectiveness ensures that the edited LLM f_e correctly generates the up-to-date API y for the editing input x: $f_e(x) = y$.

Generalization requires that for any input x_g which is a syntactic variation of the original input x but semantically equivalent (i.e., both x and x_g have the same meaning or behavior), and for which the original LLM f still completes with the deprecated API y_d for x_g, the edited LLM f_e should generate the correct, up-to-date API y for x_g: $f_e\left(x_g\right) = y$.

Portability requires that for any input x_p, which represents a real-world input that is both semantically and syntactically different from x and is completed by the original LLM f using the deprecated API y_d, the edited LLM f_e should generate the up-to-date API y for x_p: $f_e\left(x_p\right) = y$.

Specificity ensures that model editing on the original LLM f does not affect other irrelevant knowledge, preserving unchanged outputs for inputs irrelevant to the editing task. Let $\mathbb{U} = \{U_1, U_2, \ldots\}$ denote a set of unrelated instances, where each $U = (x_u, y_u)$ consists of a non-target input x_u, and its corresponding API completion output by the original LLM f (i.e., $y_u = f(x_u)$). Here, x_u is unrelated to the editing input x and involves no overlapping API calls with the target API y or the deprecated API y_d. The edited LLM f_e should satisfy: $f_e\left(x_u\right) = f\left(x_u\right), \forall U \in \mathbb{U}$.

3 Study Design

3.1 Model Editing Techniques Selection

The model editing methods can be categorized into four types based on the way knowledge is updated [17, 53]: **Locate-then-edit**: This common model editing paradigm first involves locating the storage positions of knowledge and then modifying the parameters of these positions to achieve knowledge modification. **Parameter-efficient Fine-tuning**: In contrast to standard full fine-tuning, this method focuses on fine-tuning only the parameters of specific components of the model, thereby reducing computational overhead and memory consumption. **Memory-based method**: This approach does not directly alter the parameters of the model; instead, it introduces an additional module to record new knowledge. When the model's input relates to the edited knowledge, the model's hidden layer state is replaced with the output of the additional module; otherwise, it remains unchanged. **Meta Learning**: This method involves training a hypernetwork to predict how parameters should shift when editing a given instance. During the editing phase, for each edited instance, the hypernetwork is utilized to calculate the parameter shift, and the predicted shifts are subsequently applied to the model's parameters to accomplish the editing process. We investigate 10 state-of-the-art model editing methods across the four categories, with brief descriptions of each provided in Table 1. These ten methods are selected to cover all methods investigated by Li et al. [24] (a recent empirical study on model editing in software engineering). Additionally, we include three extra methods—AlphaEdit, LoRA, and AdaLoRA—to enhance the generalizability of our study.

Table 1. The brief descriptions of the ten model editing methods.

Locate-then-edit	
ROME [31]	It utilizes causal tracing to identify a particular middle-layer Feed Forward Network (FFN) where knowledge is stored. ROME conceptualizes model editing as a constrained least squares problem, implementing a rank-one update to the FFN's projection weights to incorporate new knowledge.
MEMIT [32]	This is an enhanced version of ROME, allowing for multi-layer editing. It modifies key FFNs crucial for knowledge recall by distributing intended changes across these layers as key-value memories.
PMET [23]	It analyzes the hidden states of Multi-Head Self-Attention (MHSA) and FFN, showing that MHSA acts as a knowledge extractor encoding general patterns without needing weight updates. Thus, PMET focuses solely on updating FFN weights for more accurate knowledge updates.
AlphaEdit [7]	It projects parameter perturbations onto the key matrix's null space to safeguard knowledge, preserving existing information undisturbed. By prioritizing minimal update errors, it maintains hidden representation stability, preventing forgetting during editing.
Parameter-efficient Fine-tuning	
FT-L [65]	It directly fine-tunes the FFN of a single layer and minimizes the impact on irrelevant data by constraining parameter norms.
LoRA [15]	It fine-tunes the attention modules across all layers. It first freezes the model weights and introduces low-rank matrices into the attention modules, minimizing the volume of parameter updates by optimizing only these matrices.
AdaLoRA [62]	As an optimized version of LoRA, it also fine-tunes the attention modules across all layers. The difference is that it dynamically allocates more parameter update budget to critical attention layers, while assigning fewer parameter updates to less important ones.
Memory-based	
GRACE [11]	It introduces a module with a discrete key-value codebook for storing edited knowledge and a deferral mechanism. This mechanism enables the codebook to identify modified key-value pairs, whose values then replace the hidden layer output of the original model.
A-GRACE [24]	This is an improved version of GRACE. By incorporating a contrastively-trained encoder, it successfully boosts GRACE's generalization without significantly impacting other performance dimensions.
Meta Learning	
MALMEN [45]	Instead of computing gradients directly to fine-tune model parameters, it uses a hypernetwork to calculate parameter shifts for editing instances, applying these shifts to the model.

3.2 Editing Subjects

We select the three popular open-source code LLMs as our editing subjects, which are widely adopted in various software engineering tasks. **Qwen2.5-Coder (3B)** [18]: Proposed by Alibaba and released in September 2024, it is trained on 5.5 trillion tokens collected before February

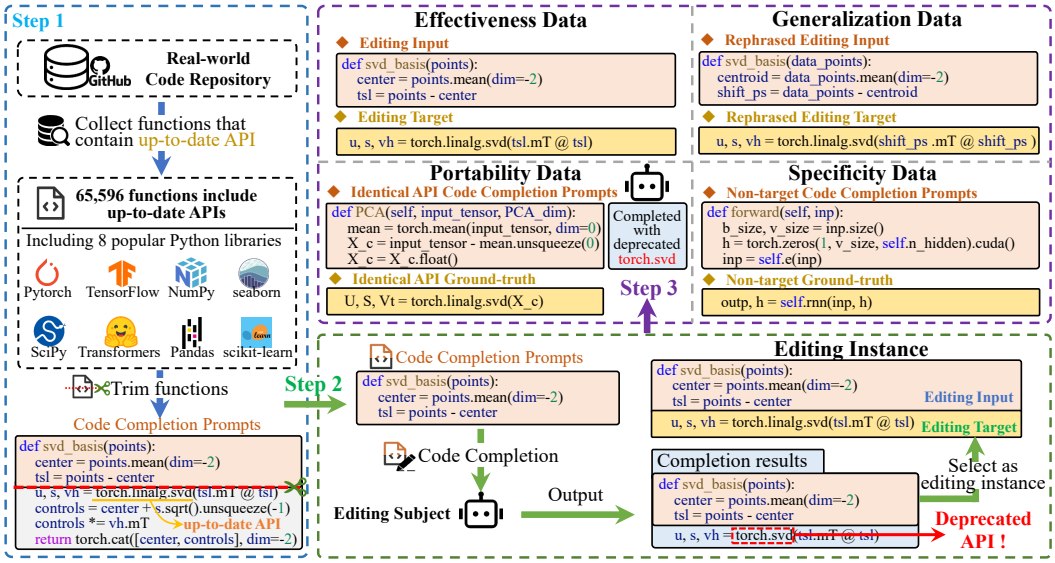


Fig. 1. The construction process of EDAPIBench.

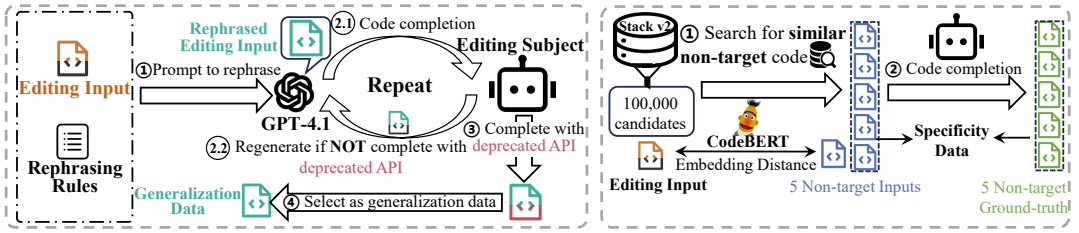
2024, covering source code, text-code grounding data, synthetic data, and other relevant datasets. **StarCoder2 (3B)** [29]: Developed by BigCode and released in January 2024, it is trained on 17 programming languages from The Stack v2 [29] up to September 2023, using the fill-in-the-middle objective and covering over 3 trillion tokens. **DeepSeek-Coder (1.3B)** [10]: Developed by DeepSeek and released in June 2024, it is trained from scratch on 2 trillion tokens collected before February 2023, with the training data consisting of 87% code and 13% natural language (English and Chinese).

3.3 EDAPIBench Construction

We construct EDAPIBench, which comprises over 1,000 editing instances $M = (x, y)$ per LLM. For each instance, the editing input x is a code completion prompt—specifically, a prompt that should yield the target up-to-date API y, but which the original LLM f completes with the deprecated API y_d. The editing target is the corresponding up-to-date API y. Each editing instance serves as core data to evaluate the Effectiveness of model editing; additionally, for every instance, we construct Generalization, Portability, and Specificity datasets to assess the performance of the edited LLM f_e across the three key dimensions. Figure 1 illustrates the construction process of EDAPIBench.

Step 1: Code Completion Prompts Collection. This step focuses on collecting code snippets that can serve as editing inputs x—specifically, prompts that require the up-to-date API y for correct completion. To support this, we directly adopt the 145 API mappings collected from Wang et al. [49]: these authors reviewed the documentation and change logs of each library, manually identified deprecated API y_d occurrences, and paired each y_d with its corresponding up-to-date replacement y. The mappings cover eight popular Python libraries (with specific version ranges shown in Figure 3): PyTorch, TensorFlow, scikit-learn, SciPy, Pandas, Seaborn, Transformers, and NumPy. We then use the Sourcegraph tool[1] to search open-source Python repositories for real-world code snippets that use the up-to-date APIs from these mappings. This process yields 65,596 functions containing calls to the target up-to-date APIs. For each function, we identify the line where the up-to-date API is invoked: the lines preceding this invocation line serve as candidate editing inputs x, while the invocation line itself acts as the ground truth for the target API y.

[1]https://sourcegraph.com/search



(a) Generalization Data (b) Specificity Data

Fig. 2. The construction process of generalization data and specificity data in EDAPIBench.

Step 2: Data Filtering. A critical prerequisite is ensuring each of the three editing subjects (i.e., the three LLMs introduced in Section 3.2) generates deprecated APIs in response to prompts prior to model editing. To verify this, we analyze completions by inputting each candidate editing input into each LLM separately and prompting each to complete the input three times with temperature set to 0 (using greedy sampling). For each model, we retain only those prompts where the model outputs the deprecated API in all three completions. This yields valid, LLM-specific editing instances tailored to each model that truly require model editing. For example, as shown in Figure 1, the candidate editing input `def svd_basis(points): ... tsl=points-center` is retained since the LLM consistently generates the deprecated API `torch.svd()` in all three completions.

Step 3: Evaluation Data Preparation.

1) Effectiveness Data. The editing instances $M = (x, y)$ themselves constitute the Effectiveness data. After model editing, Effectiveness is evaluated by checking whether the edited LLM f_e can correctly complete the prompt x with the target API y.

2) Generalization Data. The generalization dimension evaluates whether the edited model f_e can successfully complete the up-to-date API call y based on code completion prompts that are semantically identical but syntactically different from the original editing input x. This requires rephrasing the code of the original editing input while preserving their meanings and behaviors. Inspired by Yu et al. [57], who proposed 18 rephrasing rules capable of altering code while preserving semantic and syntactic naturalness, we adopt 14 of these rules, excluding four that are inapplicable to Python language (e.g., converting "switch-case" to "if-else"). As illustrated in Figure 2, and following Wang et al. [49], we provide these 14 rephrasing rules to GPT-4.1 [2], instructing it to apply them as extensively as possible to generate rephrased editing inputs that are maximally dissimilar from the original. We then prompt the editing subjects (i.e., the LLMs) to complete these rephrased inputs. If the completion contains no deprecated API y_d, GPT-4.1 is instructed to rephrase the input again until the model completes it with the deprecated API in all three completions. This validated rephrased input then serves as generalization data. For example, as shown in Figure 1, the rephrased input appears as `def svd_basis(data_points): ... shift_ps=data_points-centroid`.

3) Portability Data. The portability dimension evaluates whether the edited model can correctly complete the up-to-date API call on different editing instances where the original model also completes the prompt with the same deprecated API. For each editing instance $M = (x, y)$, we randomly select another editing instances with the same target API y from EDAPIBench as portability evaluation data. For example, as shown in Figure 1, one such selected editing instance is: x= "`def PCA (self, input_tensor, PCA_dim): X_c=X_c.float()` ", y="`torch.linalg.svd()`".

4) Specificity Data. The Specificity dimension requires the edited LLM f_e to preserve consistent outputs for non-target inputs—defined as inputs for which the original LLM f does not generate either the target API y or the deprecated API y_d during completion. In line with Wang et al. [49], our benchmark focuses on the most challenging scenario: whether f_e exhibits output shifts for non-target inputs that are semantically similar to the original editing input x. As shown in Figure 2, we randomly select 100,000 Python files from The Stack v2 dataset [29] as non-target input candidates.

Table 2. The number of instances and APIs for different LLMs across evaluation dimensions in EDAPIBench.

Models	Qwen2.5-Coder		StarCoder2		DeepSeek-Coder	
Statistical value	# instances	# APIs	# instances	# APIs	# instances	# APIs
Effectiveness	1,401	75	1,016	76	1,072	76
Generalization	1,401	75	1,016	76	1,072	76
Portability	1,390	67	1,001	62	1,064	65
Specificity	7,005	3,375	5,080	2,672	5,360	2,626

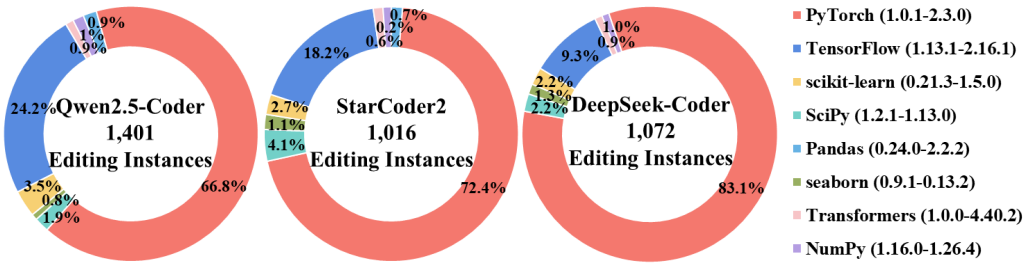


Fig. 3. The distribution of the number of target APIs from different libraries in EDAPIBench.

Since The Stack v2 has an extremely large scale, full retrieval would be computationally expensive, so this sampling approach is adopted for practicality. To collect semantically similar non-target inputs, we employ CodeBERT [8]—a widely used code embedding model in software engineering—to generate embeddings for each editing input x. For every x, we select five non-target inputs with the smallest embedding distance from the candidates, while ensuring that these inputs contain neither the target API y nor the deprecated API y_d from the corresponding editing instance. We then instruct the original LLMs (editing subjects) to complete these five non-target inputs prior to model editing, and preserve their API outputs as "non-target ground truth". These non-target inputs and their corresponding ground truth together constitute the Specificity data. For example, as shown in Figure 1, the non-target API for one piece of Specificity data is `self.nn()`.

Dataset Statistics. Table 2 presents the number of editing instances and APIs included in EDAPIBench. The volume of Generalization data matches that of Effectiveness data, as each instance has a corresponding rephrased version. The volume of Specificity data is five times that of Effectiveness data, since we select five non-target inputs per instance. Portability data volume is slightly smaller than that of Effectiveness data, as some APIs have only one editing instance—preventing the selection of another instance with the same deprecated API. Figure 3 illustrates the distribution of editing instances across libraries. Instances with PyTorch API calls account for the largest proportion, followed by those with TensorFlow. These popular deep learning libraries, which update frequently, naturally have more deprecated APIs. In contrast, classic data processing libraries such as NumPy and Pandas have relatively fewer deprecated APIs.

3.4 Evaluation Metrics

The performance of model editing is evaluated based on how closely the single-line completion outputs of edited LLMs match the ground-truth. To quantify the performance, following Li et al. [24], we utilize three common metrics: **Exact Match** (EM), **BLEU** [37], and **ROUGE-L** [25]. Additionally, we introduce **API Exact Match** (AEM), specifically tailored for the EDAPIBench scenario. EM measures whether the LLM's single-line completion exactly matches the entire ground-truth line at the token level, assigning a score of 1 for a perfect match and 0 otherwise. AEM is a variant of EM that only checks whether the single-line completion contains the ground-truth API, regardless

of other tokens. BLEU [37] and ROUGE-L [25] are two commonly adopted evaluation metrics in code-related tasks [5, 46]. Specifically, BLEU measures the n-gram overlap between the single-line completion and the entire ground-truth line. ROUGE-L is used to calculate the longest common subsequence between the single-line completion and the entire ground-truth line. Both BLEU and ROUGE-L values range from 0 to 1, with higher scores indicating greater overlap and better consistency between the single-line completion and the entire ground-truth line.

3.5 Experiment Settings

All experiments are conducted on RTX 4090 GPUs paired with an Intel Core i9 processor. Following existing code completion works [58, 60], we use incomplete code snippets directly as prompts for code completion, without adding any additional instructions. For the method requiring training (i.e., MALMEN), we follow existing works [6, 24, 64], splitting EDAPIBench into training and test sets at a 1:1 ratio using a two-fold cross-validation approach, and the results of the two rounds of testing are integrated to serve as the final results. Regarding the selection of editing layers for locate-then-edit methods, following Li et al. [24], we maintain consistency with other benchmarks [52, 53] and do not perform additional causal intervention [31] to identify the editing layers, as these studies have already leveraged causal intervention to locate the key layers. For the hyperparameters of model editing methods, we keep them consistent with the settings of existing model editing benchmarks [24, 52, 53]. The details of the hyperparameter settings are as follows.

ROME, MEMIT, PMET, AlphaEdit: In the case of ROME, we set the 5th layer as the editing layer, while for the other three methods, we choose the {4, 5, 6, 7, 8}-th layers as the editing layers. All methods are trained with 25 iterations and a learning rate of 5e-1.

GRACE, A-GRACE: We insert an adapter into the down projection matrix of the 27th layer. The initial deferral radius of the key is set to 1, and the value corresponding to this key is fine-tuned for 30 iterations with a learning rate of 1. For the additional encoder introduced in A-GRACE, we follow the settings in the original study [24], with a hidden layer dimension of 256, and train it for 100 epochs at a learning rate of 1e-4.

MALMEN: The projection matrices of the last 5 layers of LLMs serve as the update targets for MALMEN. Initially, a hypernetwork containing two MLPs with a hidden layer dimension of 1920 is trained on the training set for predicting parameter shifts. This training process involves 1,000 iterations and a learning rate of 1e-5. During editing, the hidden states of the editing instance at the editing layers are cached first. Subsequently, the hypernetwork predicts the parameter shifts to facilitate the weight update process.

FT-L, LoRA, AdaLoRA: We set the MLP in the 21st layer as the editing layer for FT-L to conduct fine-tuning, with 40 epochs and a learning rate of 5e-4. For LoRA and AdaLoRA, consistent with the settings in [15], we select the query matrices and value matrices of the attention modules across all layers as the fine-tuning targets, with 30 training epochs, a rank of 8, and a learning rate of 5e-3.

To reduce randomness, all methods are run five times. The median of the five runs is reported as the final performance on `EDAPIBench`, presented in the subsequent tables and figures.

4 Experimental Results

We organize the results by addressing the Research Questions (RQs), used as titles for each section.

4.1 RQ1: How do the methods perform at editing deprecated API knowledge in LLMs?

As shown in Tables 3, 4, and 5, no single method consistently excels across all four dimensions: Effectiveness, Generalization, Portability, and Specificity. Take AdaLoRA as an example—a standout method that achieves the best overall performance in three dimensions (Effectiveness, Generalization, and Portability). However, its Specificity is noticeably lower than that of methods like GRACE.

Table 3. The performance of the methods on editing deprecated API knowledge within Qwen2.5-Coder.

Editor	Effectiveness				Generalization				Portability				Specificity			
	AEM	EM	BU	RL	AEM	EM	BU	RL	AEM	EM	BU	RL	AEM	EM	BU	RL
Pre-editd	2.7	0.7	40.4	63.4	5.1	1.0	36.9	58.2	2.6	0.7	39.8	62.9	96.8	76.4	97.6	97.2
ROME	2.0	0.4	6.6	17.5	2.9	0.9	7.9	18.8	4.4	1.8	15.0	29.1	5.0	1.0	40.9	52.2
MEMIT	6.5	2.3	26.4	46.0	6.7	2.4	26.1	44.4	4.8	1.3	36.3	58.0	70.5	45.5	89.1	91.9
PMET	5.9	2.2	35.0	56.3	7.2	2.1	33.1	53.0	3.4	0.9	38.6	61.4	85.8	63.2	94.4	95.1
AlphaEdit	7.1	2.9	15.8	31.9	6.7	2.6	17.8	33.5	5.8	1.5	36.2	58.3	52.8	28.2	82.6	87.2
GRACE	97.8	55.5	73.0	80.9	6.3	1.4	37.3	58.4	2.8	1.0	39.9	62.9	**96.5**	**75.1**	**97.5**	**97.2**
A-GRACE	98.0	57.4	74.3	81.8	70.3	31.9	60.9	72.8	19.5	3.8	39.4	59.1	84.9	65.6	92.7	94.3
MALMEN	53.8	15.1	46.0	60.7	48.6	10.1	40.5	54.1	42.9	9.4	41.9	57.7	13.4	3.1	57.9	68.4
FT-L	58.4	33.8	67.4	78.7	48.8	22.3	56.2	69.7	16.2	5.9	44.8	**65.6**	86.8	64.4	95.0	95.5
LoRA	24.0	0.0	9.7	26.1	17.4	0.0	6.6	20.2	7.8	0.0	2.3	10.6	0.0	0.0	0.1	1.3
AdaLoRA	**98.1**	**60.4**	**81.0**	**87.6**	**89.5**	**39.3**	**69.9**	**77.7**	**71.4**	**11.7**	**49.7**	64.2	5.8	0.8	47.9	58.3

Conversely, while GRACE excels in Specificity, it exhibits a clear performance gap compared to AdaLoRA in Generalization and Portability. Although A-GRACE improves upon GRACE's Generalization, the gap between A-GRACE and AdaLoRA in Portability remains considerable, with a difference of 0.363 to 0.519 in AEM. This limitation likely stems from its contrastive training process: the model primarily optimizes for generalization data involving simple rephrasing, where the core context of the input remains largely consistent with the original editing instance. Consequently, it only learns to adapt to such straightforward rephrasing patterns. When confronted with portability data—inputs that differ substantially from the original editing input in both semantics and syntax, A-GRACE struggles to effectively apply the edited API knowledge to these more complex, scenario-specific cases. Additionally, compared to AdaLoRA, LoRA was originally designed as a general-purpose parameter-efficient fine-tuning method suitable for tasks such as text classification and general generation. Its goal is low-cost adaptation rather than precise knowledge updating. Regardless of the importance of parameters to the target API knowledge, LoRA adjusts all parameters using low-rank matrices of the same dimension, which leads to suboptimal overall performance in this context. While locate-then-edit methods perform well in NLP, they yield poor results on our benchmark. The main reason is that these methods are designed to edit factual knowledge in natural language, which can typically be decomposed into triple structures of "subject–relation–object," and they perform model editing based on such triples. However, API knowledge in the code domain lacks such fixed linguistic structures, making it difficult for these methods to generalize to the tasks in this benchmark. Meta-learning methods, on the other hand, show no clear advantages or obvious shortcomings across the four dimensions, resulting in mediocre overall performance.

> **Answer to RQ1:** No single method performs well across all the four dimensions. AdaLoRA achieves the best overall performance in Effectiveness, Generalization, and Portability, but performs poorly in terms of Specificity.

4.2 RQ2: What is the efficiency of the methods for deprecated API knowledge editing?

Figure 4 illustrates the average time cost per editing instance for each model editing method. Methods in the locate-then-edit category incur the highest time costs: PMET, the most costly, averages 31.8 seconds per edit across models, while ROME—more efficient than peers in this category—still averages 8.8 seconds per edit. Among all methods, MALMEN demonstrates the lowest time cost, with an average editing time of less than 1 second per edit. Memory-based methods and parameter-efficient fine-tuning approaches show similar time costs, ranging from 2.3 to 5.6 seconds per edit. AdaLoRA-L (averaging 3.8 seconds per edit) is faster than AdaLoRA (averaging 5.1

Table 4. The performance of the methods on editing deprecated API knowledge within StarCoder2.

Editor	Effectiveness				Generalization				Portability				Specificity			
	AEM	EM	BU	RL	AEM	EM	BU	RL	AEM	EM	BU	RL	AEM	EM	BU	RL
Pre-editd	3.4	1.0	36.4	59.8	4.4	1.2	32.8	53.0	2.1	0.5	34.0	57.3	91.8	89.6	98.1	96.1
ROME	4.2	0.4	7.9	26.8	3.7	0.3	9.1	25.3	2.3	0.0	5.3	18.4	7.3	2.1	51.7	61.7
MEMIT	4.4	0.4	8.9	24.5	3.1	0.7	9.2	23.8	3.6	0.7	30.1	51.1	65.6	53.1	89.8	90.7
PMET	5.2	1.0	14.8	30.3	4.1	1.2	16.9	32.5	2.6	0.8	32.0	54.5	80.1	70.2	94.5	93.7
AlphaEdit	2.1	0.5	3.1	13.4	1.9	0.2	3.0	11.9	3.8	1.2	25.7	45.7	36.2	24.0	74.8	80.3
GRACE	18.3	16.5	45.9	65.8	4.4	1.2	32.8	53.0	2.1	0.5	33.7	57.0	**90.9**	**87.9**	**97.8**	**96.0**
A-GRACE	29.8	27.0	53.6	70.7	23.9	13.8	43.3	59.9	13.9	1.5	35.6	56.1	75.4	71.3	92.8	92.1
MALMEN	28.5	4.6	34.1	51.1	25.6	4.1	31.2	46.4	25.5	3.9	34.3	51.7	22.3	9.7	67.7	75.2
FT-L	80.8	37.7	71.8	79.9	72.9	26.3	61.1	70.3	55.0	**9.6**	**47.7**	**63.8**	60.5	44.0	89.0	90.9
LoRA	16.9	0.0	8.0	24.3	11.8	0.0	5.1	18.6	4.5	0.0	1.6	9.2	0.0	0.0	0.0	1.1
AdaLoRA	**83.4**	**67.9**	**80.3**	**84.6**	**81.1**	**44.3**	**71.1**	**76.3**	**64.2**	10.5	45.4	59.9	24.5	9.9	71.1	78.1

Table 5. The performance of the methods on editing deprecated API knowledge within DeepSeek-Coder.

Editor	Effectiveness				Generalization				Portability				Specificity			
	AEM	EM	BU	RL	AEM	EM	BU	RL	AEM	EM	BU	RL	AEM	EM	BU	RL
Pre-editd	3.3	0.8	37.2	59.2	4.3	0.8	35.3	55.5	2.2	0.4	37.7	60.0	51.2	34.3	85.8	88.5
ROME	0.3	0.0	0.4	3.4	0.2	0.0	0.3	3.0	1.3	0.3	5.0	14.5	3.1	0.9	28.5	38.9
MEMIT	4.8	0.7	21.5	42.3	4.3	0.6	20.4	39.3	3.7	0.4	35.8	57.7	45.4	29.1	83.4	86.9
PMET	5.5	0.8	32.6	54.4	5.9	1.2	31.7	51.5	2.6	0.5	37.5	59.5	48.0	32.6	84.7	87.9
AlphaEdit	0.8	0.1	1.1	7.7	0.4	0.0	1.0	7.1	4.0	0.4	27.8	47.7	20.3	10.4	65.8	73.2
GRACE	71.6	39.8	73.0	83.9	6.9	1.9	36.7	56.8	2.2	0.4	37.8	60.1	**50.7**	**34.2**	**85.8**	**88.4**
A-GRACE	70.8	38.8	72.5	83.9	40.0	15.7	52.1	68.0	13.7	1.5	39.6	60.1	49.6	33.1	85.5	88.2
MALMEN	38.6	4.8	35.0	49.9	37.8	4.7	32.0	45.8	38.6	4.5	37.2	53.1	7.8	2.1	51.1	61.9
FT-L	57.2	12.7	54.0	69.1	52.4	9.1	48.3	63.3	40.8	6.7	47.8	**65.2**	50.1	33.8	85.5	88.4
LoRA	7.7	0.9	6.5	12.1	7.7	0.4	5.0	9.8	4.2	0.0	1.5	4.2	0.1	0.0	0.0	0.7
AdaLoRA	**98.3**	**89.5**	**94.7**	**96.5**	**93.9**	**56.9**	**84.2**	**88.1**	**58.0**	**7.8**	**45.5**	61.5	8.8	2.3	54.6	64.3

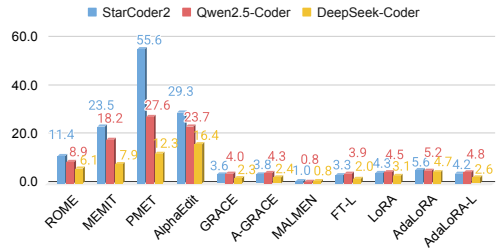


Fig. 4. The average time cost (seconds) of the model editing methods per edit.

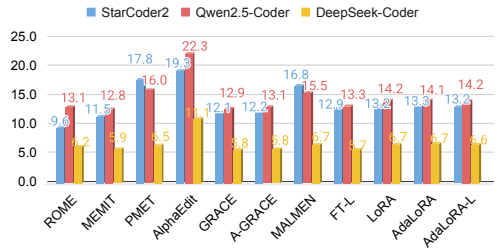


Fig. 5. The average peak memory cost (GB) of the model editing methods per edit.

seconds per edit), primarily because it edits fewer parameters, resulting in reduced computational load. Figure 5 presents the average peak memory cost per editing instance for each method. PMET and AlphaEdit have the highest memory cost (13.5GB and 17.5GB on average across models), while ROME is the most memory-efficient at 5.7GB. The remaining methods have similar memory cost, averaging around 10 GB across the three models.

> **Answer to RQ2:** The model editing methods generally have low time and memory overheads, enabling a single edit on a 3B-parameter model to be completed within a few seconds using a GPU with 24 GB of memory.

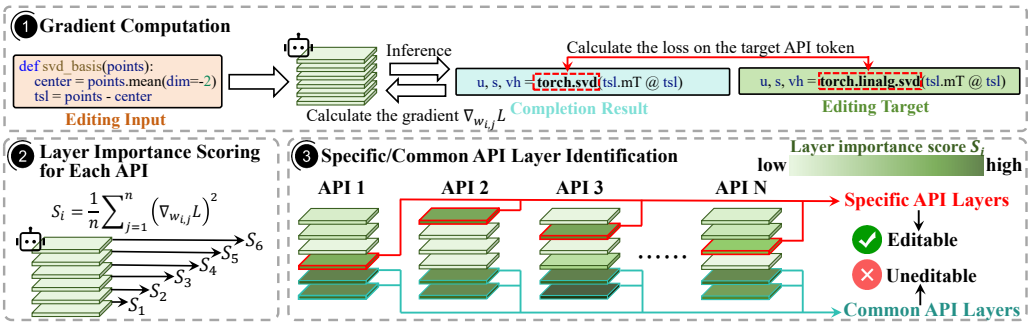


Fig. 6. The identification process of Specific API Layers and Common API Layers.

4.3 RQ3: How to improve the performance of AdaLoRA?

4.3.1 Approach. To address RQ3, we aim to improve the performance of AdaLoRA, which exhibits the highest Effectiveness, Generalization, and Portability among all methods explored in RQ1. Despite these strengths, AdaLoRA performs poorly in terms of Specificity. This limitation likely stems from its indiscriminate editing of parameters across all layers: such broad modifications alter parameters that store general knowledge, disrupting target API-irrelevant information and thereby leading to poor Specificity. A natural approach to enhance Specificity is to reduce the number of layers edited by AdaLoRA, thereby limiting the scale of parameter updates and avoiding unnecessary changes to general knowledge. However, selecting which layers to edit presents a critical challenge: failing to select layers relevant to the target API knowledge would degrade Effectiveness, Generalization, and Portability, while selecting layers that store general knowledge for editing would still compromise Specificity. To overcome this challenge, we propose AdaLoRA-L, an improved variant of AdaLoRA. It first precisely identifies layers storing general knowledge, then identifies API-specific layers for each target API from the remaining layers, and restricts editing operations exclusively to these API-specific layers. This design ensures that Specificity is enhanced without sacrificing performance in other dimensions. The overall layer selection process for AdaLoRA-L is illustrated in Figure 6.

Gradient Computation: For each editing instance, we have the LLM perform code completion on the editing input. We then calculate the loss based on the output—specifically, we calculate the loss on the target API token (i.e., the tokens corresponding to the up-to-date API in the editing target) while ignoring loss from other tokens. Next, we backpropagate this loss to calculate the gradient for each editable parameter across all layers. Consistent with neural network pruning methods [35, 63], which employ gradients as a metric to quantify parameter importance with respect to specific samples, these computed gradients directly indicate a parameter's relevance to the target API: a larger absolute gradient value signifies that even minor adjustments to the parameter would substantially impact the model's capacity to output the correct up-to-date API token, meaning such parameters are critical for updating the current API.

Layer Importance Scoring for Each API: Using these parameter gradients, we further compute the layer importance score with the following formula: $S_i = \frac{1}{n}\sum_{j=1}^{n}\left(\nabla_{w_{i,j}} L\right)^2$, where S_i denotes the importance score of the i-th layer, L is the loss computed on the up-to-date API tokens, w_{ij} represents the j-th editable parameter in the i-th layer, and n is the total number of editable parameters in that layer. Intuitively, this score quantifies the average squared gradient magnitude of all editable parameters in the layer, reflecting the layer's overall importance for enabling the model to predict the up-to-date API token.

For each editing instance, we calculate the importance scores of all layers. Then, for each API, we average the scores across all its editing instances to obtain API-specific layer importance scores,

which intuitively indicate each layer's relevance to the API. By analyzing these scores across all APIs in our EDAPIBench, we identify:

Common API Layers: Layers that exhibit high importance across all APIs. These layers encode general knowledge critical to all APIs and are excluded from editing;

Specific API Layers: Excluding Common API Layers, these are layers that demonstrate high importance only for the target API. These layers store the target API-specific knowledge and serve as the core targets for editing.

Through this layer identification and selection process, AdaLoRA-L precisely isolates a set of specific API layers for each target API. These layers constitute the focused editing scope, ensuring parameter updates concentrate on API-specific knowledge while avoiding interference with general knowledge layers. This design enhances the Specificity of model editing and maximally preserves performance in Effectiveness, Generalization, and Portability.

4.3.2 Results. We set hyperparameters for AdaLoRA-L on the three models as follows: the number of frozen Common API Layers is set to 8, 8, and 10 for Qwen2.5-Coder, DeepSeek-Coder, and StarCoder2, respectively; the number of edited Specific API Layers is uniformly set to 8 across all three models. Table 6 presents the performance improvements of AdaLoRA-L compared to AdaLoRA. To evaluate the statistical significance of the differences between AdaLoRA-L and AdaLoRA, we employ the Wilcoxon signed-rank test [55] with the Benjamini-Hochberg correction procedure [12]. An asterisk (*) following the performance value indicates a statistically significant difference. In terms of Effectiveness: AdaLoRA-L achieves performance gains on two models, except for a slight underperformance relative to AdaLoRA on StarCoder2. In terms of Generalization: It outperforms AdaLoRA on two models, with only a minor deficit observed on DeepSeek-Coder. Notably, in both Portability and Specificity, AdaLoRA-L consistently surpasses AdaLoRA across all three LLMs. Specifically, Portability shows relative improvements of 6.9%, 4.7%, and 29.8% on the three models in terms of AEM. Specificity exhibits substantial relative gains of 836.2%, 33.5%, and 310.2% in AEM. Moreover, statistical tests confirm that these improvements are significant. This result clearly demonstrates that our proposed layer localization method not only substantially enhances the Specificity of AdaLoRA but also maximally preserves its performance in the other three dimensions. It is worth noting that while AEM values are not particularly high, BLEU and ROUGE-L scores reach above 0.762 (with a perfect match at 1), confirming that AdaLoRA-L still achieves reasonable Specificity. Furthermore, the API updating method `REPLACEAPI` proposed by Wang et al. [49] only achieves an EM score of 0.422 and 0.317 in terms of Effectiveness on StarCoder2 and DeepSeek-Coder respectively, as reported in the paper, which lags behind AdaLoRA-L.

> **Answer to RQ3:** AdaLoRA-L, leveraging the localization of Specific API Layers and Common API Layers, accurately identifies the storage locations of knowledge for different APIs, enabling targeted editing of API knowledge. This approach not only effectively improves Specificity but also maximally retains performance in Effectiveness, Generalization, and Portability.

5 Discussion

5.1 Impact of Hyperparameters on the Performance of AdaLoRA-L

For RQ3, we set the number of frozen Common API Layers to 8 for DeepSeek-Coder and Qwen2.5-Coder, and 10 for StarCoder2, while unifying the number of edited Specific API Layers to 8 across all models. In the experiment, we first fix the number of edited Specific API Layers at 8, then adjust the number of frozen Common API Layers within the range of 0–12. As shown in Figure 7, increasing the number of frozen Common API Layers enhances the model's Specificity (with a less obvious trend for StarCoder2). Meanwhile, all models exhibit minimal changes in Effectiveness and Generalization—this is due to the fact that Generalization here targets edited inputs that only

Table 6. The performance comparison of AdaLoRA-L and AdaLoRA.

Editor	Effectiveness				Generalization				Portability				Specificity			
	AEM	EM	BU	RL	AEM	EM	BU	RL	AEM	EM	BU	RL	AEM	EM	BU	RL
Qwen2.5-Coder																
Pre-editd	2.7	0.7	40.4	63.4	5.1	1.0	36.9	58.2	2.6	0.7	39.8	62.9	96.8	76.4	97.6	97.2
AdaLoRA	98.1	60.4	81.0	87.6	89.5	39.3	69.9	77.7	71.4	11.7	49.7	64.2	5.8	0.8	47.9	58.3
AdaLoRA-L	**98.2**	**63.9**	**84.6**	**89.7**	**90.8**	**42.4**	**74.9**	**81.2**	**76.3**	**15.3**	**57.6**	**71.3***	**54.3***	**29.8***	**85.0***	**88.8***
StarCoder2																
Pre-editd	3.4	1.0	36.4	59.8	4.4	1.2	32.8	53.0	2.1	0.5	34.0	57.3	91.8	89.6	98.1	96.1
AdaLoRA	**83.4**	**67.9**	**80.3**	84.6	81.1	44.3	71.1	76.6	64.2	10.5	45.4	59.9	24.5	9.9	71.1	78.1
AdaLoRA-L	82.9	65.0	79.3	**84.9**	**81.8**	**45.5**	**73.5**	**78.7**	**67.2**	**11.2**	**47.6**	**61.8**	**32.7***	**15.8***	**76.2***	**81.7***
DeepSeek-Coder																
Pre-editd	3.3	0.8	37.2	59.2	4.3	0.8	35.3	55.5	2.2	0.4	37.7	60.0	51.2	34.3	85.8	88.5
AdaLoRA	98.3	89.5	94.7	96.5	**93.9**	**56.9**	84.2	88.1	58.0	7.8	45.5	61.5	8.8	2.3	54.6	64.3
AdaLoRA-L	**98.4**	**89.7**	**95.1**	**96.9**	93.0	55.6	**85.1**	**88.6**	**75.3**	**9.5**	**52.6**	**67.9***	**36.1***	**20.7***	**79.5***	**84.4***

undergo syntactic rephrasing (the core of the input related to API editing remains unchanged), and both metrics depend on API-specific knowledge stored in Specific API Layers. The general knowledge in Common API Layers has low relevance to this kind of input scenario, so it has little impact on the two metrics. In contrast, Portability decreases as the number of frozen Common API Layers increases. Portability reflects the model's ability to handle any input x_p, which represents a real-world input that is both semantically and syntactically different from the original editing input x and is completed by the original LLM f using the deprecated API y_d. Compared to the inputs for Generalization, such inputs require a greater amount of general knowledge to be processed effectively. Therefore, as more Common API Layers are frozen, the model's Portability declines.

Subsequently, with the number of frozen Common API Layers fixed, we further vary the number of edited Specific API Layers from 1 to 10. In Figure 8, a consistent pattern emerges across all three models: as the number of edited Specific API Layers increases, Effectiveness, Generalization, and Portability all improve, while Specificity noticeably declines. This phenomenon can be explained as follows: increasing the number of edited layers expands the scale of parameter updates in the model. On one hand, extensive parameter updates are more likely to disrupt knowledge unrelated to the target API, thereby reducing Specificity; on the other hand, involving more parameters in updates enhances the success rate of editing tasks and enables the edited effects to generalize better to other samples, ultimately boosting the three metrics. Based on these experimental results and balancing all performance dimensions, we finalize the hyperparameters for the three models as follows: the number of frozen Common API Layers is set to 8 for DeepSeek-Coder and Qwen2.5-Coder, and 10 for StarCoder2; the number of edited Specific API Layers is uniformly set to 8 across all models.

5.2 Impact of Different Libraries on Model Editing Performance

This section analyzes the editing performance of AdaLoRA-L on APIs from different libraries. To ensure statistical significance, we focus on the four most sample-rich libraries: PyTorch, SciPy, scikit-learn, and TensorFlow. Figures 9, 10, and 11 present the API Exact Match (AEM) results across the four evaluation dimensions (Effectiveness, Generalization, Portability, and Specificity) for the three models, respectively. Our observations reveal that Effectiveness and Generalization show little variation across libraries. In contrast, Portability and Specificity exhibit more noticeable

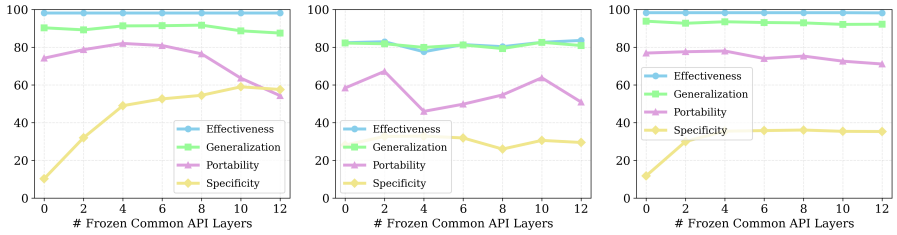


Fig. 7. The impact of the number of frozen Common API Layers on AdaLoRA-L.

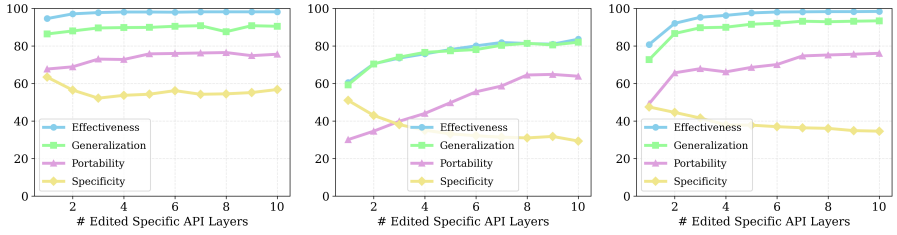


Fig. 8. The impact of the number of edited Specific API Layers on AdaLoRA-L.

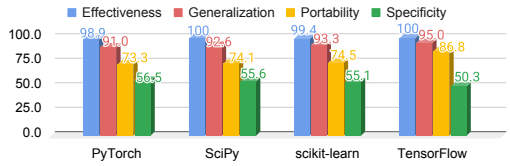


Fig. 9. The performance of AdaLoRA-L across different libraries on Qwen2.5-Coder.

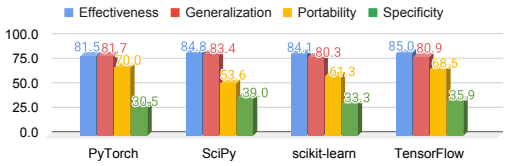


Fig. 10. The performance of AdaLoRA-L across different libraries on StarCoder2.

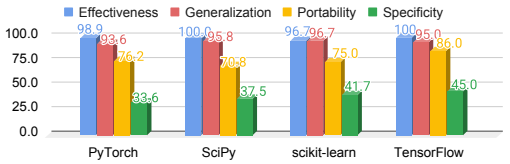


Fig. 11. The performance of AdaLoRA-L across different libraries on DeepSeek-Coder.

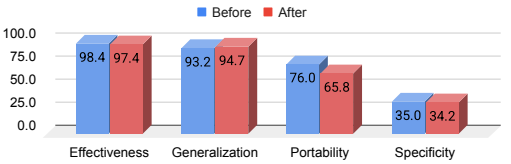


Fig. 12. The impact of API deprecation timing (Before vs. After LLM training data cutoff) on AdaLoRA-L.

differences across libraries. For example, on DeepSeek-Coder and StarCoder2, PyTorch achieves notably lower Specificity scores (33.6 and 30.5, respectively) compared to TensorFlow (45.0 and 35.9). Across all three models, SciPy consistently exhibits the lowest Portability, averaging 66.2, whereas TensorFlow achieves the highest average Portability of 80.4. Overall, while no strong or consistent patterns emerge across libraries, AdaLoRA-L demonstrates robust performance across all four evaluation dimensions for these libraries, confirming its capability to effectively edit deprecated API knowledge regardless of the library.

5.3 Impact of API Deprecation Timing on Model Editing Performance

This section investigates whether the timing of API deprecation affects the performance of deprecated API knowledge editing. We first analyze the official deprecation dates of each deprecated API in EDAPIBench, categorizing them as deprecated either before or after each model's training data cutoff. Statistics show all APIs are deprecated before StarCoder2 and Qwen2.5-Coder's cutoffs. For DeepSeek-Coder, 6 APIs (38 editing instances) are deprecated post-cutoff, while 70 APIs (1,033 instances) are deprecated pre-cutoff. Thus, Figure 12 presents AdaLoRA-L results on

DeepSeek-Coder across these two groups in terms of AEM. The results show that the deprecation timing relative to DeepSeek-Coder's cutoff has minimal impact on Effectiveness, Generalization, or Specificity. However, Portability is slightly lower for APIs deprecated post-cutoff. A plausible explanation is as follows: during DeepSeek-Coder's pre-training, the model only encountered non-deprecated usages of these 6 post-cutoff APIs, without exposure to their deprecation status or any updated alternatives. In contrast, for the 70 pre-cutoff APIs, the training data likely included both deprecated and up-to-date usages. This discrepancy causes DeepSeek-Coder to form a strongly entrenched understanding of the 6 post-cutoff APIs as valid, non-deprecated entities. Consequently, when updating the deprecated API knowledge for these 6 APIs, the newly injected information must counteract the model's rigid pre-existing understanding of "valid usage"—rendering this new knowledge less robust and ultimately resulting in slightly lower Portability.

6 Threats of Validity

External validity refers to the extent to which our findings can be generalized to other models, APIs, or scenarios. A key design feature of EDAPIBench addresses potential threats to this dimension: the benchmark employs a fully automated construction pipeline. Unlike manually curated benchmarks that require labor-intensive adaptation for new models, EDAPIBench can automatically generate LLM-specific editing instances for emerging LLMs by filtering candidate inputs based on whether the new model outputs deprecated APIs. This automation ensures the benchmark's adaptability and scalability to future LLMs.

Construct validity concerns whether our evaluation metrics and scope accurately capture the intended concept of "model editing performance for deprecated API knowledge." Our study focuses on single-line API completion—evaluating whether the edited LLM generates the correct, up-to-date API within a single line—rather than multi-line code completion or end-to-end functional correctness. This choice is motivated by two practical constraints: first, verifying the functional correctness of multi-line code requires task-specific test cases (e.g., runtime environment setup, input data preparation), which are difficult to standardize across eight diverse Python libraries and 145 APIs. Second, the prompts used for code completion often lack sufficient context for the LLM to infer the full logic of subsequent lines, making multi-line evaluation susceptible to bias from incomplete context. Therefore, following common practices in code completion studies [58, 60], we focus on this single-line evaluation.

Internal validity relates to factors that may confound the interpretation of our experimental results. A primary threat here is the inherent randomness of LLMs: even with fixed temperature settings and the use of a greedy decoding strategy, LLM outputs can vary across runs. This randomness manifests in two observable phenomena: pre-edited models (i.e., before any editing) occasionally achieve non-zero AEM scores on Effectiveness evaluations, indicating that they sometimes generate correct up-to-date APIs by chance; and Specificity AEM scores fail to reach 100%, reflecting unintended preservation or deviation in non-target API outputs—particularly notable in DeepSeek-Coder, which may be more susceptible to randomness [36]. We acknowledge that such randomness is an intrinsic characteristic of current generative models. While it cannot be completely eliminated, we mitigate its impact by running each model editing experiment five times and taking the median of the results, thereby reducing the influence of isolated random outputs on our conclusions. Furthermore, we do not perform hyperparameter tuning on baseline methods; instead, we adopt hyperparameter settings directly from existing literature. Preliminary experiments indicate that tuning these hyperparameters has minimal effect on their Effectiveness, Generalization, and Portability metrics, with AdaLoRA consistently outperforming other methods.

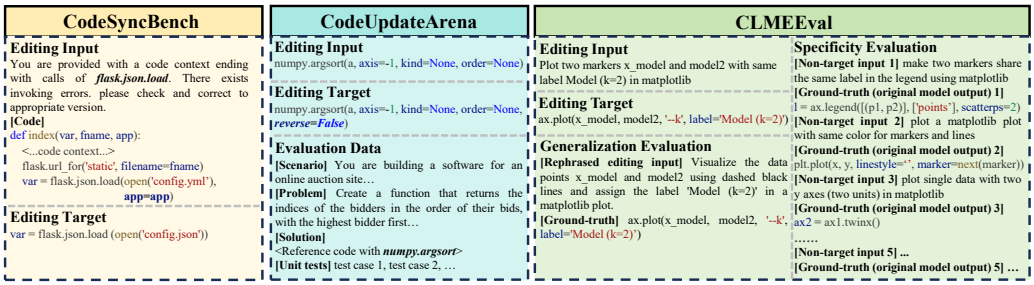


Fig. 13. The three benchmarks similar to our EDAPIBench.

7 Related Works

7.1 API Evolution and LLM API Knowledge Updating

Prior studies [34, 41, 42, 44] have examined the motivations behind API deprecation and how client developers respond to these changes. Common reasons for deprecating APIs include improving code readability, reducing redundancy, addressing poor coding practices, and fixing functional bugs. Deprecated APIs can impact hundreds of dependent projects [40], particularly as developers struggle to keep pace with fast-evolving software ecosystems [26, 54]. For instance, McDonnell et al. [30] found that only 22% of deprecated API usages are eventually migrated to replacement APIs. Similarly, Hora et al. [14] observed that while developers invest considerable effort in identifying and adopting alternatives, most projects fail to take timely action to address deprecations. Such delays lead to the silent accumulation of technical debt, complicating future migrations as multiple API changes must be resolved simultaneously [43]. Against this backdrop, deprecated APIs pose a critical problem for LLMs—a challenge amplified by their growing integration into software development workflows. Wang et al. [49] reported that 37.4% of GPT-3.5's API predictions involve deprecated APIs, underscoring the practical urgency of this issue. Motivated by this, our work focuses on updating deprecated API knowledge in LLMs through model editing techniques, addressing the gap between static pre-trained knowledge in LLMs and evolving APIs.

In the broader domain of LLM API knowledge updating, two closely related studies are Wang et al.'s CodeSyncBench [47] and Liu et al.'s CodeUpdateArena [28]. Both focus on evaluating LLMs' ability to adapt to **API parameter changes** (e.g., addition or removal of parameters) and generate code with correct parameter usage. CodeSyncBench focuses on API parameter correction tasks, such as fixing code like `var= flask.json.load(open('config.yml'), app=app)` to the correct form `var = flask.json.load(open('config.json'))` (as illustrated in Figure 13) to test whether LLMs can identify and fix parameter mismatches. CodeUpdateArena employs code generation tasks synthesized by LLMs (may not fully represent real-world development scenarios) that require proper use of updated API parameters—for example, instructing LLMs to "*Create a function that returns the indices of the bidders in the order of their bids, with the highest bidder first...*" which involves calling `numpy.argsort` with the newly added `reverse` parameter (also shown in Figure 13). In contrast, our work tackles the more critical problem of LLMs recommending **deprecated APIs**, which poses greater risks in software development and is not covered by these benchmarks.

Regarding methods for updating API parameter knowledge, Liu et al. [28] relied solely on LoRA, which yielded limited Effectiveness: the maximum improvement in Pass@1 reached only 6%. Wang et al. [47], meanwhile, employed SFT-LoRA [38] alongside three reinforcement learning (RL)-based approaches: DPO [39], SimPO [33], and ORPO [13]. While these RL methods also use LoRA for fine-tuning, they differ from standard LoRA in their fine-tuning signals: RL relies on reward-driven feedback (e.g., DPO's preference rankings, ORPO's pairwise comparisons) rather than direct supervised signals for targeted knowledge updates. Wang et al. [47] found that DPO

achieved the best performance, improving CodeBLEU scores from 32.68 (original model) to 44.95, though the absolute improvement remained modest. They also noted that RL methods required two to three times more training time than standard LoRA and involved additional reward engineering. These factors conflict with our focus on lightweight, efficient editing of deprecated API knowledge, where simplicity and speed are prioritized. Consequently, our study concentrates exclusively on LoRA-based techniques and does not incorporate RL-based approaches.

Another key distinction between these benchmarks and our `EDAPIBench` lies in the evaluation of Specificity. Both `CodeSyncBench` and `CodeUpdateArena` assess Specificity indirectly by tracking changes in Pass@1 scores on the HumanEval benchmark after fine-tuning. However, we argue that a more rigorous evaluation—aligned with `EDAPIBench`'s design—should directly verify whether updated LLMs can correctly generate parameters for other non-updated APIs within `CodeSyncBench` and `CodeUpdateArena`. This direct assessment ensures that the knowledge update process does not inadvertently degrade the model's existing knowledge of unaffected APIs, providing a more precise and targeted measure than indirect metrics such as HumanEval Pass@1.

7.2 Model Editing

Model editing is an active research direction in the field of NLP, with many techniques and benchmarks proposed to refine general-purpose LLMs. For example, $\text{WikiData}_{\text{recent}}$ [6], ZsRE [22], and KnowEdit [61] are widely used benchmarks for assessing model editing methods' performance in modifying factual knowledge, such as revising OpenAI's Chief Scientist from Ilya Sutskever to Jakub Pachocki. However, this design deviates from the actual paradigm of model editing: in practice, editing is only carried out when the model first outputs incorrect results—for instance, our `EDAPIBench` benchmark only triggers editing when LLMs actually generate deprecated APIs.

The application of model editing to software engineering tasks remains relatively limited. Gu et al. [9] proposed MENT, which repaired next-token errors in code generation by patching specific neurons in LLMs. Liu et al. [27] developed CREME to enhance LLMs' robustness against prompt perturbations in code generation (e.g., typos) via targeted parameter updates in robustness-sensitive layers. Nevertheless, these approaches focus on code generation tasks where there is no fixed correct output. This differs from our focus on factual knowledge updating, and thus these methods are not included in our evaluation. A more closely related work is Li et al.'s `CLMEEval` [24]—a model editing benchmark derived from `CoNaLa` [56] and `CodeSearchNet` [19]. However, the task setup in CLMEEval is somewhat detached from practical scenarios. For example (as illustrated in Figure 13), given a simple plotting description (e.g., "*Plot two markers x_model and model2 with same label Model (k=2) in matplotlib* ") as the editing input, the benchmark directly sets the target output to `ax.plot(x_model, model2, '--k', label='Model (k=2)')` **without first verifying whether the LLM already generates the correct code from the prompt prior to editing**. This diverges from a realistic editing paradigm, where edits are applied only when the model produces errors. Furthermore, CLMEEval relies on exact-match evaluation, which is limiting for code generation tasks since LLMs may generate functionally correct code that differs syntactically from the target. The benchmark also evaluates generalization solely through simple input paraphrases (e.g., textual rewrites of the original task), with limited diversity in paraphrase types. In contrast, our benchmark triggers editing only when LLMs output deprecated APIs. We introduce an "API Exact Match" metric to more precisely measure editing Effectiveness. To assess Generalization, we utilize GPT-4.1-generated code rewrites and incorporate a Portability dimension that evaluates whether the edited model can correctly complete the updated API calls across different editing instances, where the original model also completes the prompt with the same deprecated API.

8 Conclusion and Future Works

This study develops EDAPIBench, the first benchmark that focuses on editing deprecated API knowledge in LLMs and features full automation in its construction. It covers over 70 deprecated APIs from 8 popular Python libraries, with more than 1,000 editing instances per model. Unlike most existing model editing benchmarks in NLP and software engineerring, EDAPIBench only includes cases where LLMs initially generate deprecated APIs, ensuring meaningful "needs-editing" evaluations. We evaluate 10 state-of-the-art editing methods and find AdaLoRA performs best in enabling LLMs to generate up-to-date APIs but has low Specificity. Therefore, we propose AdaLoRA-L to improve Specificity effectively. Overall, EDAPIBench serves as a standardized, rigorous, and fully automated platform for future research on LLM API knowledge editing. It delivers cross-domain value by serving not only the software engineering community but also the broader NLP field for validating model editing methods. AdaLoRA-L offers a lightweight solution for updating deprecated API knowledge, enabling LLMs to generate more reliable, up-to-date code in practice. In future work, we plan to further expand EDAPIBench by adding more deprecated APIs.

9 Data Availability

Our EDAPIBench, the fully automated code for its construction, and model editing source code are available in [3].

References

[1] [n. d.]. *API Lifecycle Stages.* https://developers.meetmarigold.com/engage/terms/versioning-deprecation/#api-lifecycle-stages

[2] [n. d.]. *GPT-4.1.* https://openai.com/index/gpt-4-1/

[3] [n. d.]. *Our source code.* https://figshare.com/s/f18cf1ee0f8558b84d97

[4] [n. d.]. *Pytorch: A python package that provides tensor computation and deep neural networks.* https://pytorch.org/

[5] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-level encoding for neural source code summarization of subroutines. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 253–264.

[6] Roi Cohen, Eden Biran, Ori Yoran, Amir Globerson, and Mor Geva. 2024. Evaluating the ripple effects of knowledge editing in language models. *Transactions of the Association for Computational Linguistics* 12 (2024), 283–298.

[7] Junfeng Fang, Houcheng Jiang, Kun Wang, Yunshan Ma, Shi Jie, Xiang Wang, Xiangnan He, and Tat-Seng Chua. 2024. Alphaedit: Null-space constrained knowledge editing for language models. *arXiv preprint arXiv:2410.02355* (2024).

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[9] Jian Gu, Aldeida Aleti, Chunyang Chen, and Hongyu Zhang. 2024. Neuron Patching: Semantic-based Neuron-level Language Model Repair for Code Generation. arXiv:2312.05356 [cs.SE] https://arxiv.org/abs/2312.05356

[10] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[11] Tom Hartvigsen, Swami Sankaranarayanan, Hamid Palangi, Yoon Kim, and Marzyeh Ghassemi. 2023. Aging with grace: Lifelong model editing with discrete key-value adaptors. *Advances in Neural Information Processing Systems* 36 (2023), 47934–47959.

[12] Winston Haynes. 2013. *Benjamini–Hochberg Method.* Springer New York, New York, NY, 78–78.

[13] Jiwoo Hong, Noah Lee, and James Thorne. 2024. Orpo: Monolithic preference optimization without reference model. *arXiv preprint arXiv:2403.07691* (2024).

[14] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. 2015. How do developers react to api evolution? the pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 251–260.

[15] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

[16] Mingzhe Hu and Yu Zhang. 2023. An empirical study of the Python/C API on evolution and bug patterns. *Journal of Software: Evolution and Process* 35, 2 (2023), e2507.

[17] Baixiang Huang, Canyu Chen, Xiongxiao Xu, Ali Payani, and Kai Shu. 2024. Can Knowledge Editing Really Correct Hallucinations? *arXiv preprint arXiv:2410.16251* (2024).
[18] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).
[19] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).
[20] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language models for code completion: A practical evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[21] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. 2018. An empirical study on the impact of refactoring activities on evolving client-used apis. *Information and Software Technology* 93 (2018), 186–199.
[22] Omer Levy, Minjoon Seo, Eunsol Choi, and Luke Zettlemoyer. 2017. Zero-shot relation extraction via reading comprehension. *arXiv preprint arXiv:1706.04115* (2017).
[23] Xiaopeng Li, Shasha Li, Shezheng Song, Jing Yang, Jun Ma, and Jie Yu. 2024. Pmet: Precise model editing in a transformer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 18564–18572.
[24] Xiaopeng Li, Shangwen Wang, Shasha Li, Jun Ma, Jie Yu, Xiaodong Liu, Jing Wang, Bin Ji, and Weimin Zhang. 2025. Model Editing for LLMs4Code: How Far are we? (2025), 937–949.
[25] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.
[26] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. 477–487.
[27] Shuhan Liu, Xing Hu, Kerui Huang, Xiaohu Yang, David Lo, and Xin Xia. 2025. Improving Code LLM Robustness to Prompt Perturbations via Layer-Aware Model Editing. *arXiv preprint arXiv:2507.16407* (2025).
[28] Zeyu Leo Liu, Shrey Pandit, Xi Ye, Eunsol Choi, and Greg Durrett. 2024. Codeupdatearena: Benchmarking knowledge editing on api updates. *arXiv preprint arXiv:2407.06249* (2024).
[29] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE]
[30] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 70–79.
[31] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and editing factual associations in gpt. *Advances in neural information processing systems* 35 (2022), 17359–17372.
[32] Kevin Meng, Arnab Sen Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. 2022. Mass-editing memory in a transformer. *arXiv preprint arXiv:2210.07229* (2022).
[33] Yu Meng, Mengzhou Xia, and Danqi Chen. 2024. Simpo: Simple preference optimization with a reference-free reward. *Advances in Neural Information Processing Systems* 37 (2024), 124198–124235.
[34] Ariana Mirian, Nikunj Bhagat, Caitlin Sadowski, Adrienne Porter Felt, Stefan Savage, and Geoffrey M Voelker. 2019. Web feature deprecation: a case study for chrome. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 302–311.
[35] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. 2019. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 11264–11272.
[36] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. *arXiv e-prints* (2023), arXiv–2308.
[37] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
[38] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with gpt-4. *arXiv preprint arXiv:2304.03277* (2023).
[39] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems* 36 (2023), 53728–53741.
[40] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
[41] Anand Ashok Sawant, Mauricio Aniche, Arie van Deursen, and Alberto Bacchelli. 2018. Understanding developers' needs on deprecation as a language feature. In *Proceedings of the 40th international conference on software engineering*. 561–571.

[42] Anand Ashok Sawant, Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski, and Alberto Bacchelli. 2018. Why are features deprecated? an investigation into the motivation behind deprecation. In *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 13–24.

[43] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2016. On the reaction to deprecation of 25,357 clients of 4+ 1 popular Java APIs. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 400–410.

[44] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2019. To react, or not to react: Patterns of reaction to API deprecation. *Empirical Software Engineering* 24, 6 (2019), 3824–3870.

[45] Chenmien Tan, Ge Zhang, and Jie Fu. 2024. Massive Editing for Large Language Models via Meta Learning. arXiv:2311.04661 [cs.CL] https://arxiv.org/abs/2311.04661

[46] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 397–407.

[47] Chenlong Wang, Zhaoyang Chu, Zhengxiang Cheng, Xuyi Yang, Kaiyue Qiu, Yao Wan, Zhou Zhao, Xuanhua Shi, and Dongping Chen. 2025. CODESYNC: Synchronizing Large Language Models with Dynamic Code Evolution at Scale. *arXiv preprint arXiv:2502.16645* (2025).

[48] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. How practitioners expect code completion?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1294–1306.

[49] Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. 2024. Llms meet library evolution: Evaluating deprecated api usage in llm-based code completion. *arXiv preprint arXiv:2406.09834* (2024).

[50] Chaozheng Wang, Zezhou Yang, Shuzheng Gao, Cuiyun Gao, Ting Peng, Hailiang Huang, Yuetang Deng, and Michael Lyu. 2025. RAG or Fine-tuning? A Comparative Study on LCMs-based Code Completion in Industry. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 93–104.

[51] Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. 2020. Exploring how deprecated python library apis are (not) handled. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 233–244.

[52] Peng Wang, Ningyu Zhang, Bozhong Tian, Zekun Xi, Yunzhi Yao, Ziwen Xu, Mengru Wang, Shengyu Mao, Xiaohan Wang, Siyuan Cheng, et al. 2023. Easyedit: An easy-to-use knowledge editing framework for large language models. *arXiv preprint arXiv:2308.07269* (2023).

[53] Song Wang, Yaochen Zhu, Haochen Liu, Zaiyi Zheng, Chen Chen, and Jundong Li. 2024. Knowledge editing for large language models: A survey. *Comput. Surveys* 57, 3 (2024), 1–37.

[54] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–45.

[55] Robert F Woolson. 2005. Wilcoxon signed-rank test. *Encyclopedia of biostatistics* 8 (2005).

[56] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*. 476–486.

[57] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data augmentation by program transformation. *Journal of Systems and Software* 190 (2022), 111304.

[58] Xiao Yu, Lei Liu, Xing Hu, Jacky Wai Keung, Jin Liu, and Xin Xia. 2024. Fight Fire with Fire: How Much Can We Trust ChatGPT on Source Code-Related Tasks? *IEEE Transactions on Software Engineering* 50, 12 (2024), 3435–3453.

[59] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. 2021. Research on third-party libraries in android apps: A taxonomy and systematic literature review. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4181–4213.

[60] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 2471–2484.

[61] Ningyu Zhang, Yunzhi Yao, Bozhong Tian, Peng Wang, Shumin Deng, Mengru Wang, Zekun Xi, Shengyu Mao, Jintian Zhang, Yuansheng Ni, et al. 2024. A Comprehensive Study of Knowledge Editing for Large Language Models. *arXiv preprint arXiv:2401.01286* (2024).

[62] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. 2023. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.10512* (2023).

[63] Qingru Zhang, Simiao Zuo, Chen Liang, Alexander Bukharin, Pengcheng He, Weizhu Chen, and Tuo Zhao. 2022. Platon: Pruning large transformer models with upper confidence bound of weight importance. In *International conference on*

machine learning. PMLR, 26809–26823.

[64] Zexuan Zhong, Zhengxuan Wu, Christopher D Manning, Christopher Potts, and Danqi Chen. 2023. Mquake: Assessing knowledge editing in language models via multi-hop questions. *arXiv preprint arXiv:2305.14795* (2023).

[65] Chen Zhu, Ankit Singh Rawat, Manzil Zaheer, Srinadh Bhojanapalli, Daliang Li, Felix Yu, and Sanjiv Kumar. 2020. Modifying memories in transformer models. *arXiv preprint arXiv:2012.00363* (2020).