Understanding Robustness of Model Editing in Code LLMs: An Empirical Study

VINAIK CHHETRI, Louisiana State University, USA
A.B SIDDIQUE, University of Kentucky, USA
UMAR FAROOQ, Louisiana State University, USA

Large language models (LLMs) are increasingly used in software development to assist with tasks such as code generation, refactoring, and bug fixing. However, while LLMs remain static after pretraining, programming languages and APIs continue to evolve, leading to the generation of deprecated or incompatible code that undermines reliability. Retraining LLMs from scratch to reflect such changes is computationally expensive, making model editing a promising lightweight alternative that updates only a small subset of parameters. Despite its potential, it remains unclear whether model editing yields genuine syntactic and semantic adaptations or merely superficial fixes. In this work, we present a systematic study of five state-of-the-art model editing methods: Constrained Fine-Tuning (FT), GRACE, MEMIT, PMET, and ROME. We apply these methods to three leading open-source code LLMs, CodeLlama, CodeQwen1.5, and DeepSeek-Coder, under controlled API deprecation scenarios. Our evaluation covers both instant and sequential editing settings, using three disjoint evaluation sets designed to assess reliability, generalization, and specificity. We measure model correctness at three levels: successful compilation, partial test case pass, and full test pass. Our findings show that instant edits consistently degrade model performance, with syntactic validity dropping by up to 86 percentage points and functional correctness declining by 45 points even in the best-performing setting. Sequential edits further amplify this degradation, and in some cases, model performance collapses entirely. Across all models, most passing generations relied on workarounds rather than correctly adopting the intended changes, while faulty adoptions that result in test failures or compilation errors were significantly more frequent. Correct adoptions, where the model correctly integrates the intended change, occurred in only about 6% of cases. We observe recurring failure modes, including syntax errors, compilation failures, unstable behavior, and regressions on unrelated tasks. These results indicate that current editing methods offer only limited and unstable adaptation in code LLMs. Our study underscores the need for editing techniques specifically designed for code LLMs to ensure correctness, robustness, and maintainability as software ecosystems evolve.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Machine learning*; *Natural language processing*.

ACM Reference Format:
Vinaik Chhetri, A.B Siddique, and Umar Farooq. 2025. Understanding Robustness of Model Editing in Code LLMs: An Empirical Study. *Proc. ACM Softw. Eng.* 1, 1 (November 2025), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large language models (LLMs) for code have become indispensable in modern software development, supporting tasks such as code completion, bug fixing, and migration across languages and platforms [1, 15]. However, these models face a persistent challenge: software ecosystems evolve rapidly, and each release introduces significant changes. In particular, APIs are frequently deprecated, renamed, or removed. For example, Android deprecated more than *800* symbols between API levels 34 and 35 [28], pandas deprecated about *200* APIs between versions 1.0 and 1.5 [35], and even core languages and runtimes such as Java [8], NumPy [9], and Node.js [7] introduced dozens

Authors' Contact Information: Vinaik Chhetri, Louisiana State University, Baton Rouge, LA, USA, vchhet2@lsu.edu; A.B Siddique, University of Kentucky, Lexington, KY, USA, ab.sidique@uky.edu; Umar Farooq, Louisiana State University, Baton Rouge, LA, USA, ufarooq@lsu.edu.

2025. ACM 2994-970X/2025/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1. API deprecations across major ecosystems. Counts reflect the number of deprecated or removed APIs between the listed releases.

Ecosystem	Description	Release Span	# of Deprecations
Android [29]	Mobile app framework	34 → 35 (API level)	834
Java (Oracle JDK) [34]	Core standard library	11 → 17	88
NumPy [10]	Scientific computing library	1.9 → 2.0	104
Node.js [13]	JavaScript runtime	23.0 → 24.0	196
pandas [38]	Data analysis library	1.0 → 1.5	200

to hundreds of deprecations. Table 1 reports the number of deprecated APIs across different release spans in popular ecosystems. Since code LLMs are trained on fixed corpora, they cannot adapt to these ongoing changes. As a result, code LLMs often generate outdated or misleading suggestions, which reduces their reliability.

For human developers, such changes are disruptive but manageable. Release notes and migration guides describe new and deprecated APIs, and over time, developers adopt the updated APIs. For example, after the deprecation of `DataFrame.append()` [37], developers transitioned to `pandas.concat()` [36] using documentation as guidance. Code LLMs, in contrast, are frozen at the time of training. Unless they are retrained on new corpora, they continue to produce outdated calls. A full retraining cycle is prohibitively expensive and slow, especially when APIs change frequently across multiple libraries. It also requires substantial manual effort to remove training examples containing outdated API calls and to manually generate coding examples for newly released APIs. To address this situation, recent research has proposed *model editing*, where lightweight interventions modify a model's weights to revise a specific fact or function without retraining from scratch.

Model Editing. In natural language settings, model editing is typically formulated as a targeted modification to the model's internal knowledge. The goal is to revise specific facts without retraining the model from scratch and without disrupting its performance on unrelated tasks. For example, consider the fact that "the president of country X is Y." A successful edit revises the model's internal representation so that, when asked "Who is the president of country X?", it now answers "Z" instead of "Y." Importantly, this change must be localized: the model should still respond correctly to other queries about the entities "X", "Y", and "Z", and should not exhibit unintended shifts in behavior outside the scope of the edited fact.

Compared to this formulation, editing in code LLMs is considerably more demanding. In the coding domain, API updates must preserve *syntactic validity* so that generated code compiles, achieve *functional correctness* so that tests pass, and ensure *semantic alignment* with the intended new functionality. Additionally, programming introduces unique failure modes. A model may generate code that compiles but is semantically incorrect, or it may sidestep the intended update by producing an alternative implementation that passes tests without using the new API. Such workarounds create the illusion of successful model editing while undermining the goal of aligning the model with the evolving ecosystem. Moreover, edits must generalize beyond the training context, avoid regressions on unrelated tasks, and remain stable when applied sequentially across multiple API changes.

Yet, despite these stricter requirements, there has been little systematic study of editing in code LLMs. Most prior work has focused on factual updates in natural language, leaving *open questions* about how editing methods behave when confronted with evolving APIs, compilation constraints, and program semantics.

Systematic evaluation of editing methods across multiple code LLMs is difficult to conduct fairly. Each model is trained on different corpora with different cutoff dates, and real API changes rarely align with these timelines. As a result, the timing of real-world updates can easily confound

Table 2. Study setting summarizing models, regimes, methods, and evaluation scope.

Dimension	Description	Captures
Models	CodeLlama-7B, CodeQwen1.5-7B, DeepSeek-Coder-6.7B	Cross-model robustness
Methods	FT, GRACE, MEMIT, PMET, ROME	Cross- method robustness
Regimes	*Instant:* one API edit; *Sequential:* multiple API edits	Local vs. cumulative effects
Subsets	Reliability, Generalization, Specificity	Accuracy, transfer, regressions
Metrics	Compile, Partial-Pass, Full-Pass; Pass@k (k=1, 3, 5)	Syntactic, semantic preservation
Adoption	Correct Adoption, Faulty Adoption, Workaround	Genuine vs. superficial adoption

evaluation. One model may have already seen the new API during pretraining or instruction tuning, while another may not have. In such cases, an apparent success may reflect memorized recall rather than true editing. To this end, synthetic changes in APIs provide the control that real data cannot. That is, we can eliminate the risk of training data leakage and ensure that every model faces the same update under identical conditions by introducing synthetic edits that we know no model has seen. This evaluation design makes it possible to attribute outcomes directly to the editing method and the model's capabilities, rather than to incidental differences in the training data.

In addition, generating synthetic APIs requires making those APIs available to the compiler so that updated calls are recognized as valid, while simultaneously ensuring that deprecated versions are removed from the compiler environment. This step enforces a realistic setting in which only the intended updates can succeed. Paired with execution-based unit tests, this setup makes it possible to verify whether a model genuinely adopts the new API or merely produces a workaround that happens to pass. This compiler integration and testing requirement is unique to code editing. In natural language, synthetic edits can be validated at the surface level (e.g., phrase substitution), but in code, an edit must survive compilation and runtime checks to be considered correct. *Controlled edits, compiler enforcement, and test suites together provide a principled foundation for evaluating the robustness of model editing in code LLMs.*

State-of-the-Art. The most comprehensive study on model editing for code LLMs to date is CLMEEval [26], which evaluates six editing techniques across multiple code generation and summarization tasks. However, CLMEEval relies on existing datasets such as CoNaLa (released in 2018) [46] and CodeSearchNet (released in 2019) [22], making it susceptible to pretraining data leakage and limiting control over what models have previously seen. Moreover, its evaluation focuses on textual correctness and fluency without compiler enforcement or execution-level validation, so edits may appear successful while remaining semantically incorrect. In contrast, our work introduces a compiler-integrated benchmark based on synthetic API deprecations, eliminating data leakage and enforcing runtime validity. This design enables precise attribution of editing effects to the methods themselves and directly measures whether edits result in genuine adoption rather than superficial change.

Overview of this Work. This work presents a systematic investigation into model editing for code language models (code LLMs), focusing on how state-of-the-art techniques perform when adapting to evolving software environments. Our study design, summarized in Table 2, spans six key dimensions: models, methods, editing regimes, evaluation subsets, correctness metrics, and API adoption. This rigorous and systematic analysis allows us to examine how edits affect syntactic validity, semantic correctness, and behavioral stability, providing a detailed view of the current state of model editing for code and highlighting key challenges and directions for future research.

We evaluate five representative editing methods: Constrained Fine-Tuning (FT) [47], GRACE [20], MEMIT [31], PMET [25], and ROME [30], across three leading open-source code LLMs: CodeLlama-7B [40], CodeQwen1.5-7B [2], and DeepSeek-Coder-6.7B [17].

We examine two editing regimes: instant editing and sequential editing. In instant editing, a single API change is introduced and evaluated in isolation. In the sequential editing setting, a sequence of edits is applied across multiple APIs, each introducing a new change. This setup mimics cumulative adaptation over time and allows us to evaluate whether models can integrate successive updates without degrading performance.

To simulate realistic software evolution, we construct a benchmark based on synthetic API deprecations in a Python sandbox environment. This benchmark builds on two widely used code generation datasets, HumanEval [6] and MBPP [39], with modifications that require replacing deprecated API calls with their updated counterparts while preserving program behavior. This controlled setting enables consistent and repeatable evaluation of model adaptation to changes.

We organize the benchmark into three disjoint subsets. The reliability subset evaluates model performance on edited APIs on the seen tasks to assess immediate correctness. The generalization subset tests the same APIs in unseen contexts to examine transfer beyond the training context. The specificity subset evaluates unrelated APIs to detect regressions or unintended side effects. Each subset is measured at three levels of correctness: whether the generated code compiles (valid code without error), partially passes tests (at least one test passes), or fully passes all tests. Evaluations are conducted at multiple sampling levels, $k \in \{1, 3, 5\}$, using the Pass@k metric, which is standard for code generation. This evaluation design enables fine-grained analysis of syntax, partial reasoning, and functional correctness.

Key Findings. We structure our analysis around four research questions:

First, *we examine how robust models are to a single, isolated edit.* A single edit can disrupt model behavior, reducing reliability, generalization, and specificity. Among the editing methods studied, GRACE exhibits the most controlled degradation in specificity, preserving pre-edit accuracy and localizing changes effectively, though it still suffers up to a 45 percentage point drop in generalization. Other methods are more fragile, with some edits causing performance to fall by as much as 86 points. While sampling improves the chance of success by 8–10 percentage points, it does not address the core issue of robustness. No existing method provides truly stable or isolated edits.

Second, *we investigate the effects of cumulative edits across APIs.* Sequential editing leads to rapid degradation across all models, with correctness dropping by 60 to 80 percent after just a few sequential updates. Most failures occur within the first 10 to 20 edits. GRACE maintains moderate performance over longer sequences of updates, whereas other methods degrade more sharply. Among the models, CodeQwen1.5-7B is the most resilient, CodeLlama-7B retains limited functionality, and DeepSeek-Coder-6.7B fails early. Sampling provides no meaningful benefit in this setting, as accumulated edits compound instability.

Third, we ask *whether edits lead to genuine semantic adaptation or only superficial compliance.* GRACE is the only method that substantially contributes to correct adoption, though its effectiveness declines under sequential edits. FT shows modest gains with sequential updates, but also introduces a high rate of faulty adoptions. Other methods fail to enable meaningful adoption. Our findings suggest that current editing methods promote surface-level changes without achieving the semantic alignment required for reliable API adoption.

Finally, we analyze *where edits fail during generation.* Most editing methods (ROME, MEMIT, and PMET) often fail to apply the target API or generate code with structural errors, such as unresolved imports and syntax issues. FT and GRACE preserve compilation more reliably but frequently produce outputs that pass only some test cases or vary across samples. Even successful generations often rely on legacy logic instead of adopting the intended update, and nearly 30% introduce regressions in unrelated behaviors.

Overall, our analyses reveal that current editing methods offer only limited and unstable adaptation in code LLMs. Syntactic updates are inconsistent, and semantic correctness declines as edits

accumulate. Across all models and methods, most edits are superficial, enhancing compilability without preserving intended functionality. These findings highlight the need for editing techniques that combine parameter updates with program-level reasoning to maintain behavioral stability in evolving software environments.

2 Background

2.1 Transformer-based Language Models

The transformer architecture[42] has become the dominant model for natural-language and code understanding and generation. Transformers employ an attention-based mechanism that allows each token to capture relationships with all other tokens in the sequence simultaneously. Specifically, the standard transformer architecture consists of stacked layers of encoders and decoders. Each layer contains a multi-head self-attention mechanism that computes relationships between all token pairs, followed by a position-wise feed-forward layer. Multi-head attention operates by running multiple attention mechanisms in parallel across different representational subspaces, allowing the model to simultaneously capture diverse linguistic patterns such as syntactic dependencies and semantic relationships. Since attention operations are inherently permutation-invariant, positional encodings are added to input embeddings to inject sequence order information. Throughout the architecture, residual connections and layer normalization facilitate gradient flow and stabilize training in deep networks.

Modern variants adapt this architecture for specific tasks: encoder-only models like BERT [11] excel at understanding tasks such as classification and named entity recognition, while decoder-only models like GPT3 [3] are optimized for autoregressive text generation. The decoder employs masked self-attention to prevent tokens from attending to future positions during training, ensuring proper autoregressive behavior. This architectural flexibility, combined with the ability to efficiently model long-range dependencies through parallelizable attention operations, has established transformer-based models as the foundation for state-of-the-art language models across diverse applications.

2.2 Code Language Models

Code language models are transformer-based models pre-trained on large corpora of source code to learn programming knowledge [5, 12, 23, 41, 45]. Models such as Codex [5], CodeBERT [12], CodeT5 [45], StarCoder [23], and Code Llama [41] are trained on datasets sourced from repositories like GitHub, learning to represent syntax rules, API signatures, and programming patterns through various pretraining objectives including masked language modeling, causal language modeling, and code-documentation alignment. Training these models requires massive computational resources – Code Llama required 1.4 million GPU hours [41] and StarCoderBase required over 320,000 GPU hours [23]. As software evolves, code knowledge in language models quickly becomes outdated due to API changes and framework updates. Retraining models from scratch to incorporate these changes is prohibitively expensive, which motivates the need for efficient model editing techniques that can update targeted knowledge without full retraining.

2.3 Editing Methods

To enable efficient knowledge updates without retraining, various model editing techniques have been developed. These methods differ in how they modify the model: global optimization approaches use gradient-based updates to selected parameters; local modification methods surgically edit specific weights; and external memorization techniques add memory modules without changing parameters. In the following, we provide an overview of representative methods from each approach and their characteristics for knowledge editing tasks.

Constrained Fine-Tuning (FT) [47]. Constrained Fine-Tuning is a global optimization approach that modifies a small portion of the model's parameters while keeping the remaining weights frozen, a strategy motivated by the typically small size of editing datasets. The objective is to minimize the loss over the target editing samples while simultaneously constraining the model to preserve its performance on unrelated data, thereby preventing catastrophic forgetting and maintaining the model's general knowledge.

GRACE [20]. GRACE represents an external memorization strategy that fundamentally differs from parameter modification methods by avoiding direct updates to the model's weights. Instead, it augments the model's layer chosen for editing with an external memory module (codebook) that stores and retrieves edited knowledge, thereby preserving the original model parameters entirely. While this approach guarantees that the base model's knowledge remains intact, it introduces additional computational overhead during inference due to the memory lookup operations. When GRACE performs an edit, it either creates a new key-value pair or updates an existing one in the codebook. The key is derived from the model's hidden state representation at a specific layer when processing the input that needs editing, serving as a unique identifier for the edited knowledge. The value is a learned vector that encodes the corrected information and is trained through backpropagation on the editing loss for N gradient descent steps to ensure it correctly modifies the model's behavior. During inference, when the model encounters an input that matches a stored key, the corresponding value replaces the original hidden state for the remainder of the forward pass, effectively overriding the model's original output for that specific input.

ROME [30] Rank-One Model Editing (ROME) is a local modification approach that surgically targets specific parameters within the model rather than applying global updates across the entire network. The method leverages causal tracing technique [43] to localize a critical layer where factual knowledge is encoded, operating on the principle that the Feed-Forward Network (FFN) at this layer implements an associative key-value memory mechanism for storing and retrieving factual information [14]. ROME casts knowledge editing as a constrained least-squares optimization problem where the objective is to update the FFN weight matrix to produce the correct output for the edited fact. This direct parameter modification allows ROME to integrate new knowledge without requiring iterative training or gradient descent, making it computationally efficient for individual edits. However, ROME's design imposes a significant constraint: it processes only one edit at a time. Its inability to perform editing sequentially limits ROME's applicability to scenarios requiring batch edits or continuous knowledge maintenance, though it remains highly effective for precise, single-fact corrections where surgical precision and computational efficiency are paramount.

MEMIT [31]. MEMIT extends ROME's rank-one modification approach to handle batch edits by targeting multiple layers simultaneously rather than a single layer. Where ROME modifies the MLP weights of one critical layer to insert a single memory, MEMIT identifies and modifies a range of mediating layers that collectively store factual knowledge about a given subject. Using causal tracing, MEMIT determines which layers are critical for memory recall and computes a cumulative update that is distributed across all these mediating MLP layers. This distributed update is designed such that by the final mediating layer, the model's output captures all the new memories being inserted. This multi-layer intervention strategy enables MEMIT to scale to significantly larger edit batches, however MEMIT is not always successful when performing sequential editing [18].

PMET. [25]. PMET addresses a critical limitation in existing model editing methods like ROME and MEMIT, which assume that Transformer Layer (TL) hidden states can serve as direct representations of target knowledge for updating FFN weights. However, TL hidden states aggregate information from three sources: Multi-Head Self-Attention (MHSA), FFN, and residual connections. This aggregation introduces irrelevant or noisy information that is not specifically required by the FFN, leading to imprecise weight updates and degraded editing performance. Through analysis of

MHSA and FFN hidden states, PMET discovers that MHSA encodes general knowledge extraction patterns rather than specific factual knowledge, suggesting that MHSA weights do not require updating when introducing new facts. Building on this insight, [25] optimizes the hidden states of individual Transformer Components (MHSA and FFN) separately, but uses only the optimized FFN hidden states as the target knowledge representations for updating FFN weights. This approach eliminates the contamination from MHSA information in the weight update process, enabling more precise modifications.

3 Study Design and Methodology

3.1 Research Questions

Editing code LLMs raises challenges that go beyond factual updates in natural language. An edit must yield syntactically valid code, compile under the updated API, preserve semantic correctness, and avoid regressions on unrelated tasks. To capture these different dimensions, we structure our study around four research questions:

> *RQ1. How robust are edits across reliability, generalization, and specificity?*

Editing methods differ in scope. Gradient-based approaches like fine-tuning may overfit to training-like contexts (reliability), while localized methods such as ROME or MEMIT may affect unrelated tasks (specificity). True robustness requires edits to generalize to novel contexts without degrading unrelated functionality.

> *RQ2. What happens when edits accumulate sequentially?*

Software systems evolve continuously, not one change at a time. In practice, models must absorb a stream of API updates. However, sequential edits can amplify instability or erase prior updates. We measure performance trajectories as edits accumulate, highlighting whether editing methods can sustain accuracy over multiple updates, or whether saturation and interference emerge.

> *RQ3. Do edited models truly adapt to new APIs, or do they fall back on workarounds?*

Passing tests is not sufficient if a model sidesteps the intended edit. Edited models may continue to generate obsolete APIs, re-implement deprecated behavior manually, or exploit unintended shortcuts. We distinguish genuine adoptions – where the updated API is adopted – from workaround successes that only simulate correctness.

> *RQ4. Where in the pipeline do edits fail?*

Even when editing methods fail, they fail differently. Some edits break at the compilation stage, others compile but produce semantically incorrect outputs, and others regress on unrelated tasks. By decomposing outcomes across our correctness tiers and subsets, we characterize failure modes and identify the stages at which different methods fall short.

3.2 Editing Methods

We evaluate editing methods that exemplify the three dominant paradigms in model editing: global optimization, local modification, and external memory. Table 3 summarizes each method and its underlying mechanism. The following describes the experimental setup used for each method.

Table 3. Editing techniques evaluated in this study.

Category	Method(s)	Scope and Mechanism
Global Optimization	FT (Fine-Tuning)	Lightweight gradient updates applied on task-specific examples; risk of drift across unrelated tasks.
Local Modification	ROME	Rank-one closed-form update at a single key layer; efficient but limited scope.
	MEMIT, PMET	Multi-layer interventions; MEMIT extends ROME for batch edits, PMET focuses on FFN layers for stability.
External-Memorization	GRACE	Adapter-based storage of edits without altering base weights; scalable to many edits, instance or sequential.

Constrained Fine-Tuning (FT) [47]. We adapt FT for multi-token sequence generation by fine-tuning the down-projection matrix of the 21st transformer layer for 25 iterations, using a learning rate of 5×10^{-4} and an L_∞-norm constraint of $\delta = 1 \times 10^{-4}$, similar to prior work [26].
GRACE [20]. We evaluate GRACE under three budgets: 30, 90, and 270 edits. Unlike other methods, GRACE avoids parameter updates entirely by storing edits in an external key-value memory and interpolating them at inference time. Similar to existing work [26], we insert an adapter into the down-projection matrix of the 27th transformer layer, using a key deferral radius of 1. The value vectors are optimized via fine-tuning with a learning rate of 1 for up to 30 iterations.
ROME [30]. For ROME, we do not perform causal intervention to identify critical layers. Prior work [21] shows that causal tracing yields layer localizations that are statistically uncorrelated with editing success. Instead, we adopt the layer selection established in prior studies [24, 26, 44], and apply ROME edits to the 5th transformer layer.
MEMIT [31]. For MEMIT, we follow prior work [21, 24, 26, 44] and apply edits to transformer layers 4, 5, 6, 7, 8.
PMET. [25]. For PMET, we adopt the same layer selection as in prior work [21, 24, 26, 44], editing transformer layers 4, 5, 6, 7, 8.

3.2.1 Editing Regimes. Editing methods cannot be judged in isolation; how they are applied across time is equally important. In practice, software ecosystems evolve incrementally, with new API changes arriving continuously. An editing approach that works well for a single update may fail once multiple changes accumulate. To capture this spectrum, we evaluate all methods under two complementary regimes:

- *Instant editing.* A single API update is applied in isolation. This regime probes whether an editing method can integrate one change correctly and stably without disturbing unrelated behaviors.
- *Sequential editing.* Multiple API updates are applied cumulatively, reflecting realistic scenarios where software ecosystems evolve continuously.

This regime exposes whether methods exhibit early gains, saturation, or interference as edits accumulate. Together, these regimes allow us to capture both precision at the single-edit level and robustness in long-horizon evolution.

3.3 Dataset and Experimental Environment

Evaluating editing methods fairly requires a dataset where updates are both controlled and enforceable at execution time. To this end, we construct our dataset by introducing synthetic API replacements on top of MBPP [39] and HumanEval [6], two widely used Python coding benchmarks. These benchmarks provide diverse programming tasks and include test cases to evaluate the functional correctness and semantic behavior of code generation.

Table 4. API usage statistics across MBPP, HumanEval, and their combination to use in our benchmark.

Metric	MBPP	HumanEval	Combined
Original Dataset Overview			
Problems count	974	164	1,138
Problem–API pairs detected	1,156	259	1,415
Dataset with Synthetic APIs			
Number of problems	675	122	**797**
Problem–API pairs	831	162	993
Unique synthetic APIs	62	26	**65**
Problems in Subsets			
Reliability	212	47	259
Generalization	277	51	328
Specificity	186	24	210

To construct the benchmark, we first filtered problems where both (1) the canonical solutions from the original datasets and (2) the pre-edit code LLM generations invoked at least one API. We then removed problems that could be solved without any API calls. This filtering process yielded 797 problems spanning 65 unique APIs. Table 4 summarizes the dataset statistics. Each synthetic update replaces an existing function with a new alternative that is semantically equivalent but distinct in name and usage. For example, we replaced the standard `math.sqrt()` function with a custom `math.square_root()` function that retains the original functionality to ensure consistent behavior while test case execution. A complete list of all API replacements is included in our replication package. This design stresses the model's ability to adapt to new APIs rather than rely on memorized training data. To probe robustness, we partition the dataset into three subsets:

- *Reliability.* This subset contains 259 problems, accounting for 33% of the dataset. These are tasks on which we perform editing and measure whether edits remain stable on these problems.
- *Generalization.* This subset contains 328 problems, which make up 41% of the dataset. These tasks test model performance in novel contexts where the updated API appears in previously unseen ways.
- *Specificity.* This subset contains 210 problems, representing 26% of the dataset. These tasks are unrelated to the targeted update and are used to detect regressions or unintended side effects on unrelated functionality.

Experimental Environment. To operationalize synthetic API updates, we implement a compiler-aware sandbox environment that enforces correctness at every evaluation stage. The sandbox modifies the available API surface by registering synthetic replacements while removing the deprecated counterparts. As a result, code that still relies on the old API immediately fails to compile, faithfully simulating the breaking behavior of real deprecations.

The environment integrates directly with the Python runtime and package system, exposing synthetic APIs as importable modules. From the model's perspective, these functions are indistinguishable from real library calls, allowing generated programs to link and execute without special instrumentation. This integration ensures that evaluation reflects natural developer workflows rather than artificial string substitutions.

On top of compilation checks, the sandbox executes each program against a suite of unit tests. Outcomes are categorized into three tiers:

- *Compilation Success (Compiles).* The generated code does not have any syntax or type errors.
- *At-least-one test passing (Partial-Pass).* The compiled code passes at least one of the provided unit tests, indicating partial functional progress.

Table 5. Baseline (non-edit) performance across model families used in our study. Percentages are shown relative to 797 tasks, with the pass@5 (at least one generation out of 5 passing) metric.

Model	Compiles	Partial-Pass	Full-Pass
CodeLlama-7B	98.1%	64.1%	44.0%
CodeQwen1.5-7B	99.4%	78.3%	68.6%
Deepseek-coder-6.7B	88.8%	53.3%	44.2%

- *All tests passing (Full-Pass).* The compiled code passes all unit tests, demonstrating semantic alignment with the intended API update.

To capture model performance under different sampling conditions, we compute Pass@k metrics at $k = 1, 3, 5$ for all three correctness levels (Compiles, Partial-Pass, and Full-Pass). Pass@1 measures success when only a single generated output is considered. Pass@3 and Pass@5 measure the probability of at least one success when three and five samples are independently sampled, respectively. This setup allows us to distinguish between consistent correctness (high Pass@1) and cases where additional sampling increases the likelihood of producing compilable or correct solutions (improvements at Pass@3 or Pass@5).

By embedding synthetic APIs directly into the compiler and runtime, we eliminate reliance on memorized training data and ensure that evaluation outcomes are attributable to the editing methods themselves.

3.4 Subject Models

We evaluate editing methods across three open-source code LLMs that represent diverse training strategies and usage settings. These models serve as the substrate for all editing experiments.

- *CodeLlama-7B-Instruct* [40]. A widely used instruction-tuned variant of the LLaMA family trained on code. It is a common baseline for Python program synthesis tasks.
- *CodeQwen1.5-7B-Chat* [2]. A chat-optimized model that combines general-purpose pretraining with code-oriented instruction tuning.
- *DeepSeek-Coder-6.7B-Instruct* [17]. A code-focused model trained on large-scale software repositories with instruction tuning.

Before applying editing methods, we first establish the baseline (non-edit) performance of these models on our synthetic API dataset. This allows us to contextualize editing results relative to their out-of-the-box ability to handle unseen APIs.

The baseline results reveal substantial variation across models as presented in Table 5. CodeQwen1.5 achieves the strongest performance, compiling almost all problems and solving over half of them completely. CodeLlama performs moderately, while DeepSeek struggles with compilation stability, leading to fewer opportunities for functional correctness.

4 Results

4.1 RQ1: Robustness of Editing

We evaluate success on three subsets: the Reliability, Generalization, and Specificity sets. We also measure behavioral stability and the impact of more samples.

4.1.1 RQ 1.1: Model Robustness Under Reliability Tasks. Table 6 reports the reliability performance across successive editing methods. Reliability measures a model's ability to preserve previously correct behaviors after edits, evaluated through compilation success, partial test passes, and full correctness on all tests.

Table 6. Reliability subset: Pass@k rates for *Compiles*, *Partial-Pass* (≥ 1 test passes), and *Full-Pass* (all tests pass) at $k \in \{1, 5\}$ across models and editing methods under instant editing.

Method	CodeLlama-7B						CodeQwen1.5-7B						DeepSeek-Coder-6.7B					
	Compiles		Partial-Pass		Full-Pass		Compiles		Partial-Pass		Full-Pass		Compiles		Partial-Pass		Full-Pass	
	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5
Baseline	92.74	98.07	54.52	65.64	33.75	40.54	99.23	100.00	73.44	79.54	60.39	67.95	72.51	87.26	44.32	61.00	35.21	49.03
FT	39.37	56.45	17.70	27.53	9.97	16.03	38.75	50.17	27.94	37.63	18.75	26.48	38.33	59.93	16.66	26.83	11.71	18.82
GRACE-30	45.51	62.37	16.93	28.22	10.31	16.72	41.11	53.31	27.94	37.98	19.51	27.18	15.47	24.39	0.98	4.18	0.77	3.14
GRACE-90	90.24	92.33	63.62	65.85	62.51	64.46	41.11	53.31	27.94	37.98	19.51	27.18	30.80	44.95	3.14	4.53	2.44	3.48
GRACE-270	89.90	93.38	64.39	66.90	62.86	65.16	41.11	53.31	27.94	37.98	19.51	27.18	30.03	41.11	3.21	4.53	2.51	3.83
MEMIT	5.09	11.50	0.21	0.70	0.07	0.35	18.61	31.71	6.27	9.76	4.18	6.27	10.87	22.65	0.28	0.70	0.00	0.00
PMET	9.34	16.72	1.05	2.09	0.14	0.35	21.60	34.49	11.50	16.38	7.32	10.45	4.11	10.10	0.07	0.35	0.00	0.00
ROME	22.37	39.37	5.30	9.41	3.14	4.18	33.87	50.52	10.24	15.33	7.25	11.15	5.23	15.33	0.21	0.70	0.00	0.00

All three models show substantial reliability degradation under edits. Baseline Full-Pass@5 is 40.54% (CodeLlama-7B), 67.95% (CodeQwen1.5-7B), and 49.03% (DeepSeek-Coder-6.7B). A single FT edit lowers these to 16.03%, 26.48%, and 18.82%, respectively, confirming that even minimal updates can sharply reduce functional correctness.

The performance of CodeQwen1.5-7B degrades in a notably smooth manner, with success rates declining progressively across three distinct stages: initial compilation, partial test case satisfaction, and complete test suite passing. Under GRACE-270, it retains 27.18% Full-Pass@5 (↓40.8) and 53.31% Compiles@5 (↓46.7), suggesting that functional knowledge is distributed and edits remain localized. In case of CodeLlama-7B, compiles@5 remains high at 93.38% (↓4.7) with GRACE-270, but fails to 11.5% under MEMIT (↓86.57, *worst drop in experiments*), while Full-Pass@5 increases to 65.16%. This rise stems from differences in the reliability set; within matched tasks, correctness drops sharply (see RQ 1.4 and Table 9). The widening compile–pass gap indicates that edits are internalized superficially, preserving structure without maintaining functional behavior. DeepSeek-Coder-6.7B is the most brittle. Full-Pass@5 drops to 3.83% under GRACE-270 (↓45.2) and to 0% under MEMIT, PMET, and ROME. Compiles@5 also falls to 44.95% (↓42.31), indicating that accumulated edits degrade both syntax and semantics.

Across models, functional reliability decays faster than syntactic validity. CodeQwen1.5-7B degrades gradually, CodeLlama-7B retains structure while losing correctness, and DeepSeek-Coder-6.7B collapses under perturbation. These trends suggest that edit robustness depends more on representational modularity than model size.

Table 7. Generalization subset: Pass@k rates for *Compiles*, *Partial-Pass* (≥ 1 test passes), and *Full-Pass* (all tests pass) at $k \in \{1, 5\}$ across models and editing methods under instant editing.

Method	CodeLlama-7B						CodeQwen1.5-7B						DeepSeek-Coder-6.7B					
	Compiles		Partial-Pass		Full-Pass		Compiles		Partial-Pass		Full-Pass		Compiles		Partial-Pass		Full-Pass	
	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5
Baseline	92.26	97.56	57.32	66.77	40.79	48.48	98.41	99.09	75.55	82.01	66.22	73.48	73.35	88.11	41.59	55.79	35.43	49.39
FT	37.50	50.93	23.75	31.25	13.84	18.75	36.76	46.53	28.66	36.57	21.94	28.24	40.00	59.95	15.88	25.23	13.06	20.60
GRACE-30	38.75	48.38	25.79	33.10	15.05	19.44	37.36	46.06	29.95	37.04	22.50	27.78	39.12	59.26	15.51	25.23	12.45	19.68
GRACE-90	38.19	50.00	25.37	33.10	14.86	19.21	37.36	46.06	29.95	37.04	22.50	27.78	40.23	59.95	16.30	25.00	13.19	19.68
GRACE-270	37.50	47.22	25.97	31.48	14.54	17.59	37.36	46.06	29.95	37.04	22.50	27.78	38.47	57.41	15.42	23.61	12.36	18.98
MEMIT	8.19	19.44	1.99	4.86	0.83	1.39	36.25	46.53	22.50	29.63	17.04	23.84	14.07	25.46	4.35	8.33	2.87	4.40
PMET	16.85	29.63	6.44	11.34	2.92	6.02	36.48	46.53	26.67	34.03	20.28	25.93	14.21	25.93	4.26	8.56	3.01	5.56
ROME	25.60	43.75	5.60	11.81	2.59	5.09	36.39	47.45	17.64	24.54	12.41	18.75	14.07	25.69	5.19	9.26	3.70	6.25

4.1.2 RQ 1.2: Model Generalization Under Editing. Table 7 presents the generalization performance across multiple editing methods. Generalization measures a model's ability to apply edited knowledge consistently across similar unseen tasks, evaluated through compilation success, partial test passes, and complete test success.

All models show significant performance degradation on generalization tasks after edits. Baseline Full-Pass@5 is 48.48% (CodeLlama-7B), 73.48% (CodeQwen1.5-7B), and 49.39% (DeepSeek-Coder-6.7B). A single FT edit reduces these to 18.75%, 28.24%, and 20.60%, respectively, confirming that even light edits impair generalization.

CodeQwen1.5-7B remains the most robust. Under GRACE-270, it retains 27.78% Full-Pass@5 (↓45.7, *worst drop*) and 57.41% Compiles@5 (↓41.6). All methods show consistent degradation but avoid collapse, suggesting that functional knowledge remains somewhat modular. CodeLlama-7B shows greater sensitivity. Under GRACE-270, Full-Pass@5 drops to 17.59% (↓30.9), and Compiles@5 to 47.22% (↓50.3). While syntax remains partially intact, generalization accuracy degrades sharply, indicating limited edit locality. DeepSeek-Coder-6.7B again degrades most severely. Full-Pass@5 falls to 18.98% under GRACE-270 (↓30.4) and below 7% under MEMIT, PMET, and ROME. Compiles@5 also drops from 88.11% to 57.41%, confirming broader collapse under cumulative edits.

Overall, generalization performance degrades more steeply than seen in the reliability subset. CodeQwen1.5-7B shows the best preservation of generalization under edits, CodeLlama-7B retains syntax but loses correctness, and DeepSeek-Coder-6.7B deteriorates across both. These results underscore the challenge of preserving non-local generalization behavior and highlight the importance of modular internal representations for editability.

4.1.3 RQ 1.3: Do edits cause regressions on unrelated tasks? Editing should ideally be localized, improving behavior without harming unrelated tasks. The specificity subset (Table 8) probes whether edits remain localized or inadvertently degrade performance on unaffected APIs.

Table 8. Specificity subset: Pass@k rates for *Compiles*, *Partial-Pass* (≥ 1 test passes), and *Full-Pass* (all tests pass) at $k \in \{1, 5\}$ across models and editing methods under instant editing.

Method	CodeLlama-7B						CodeQwen1.5-7B						DeepSeek-Coder-6.7B					
	Compiles		Partial-Pass		Full-Pass		Compiles		Partial-Pass		Full-Pass		Compiles		Partial-Pass		Full-Pass	
	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5	@1	@5
Baseline	94.95	99.05	49.81	58.10	33.62	41.43	98.67	99.05	66.67	70.95	54.48	61.90	79.52	91.90	26.10	40.00	19.24	30.00
FT	89.49	97.81	45.04	54.74	29.49	38.69	98.03	98.91	65.99	71.53	54.23	61.68	79.64	92.70	28.98	40.51	21.46	32.12
GRACE-30	94.53	98.91	47.15	55.84	30.80	39.42	98.47	98.91	66.72	71.17	54.16	61.68	79.93	93.43	28.10	43.07	21.09	33.94
GRACE-90	94.01	98.91	47.08	55.84	30.95	38.69	98.47	98.91	66.72	71.17	54.16	61.68	78.61	90.88	28.03	40.51	21.09	31.02
GRACE-270	94.67	98.91	47.66	55.11	31.24	39.05	98.47	98.91	66.72	71.17	54.16	61.68	80.15	91.97	28.91	42.34	21.90	33.94
MEMIT	12.55	23.72	2.55	5.47	0.51	2.19	80.15	87.96	42.77	52.55	32.77	42.70	20.22	29.20	4.82	9.12	3.72	7.66
PMET	29.64	45.99	8.10	15.33	3.36	7.30	92.26	95.26	61.31	66.06	50.36	57.66	19.34	32.48	4.16	7.30	3.36	5.84
ROME	34.74	52.19	9.49	16.79	5.33	8.03	69.78	81.39	27.01	38.32	18.61	28.10	18.10	28.47	6.79	11.31	5.11	8.39

All models retain relatively high specificity after edits, though degradation still occurs. Baseline Full-Pass@5 is 41.43% (CodeLlama-7B), 61.90% (CodeQwen1.5-7B), and 30% (DeepSeek-Coder-6.7B). A single FT edit change these to 38.69%, 61.68%, and 32.12%, respectively, showing moderate impact on targeted behavior preservation.

CodeQwen1.5-7B remains the most stable. Under GRACE-270, it maintains 61.68% Full-Pass@5 (↓0.02) and 98.91% Compiles@5 (↓0.14), with minimal variation across editing methods. This suggests strong localization of edits and minimal interference with unrelated functionality. CodeLlama-7B shows mild but consistent degradation. Full-Pass@5 drops to 39.05% under GRACE-270 (↓2.4), and to 8.03% under ROME (↓33.4). Compilation remains high under GRACE but collapses under non-gradient-based methods (e.g., 45.99% with PMET). This indicates that specific edits are often retained structurally but may disrupt execution semantics depending on the method. DeepSeek-Coder-6.7B again shows the weakest retention. Full-Pass@5 falls from 30% to 7.66% under MEMIT, and below 9% under PMET and ROME. Compilation also drops from 91.9% to 28.47% (ROME), indicating broad interference even in unrelated outputs.

Table 9. Behavioral stability (%) under instant editing across methods and models (using @5). Cells show the percentage of pre-edit correct problems that remain correct after editing.

Method	CodeLlama-7B			CodeQwen1.5-7B			Deepseek-coder-6.7B		
	Reliability	Generalization	Specificity	Reliability	Generalization	Specificity	Reliability	Generalization	Specificity
FT	37.1	34.0	93.1	35.8	35.7	95.4	35.4	37.0	84.1
GRACE-30	24.8	35.2	94.2	38.6	36.1	96.9	1.6	38.3	85.7
GRACE-90	65.7	35.2	92.0	38.6	36.1	96.9	3.9	36.4	79.4
GRACE-270	65.7	32.1	94.2	38.6	36.1	96.9	3.2	36.4	84.1
MEMIT	1.0	1.3	5.8	8.0	29.5	66.1	0.0	8.0	15.9
PMET	1.0	10.7	17.2	14.8	34.0	89.2	0.0	9.9	14.3
ROME	10.5	11.3	18.4	14.2	23.6	39.2	0.0	8.6	14.3

Specificity degrades less than generalization or reliability. CodeQwen1.5-7B preserves unrelated behavior best, CodeLlama-7B maintains syntactic stability but shows method sensitivity, and DeepSeek-Coder-6.7B suffers broad collateral effects. These results reinforce the value of modular representations that constrain the influence of edits to their intended scope.

4.1.4 RQ 1.4: Behavioral Stability Under Model Edits. When a model integrates new API knowledge, it should ideally preserve correctness on problems that were already solved before editing. We refer to this property as *behavioral stability*, which measures the percentage of pre-edit correct problems that remain correct after editing. This metric reflects how much of the model's prior competence survives the editing operation, capturing both interference and forgetting effects.

To compute this metric, we identify the subset of problems that are solved correctly by each model under the baseline configuration at Pass@5. Behavioral stability is then measured as the proportion of those problems that remain solved after the editing, providing a consistent and model-specific baseline for evaluating preservation. Table 9 reports the percentage of pre-edit correct cases that remain correct following each editing method, measured across the reliability, generalization, and specificity subsets.

Across all models, stability under editing is highly uneven. FT and GRACE preserve a moderate portion of pre-edit behaviors, while other methods, such as MEMIT, PMET, and ROME, erase nearly all prior correctness. For instance, under MEMIT, retained correctness falls to 1% for CodeLlama-7B, 8% for CodeQwen1.5-7B, and 0% for DeepSeek-Coder-6.7B.

CodeQwen1.5-7B consistently exhibits the highest behavioral stability across all subsets. Under GRACE-270, it retains 38.6% of reliable outputs, 36.1% of generalizable outputs, and 96.9% of specific behaviors. This balance shows that CodeQwen's parameter updates are well localized and minimally disruptive to pre-existing functionality. CodeLlama-7B shows strong specificity preservation but mixed reliability. Under GRACE-270, it maintains 65.7% reliability, 32.1% generalization, and 94.2% specificity, indicating that previously correct code largely remains valid when edits are localized. However, under parametric edits, retention collapses to single-digit percentages, showing that its stability depends heavily on the type of update. DeepSeek-Coder-6.7B is the most unstable. Even under GRACE, less than 40% of previously correct cases persist across all subsets. Parametric methods further reduce stability to nearly zero across reliability and generalization, confirming that small parameter perturbations spread widely through the model and overwrite prior competence.

4.1.5 RQ 1.5: How robust are editing methods across different code LLMs? Editing methods show inconsistent performance across models. GRACE performs best on CodeLlama, where it preserves syntactic validity (93.38% Compiles@5 under GRACE-270) and retains more correctness than other methods. On CodeQwen, GRACE is moderately effective, with Full-Pass@5 dropping from 67.95% to 27.18%, and compilation remaining above 53%. However, on DeepSeek, GRACE degrades sharply, with Full-Pass@5 falling to 3.83% and Compiles@5 to 41.11%.

Parametric methods such as ROME, MEMIT, and PMET also show model-dependent behavior. They perform moderately on CodeQwen1.5-7B but fail on DeepSeek, where Full-Pass@5 often drops to 0%. CodeLlama-7B retains surface-level structure under these methods but still suffers significant drops in generalization and functional correctness. In summary, some methods work better on certain models but not on others, even though all models use a similar transformer architecture. This indicates that robustness depends more on how models store knowledge internally than on architectural design.

4.1.6 RQ 1.6: How sensitive are outcomes to sampling (Pass@k)? Editing introduces stochastic uncertainty, since a single generation may not fully reflect a model's post-edit behavior. Sampling multiple completions allows us to probe whether improved scores arise from consistent learning or from chance rediscovery of correct variants. Pass@k, therefore, serves as a lens for understanding how stable each model's reliability, generalization, and specificity remain under sampling variation.

Across all subsets, increasing k raises absolute scores but leaves the relative ordering of models and methods unchanged. At baseline, expanding from Pass@1 to Pass@5 improves Full-Pass accuracy by roughly 8–10% on average. In the generalization subset, CodeQwen1.5-7B increases from 66.22% to 73.48% (↑ 7.3), CodeLlama-7B from 40.79% to 48.48% (↑ 7.7), and DeepSeek-Coder-6.7B from 35.43% to 49.39% (↑ 13.9). These uniform improvements show that larger sampling budgets amplify opportunity rather than capability, revealing output diversity without changing relative performance.

Reliability benefits most from higher sampling, since additional completions increase the chance of reproducing previously correct logic after minor perturbations. Generalization and specificity show smaller gains, implying that their errors stem from internal representation shifts rather than generation randomness. Editing methods follow the same trend: FT and GRACE gain modestly with larger k, while parametric methods such as MEMIT, PMET, and ROME remain brittle regardless of sampling scope.

> **Finding 1:** Models exhibit limited robustness to instant editing. A single update often leads to substantial degradation in reliability, generalization, and specificity, indicating that local edits tend to propagate globally. GRACE shows the strongest resilience overall, preserving more pre-edit correctness and maintaining high specificity, though it still incurs drops of up to 45 percentage points in generalization. Other methods are significantly more fragile, with some cases showing drops up to 86 points. Sampling marginally improves success rates by recovering correct variants but does not enhance robustness. Overall, current editing methods offer only partial resilience, and none reliably support isolated, targeted updates across different models.

Implications. These results highlight the need for editing methods that enforce stronger locality and reduce interference with unrelated behaviors. The lack of consistent robustness across models, suggests that internal representational structure is a key factor in how edits propagate. GRACE illustrates the potential of retrieval-based strategies to constrain edits, but its effectiveness varies by model. Parametric approaches, while flexible, often fail to preserve global correctness when applying local updates. Future editing techniques should focus on developing modular representations, incorporating explicit regularization to prevent collateral drift, and improving control over the behavioral scope of edits to support more reliable and generalizable model editing.

Table 10. Sequential editing performance across Reliability, Generalization, and Specificity subsets. Metrics: C = Compiles, P = Partial-Pass (≥1 test), F = Full-Pass (all tests). Pass@k reported for $k \in \{1, 5\}$.

		CodeLlama-7B						CodeQwen-7B						DeepSeek-6.7B					
Subset	**Method**	C@1	C@5	P@1	P@5	F@1	F@5	C@1	C@5	P@1	P@5	F@1	F@5	C@1	C@5	P@1	P@5	F@1	F@5
Reliability	Baseline	92.7	98.1	54.5	65.6	33.8	40.5	99.2	100.0	73.4	79.5	60.4	68.0	72.5	87.3	44.3	61.0	35.2	49.0
	FT	48.6	73.4	14.3	26.6	8.5	15.8	34.2	54.4	0.9	1.9	0.2	0.4	19.3	36.7	1.9	3.9	1.2	2.3
	GRACE-30	32.2	48.6	9.3	16.2	5.6	8.9	28.9	38.6	19.9	25.9	12.8	18.1	14.9	23.6	1.2	2.7	0.7	1.9
	GRACE-90	54.1	56.8	32.9	34.0	30.8	31.7	28.9	38.6	19.9	25.9	12.8	18.1	29.0	42.9	2.3	3.5	1.4	2.3
	GRACE-270	54.8	57.1	33.8	35.1	31.3	32.4	28.9	38.6	19.9	25.9	12.8	18.1	26.3	37.4	2.0	2.7	1.2	1.9
	MEMIT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	PMET	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	ROME	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Generalization	Baseline	92.3	97.6	57.3	66.8	40.8	48.5	98.4	99.1	75.6	82.0	66.2	73.5	73.4	88.1	41.6	55.8	35.4	49.4
	FT	51.9	77.1	15.7	27.1	10.9	18.9	29.6	47.0	0.4	0.9	0.4	0.9	17.7	32.9	2.0	5.2	1.0	3.3
	GRACE-30	23.0	33.5	14.7	20.1	8.4	11.0	27.2	33.5	19.8	23.8	14.8	17.7	30.3	49.4	10.1	16.2	7.7	11.9
	GRACE-90	22.8	32.0	15.7	21.0	9.0	11.6	27.2	33.5	19.8	23.8	14.8	17.7	31.6	50.0	11.6	18.9	9.0	14.6
	GRACE-270	21.5	30.2	14.9	20.1	8.5	11.3	27.2	33.5	19.8	23.8	14.8	17.7	31.2	49.7	10.4	16.5	8.7	14.3
	MEMIT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	PMET	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	ROME	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Specificity	Baseline	95.0	99.0	49.8	58.1	33.6	41.4	98.7	99.0	66.7	71.0	54.5	61.9	79.5	91.9	26.1	40.0	19.2	30.0
	FT	70.8	91.9	16.0	26.7	10.3	16.2	36.1	59.1	4.5	6.7	2.9	4.3	44.4	55.7	6.2	11.9	4.3	9.1
	GRACE-30	94.6	99.5	48.6	57.1	32.7	41.4	99.0	99.0	67.0	71.0	54.8	61.0	80.1	92.9	25.1	38.1	18.9	28.1
	GRACE-90	95.1	99.5	49.6	57.6	33.9	42.4	99.0	99.0	67.0	71.0	54.8	61.0	80.3	92.9	26.4	41.4	19.0	31.4
	GRACE-270	95.4	99.5	49.1	58.1	33.4	43.3	99.0	99.0	67.0	71.0	54.8	61.0	81.0	93.3	26.5	39.5	19.8	31.4
	MEMIT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	PMET	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	ROME	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

4.2 RQ2: What happens when edits accumulate sequentially?

Real software ecosystems evolve through a stream of updates rather than isolated changes. To test whether editing methods can handle such conditions, we apply edits sequentially and track performance as the number of updates grows.

4.2.1 RQ2.1: How does sequential editing compare to instant editing? Sequential editing evaluates whether a model can sustain multiple edits without cumulative degradation. Each update is applied to the already modified model, exposing how successive changes interact and whether prior knowledge remains intact.

Sequential editing causes severe degradation across all models and subsets, as shown in Table 10. Even the strongest configurations lose 60–80% of their baseline performance once edits accumulate. For example, in the reliability subset, CodeLlama-7B drops from 40.5% Full-Pass@5 in the non-edit setting to 32.4% (↓ 8.1) after sequential edits, CodeQwen1.5-7B from 68% to 18.1% (↓ 49.9), and DeepSeek-Coder-6.7B from 49.0% to 2.3% (↓ 46.7). A similar pattern appears in generalization, where Full-Pass@5 falls from 48.5% to 18.9% (↓ 29.6) for CodeLlama-7B and from 73.5% to 17.7% (↓ 55.8) for CodeQwen1.5-7B. Specificity remains relatively stable only under GRACE, while all parametric methods collapse to 0%.

In terms of methods, GRACE remains the most resilient, retaining partial reliability (≈ 30) and specificity (≈ 43). FT performs moderately but still declines sharply after a few edits. Parametric approaches such as MEMIT, PMET, and ROME fail completely under accumulation, producing 0% across all metrics. These results show that direct parameter rewriting is not compositional and quickly destabilizes the model.

In a nutshell, sequential editing magnifies the brittleness observed in instant editing. Even GRACE, the most stable method, cannot prevent cumulative loss as edits interact destructively

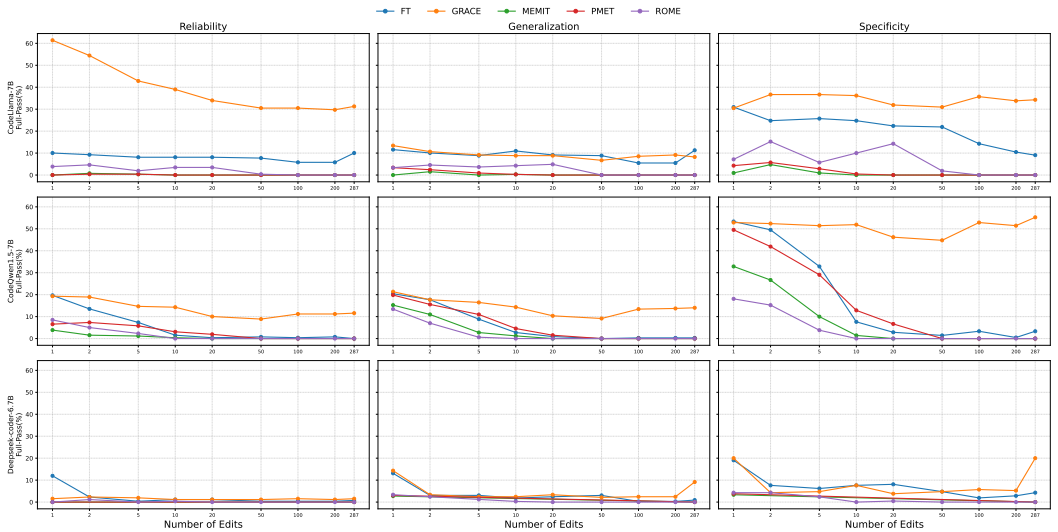

Fig. 1. Performance degradation across sequential edits. Each subplot shows how the *Full-Pass* rate declines as the number of edits increases. Rows correspond to model families (CodeLlama-7B, CodeQwen1.5-7B, and Deepseek-coder-6.7B); columns represent evaluation subsets (*Reliability*, *Generalization*, and *Specificity*). Different curves denote editing methods (FT, GRACE, MEMIT, PMET, and ROME).

over time. Parametric approaches collapse almost immediately, and none of the evaluated methods achieve sustainable robustness once updates are composed.

4.2.2 RQ2.2: How does performance degrade as the number of edits grows? Successive edits introduce cumulative pressure on model stability and representation space. To examine how editing resilience scales, we apply each method sequentially to up to 287 synthetic API updates and track the resulting Full-Pass rate on the evaluation subsets. Figure 1 plots the trajectories for all model–subset pairs.

The decline is rapid in the first few rounds and then gradually flattens as performance approaches a low steady state. Across all models, the first 10–20 edits account for more than half of the total performance loss. After that point, additional edits cause smaller but consistent erosion. For instance, in the reliability subset, CodeLlama-7B drops from about 40% to 25% within the first 20 edits (↓ 15), CodeQwen1.5-7B from 50% to 35% (↓ 15), and DeepSeek-Coder-6.7B from 49% to below 10% (↓ 39). A similar early collapse appears in generalization, where CodeLlama-7B falls from 48% to 20% and CodeQwen1.5-7B from 73% to 30% in roughly the same range. Specificity remains the most stable, with all models retaining above 50% until after about 50 edits, after which parametric methods begin to diverge sharply.

Across methods, GRACE demonstrates the slowest degradation curve, maintaining moderate accuracy even after a couple of hundred edits. FT declines faster, while MEMIT, PMET, and ROME collapse almost immediately after the first few edits. GRACE's smoother curve suggests that gradient-based updates distribute change more gradually across parameters, delaying – but not preventing – the eventual collapse.

Across models, CodeQwen1.5-7B again shows the highest resilience, followed by CodeLlama-7B, while DeepSeek-Coder-6.7B deteriorates most rapidly. The early part of the curve reveals that DeepSeek-Coder-6.7B loses nearly all reliability by 20 edits, whereas the other two models maintain partial functionality for much longer.

Overall, sequential editing exhibits a nonlinear degradation pattern: large losses occur early, followed by a long tail of gradual decline. GRACE moderates this collapse but cannot sustain

stable performance as edits accumulate. These results highlight that current models and algorithms remain highly sensitive to edit accumulation, where each additional modification compounds prior instability.

> **Finding 2:** Editing performance deteriorates rapidly under sequential edits. Across all models, even small edit budgets result in sharp declines, with 60 to 80 percent of baseline correctness lost. Most of this degradation occurs within the first 10 to 20 edits. This setting reveals that robustness declines quickly as edits accumulate. GRACE shows the slowest degradation, maintaining moderate functionality across hundreds of edits, while parametric methods collapse almost immediately. CodeQwen1.5-7B is the most resilient overall, CodeLlama-7B retains partial functionality, and DeepSeek-Coder-6.7B degrades early and severely. Sampling provides no meaningful protection, as accumulated edits increase instability rather than mitigating it. No method sustains reliable performance beyond a modest number of edits.

Implications. Sequential editing amplifies the brittleness already evident in instant editing. Parametric methods fail quickly, while GRACE degrades more gradually but remains vulnerable to cumulative interference. Sampling does not counteract this trend and offers no stability under continued updates. These findings highlight the absence of mechanisms in current methods to isolate edits or preserve prior behaviors over time. Sustaining robustness under sequential editing will require new techniques that enforce locality, mitigate interference, and maintain functional correctness across long edit sequences.

4.3 RQ3: Do edited models truly adapt to new APIs, or do they work around?

When a library deprecates an API, an ideal edit should enable the model to adopt the new interface correctly and produce compilable, behaviorally consistent code that reflects the intended update. In practice, however, models often exhibit only partial or superficial adaptation. They may insert the new API name syntactically without preserving its semantics, or bypass the update entirely using functionally equivalent but unintended constructs. Understanding whether edited models genuinely adapt or merely imitate the form of adaptation is critical to assessing the reliability of current editing techniques.

To examine this question, we focus on two complementary outcomes: (1) *API Adoption*, which measures how often edited models use the target API, and (2) *Workarounds*, which measure how often models pass tests without invoking the updated interface. Within adoption, we distinguish between *Correct Adoption (CA)*, which both compiles and passes all tests, and *Faulty Adoption (FA)*, which compiles but fails to reproduce correct behavior.

Table 11. Breakdown of API Adoptions under instance and sequential editing. Each cell reports percentages of *Correct Adoption (CA)* and *Faulty Adoption (FA)* at sampling levels k=1, 3, and 5. Only cases where the new API appears are included.

Model	Instant Editing						Sequential Editing					
	CA@1	FA@1	CA@3	FA@3	CA@5	FA@5	CA@1	FA@1	CA@3	FA@3	CA@5	FA@5
CodeLlama-7B	6.2	3.1	6.4	3.6	6.5	3.9	3.5	8.5	3.9	9.8	4.0	10.4
CodeQwen1.5-7B	0	0.1	0	0.1	0	0.2	0	3.9	0	4.8	0	5.2
Deepseek-coder-6.7B	0.2	8.1	0.3	9.4	0.3	9.8	0.1	11.6	0.2	13.8	0.2	14.4

Adoption Behavior. Table 11 shows that correct adoptions (CA) remain low and show no meaningful improvement. CodeLlama-7B reaches 6.5% CA@5 under instant editing but slightly declines to

Table 12. Distribution of Workaround outcomes under instance and sequential editing. Each cell shows the percentage of generated solutions that passed tests *without using the updated API*.

Model	Instant Editing			Sequential Editing		
	WA@1	WA@3	WA@5	WA@1	WA@3	WA@5
CodeLlama-7B	10.3	12.8	13.9	7.1	8.8	9.5
CodeQwen1.5-7B	26.4	31.1	32.9	11.9	13.5	14.0
Deepseek-coder-6.7B	8.1	11.5	12.9	4.7	6.9	7.8

4% under sequential editing. CodeQwen1.5-7B remains at 0% CA, failing to produce any successful migrations across both editing regimes. DeepSeek-Coder-6.7B similarly stagnates near zero (0.3 and 0.2, respectively). In contrast, faulty adoptions (FA) rise substantially. For CodeLlama, FA increases from 3.9% to 10.4%; for CodeQwen, from 0.2% to 5.2%; and for DeepSeek-Coder, from 9.8% to 14.4%. This trend reveals that models increasingly substitute the correct API token without implementing its intended behavior, resulting in fragile or broken outputs.

Workarounds. Sequential editing leads to a modest decline in workarounds (Table 12): from 13.9% to 9.5% in CodeLlama, from 32.9% to 14.0% in CodeQwen, and from 12.9% to 7.8% in DeepSeek-Coder. However, these declines are offset by the concurrent rise in faulty adoptions, suggesting that models are not learning to integrate the new API semantically but are instead replacing workarounds with faulty adoptions.

Table 13. CA/FA/WA breakdown by model and editing method, under instance and sequential regimes. Values are percentages of edited tasks, aggregated across all subsets.

Model	Method	Instant Editing						Sequential Editing					
		CA@1	FA@1	WA@1	CA@5	FA@5	WA@5	CA@1	FA@1	WA@1	CA@5	FA@5	WA@5
CodeLlama-7B	FT	0.0	0.0	17.8	0.0	0.0	24.5	3.3	23.4	6.6	6.1	34.0	10.9
	GRACE-30	1.5	3.1	17.2	2.3	4.8	22.9	0.8	2.8	14.7	1.0	3.6	19.4
	GRACE-90	20.8	8.3	15.3	21.5	9.5	19.3	10.2	17.0	14.4	10.4	17.7	18.1
	GRACE-270	21.0	8.7	15.3	21.7	9.6	18.9	10.4	16.4	14.0	10.8	17.4	18.2
	MEMIT	0.0	0.3	0.5	0.0	0.8	1.3	0.0	0.0	0.0	0.0	0.0	0.0
	PMET	0.0	0.2	2.1	0.0	0.8	4.6	0.0	0.0	0.0	0.0	0.0	0.0
	ROME	0.0	0.6	3.7	0.0	1.4	5.8	0.0	0.0	0.0	0.0	0.0	0.0
CodeQwen1.5-7B	FT	0.0	0.0	31.6	0.0	0.0	38.8	0.2	27.4	1.0	0.3	36.5	1.5
	GRACE-30	0.0	0.0	32.1	0.0	0.0	38.9	0.0	0.0	27.5	0.0	0.0	32.3
	GRACE-90	0.0	0.0	32.1	0.0	0.0	38.9	0.0	0.0	27.5	0.0	0.0	32.3
	GRACE-270	0.0	0.0	32.1	0.0	0.0	38.9	0.0	0.0	27.5	0.0	0.0	32.3
	MEMIT	0.0	0.2	18.0	0.0	0.5	24.3	0.0	0.0	0.0	0.0	0.0	0.0
	PMET	0.0	0.1	26.0	0.0	0.2	31.3	0.0	0.0	0.0	0.0	0.0	0.0
	ROME	0.1	0.2	12.7	0.1	0.4	19.3	0.0	0.0	0.0	0.0	0.0	0.0
Deepseek-coder-6.7B	FT	0.0	0.0	15.4	0.0	0.0	23.8	0.3	20.6	1.9	0.5	29.1	4.4
	GRACE-30	0.1	6.3	11.3	0.4	8.0	18.6	0.0	5.8	9.1	0.0	8.1	14.0
	GRACE-90	0.7	24.7	11.5	0.8	28.7	17.2	0.3	23.0	9.5	0.4	26.8	15.7
	GRACE-270	0.7	25.2	11.5	0.9	29.2	18.0	0.3	23.9	9.6	0.4	27.4	15.5
	MEMIT	0.0	0.1	2.2	0.0	0.3	4.0	0.0	0.0	0.0	0.0	0.0	0.0
	PMET	0.0	0.4	2.1	0.0	1.4	3.8	0.0	0.0	0.0	0.0	0.0	0.0
	ROME	0.0	0.3	2.9	0.0	1.2	4.9	0.0	0.0	0.0	0.0	0.0	0.0

Across Methods. Editing methods vary in their ability to balance correct adoption and workaround suppression. Table 13 reveals that GRACE contributes the vast majority of correct adoptions across settings, particularly on CodeLlama-7B, where GRACE-90 and GRACE-270 reach 21.5% and 21.7% CA@5 under instant editing and retain around 10.4% under sequential edits. Other methods fail to deliver meaningful, correct adoption. The only exception is FT, which reaches 6.1% on CodeLlama, 0.3% on CodeQwen, and 0.5% on DeepSeek-Coder-6.7B under sequential editing. All other techniques, including MEMIT, PMET, and ROME, remain at zero.

Table 14. Distribution of Compilation Errors before and after editing. Values indicate relative frequency (%) of each error class.

Error Type	Before Editing(%)	After Editing(%)
IndentationError	44.5	25.0
SyntaxError	28.9	52.5
NameError	19.7	16.5
TypeError	4.6	3.6
IndexError	0.8	0.8
ValueError	0.5	0.4
TimeoutError	0.5	0.2
AttributeError	0.2	0.3
RecursionError	0.2	0.2
No-Code	0.1	0.2
New Classes:		
ModuleNotFoundError	—	0.2
KeyError	—	0.1
UnboundLocalError	—	0.1

Faulty adoptions (FA) increase substantially under sequential editing. FT, for instance, climbs from 0% to 34% FA@5 on CodeLlama-7B and from 0.0% to 36.5% on CodeQwen. GRACE variants also show rising FA rates across models. This pattern suggests that sequential exposure encourages API insertion but fails to ensure semantic correctness. Workaround (WA) behavior remains common under instant editing, especially for FT and GRACE. FT uses 24.5% WA@5 on CodeLlama-7B and 38.8% on CodeQwen. Under sequential editing, however, WA rates generally decline. On CodeQwen, FT workarounds drop from 38.8% to 1.5%, and GRACE-90 from 38.9% to 32.3%.

> **Finding 3:** Edited models show limited and often superficial adaptation to new APIs. GRACE remains the only method consistently supporting correct adoption, though it declines in sequential edits. FT shows modest gains under sequential editing but contributes heavily to faulty updates. Other methods fail entirely to enable meaningful adoption. These results highlight that current methods can encourage surface-level changes but struggle to deliver semantically grounded adaptations.

Implications. Current editing methods prioritize surface substitution over semantic understanding. Correct adoption remains low, mostly limited to GRACE. Other methods, including FT, show minimal improvement. The rise in faulty adoptions under sequential edits indicates that models increasingly include the new API without using it correctly. Workarounds decline slightly but are replaced by brittle or incorrect updates rather than meaningful adoption. These trends suggest that models appear responsive to edits but do not internalize their intended semantics. Robust editing requires mechanisms that enforce consistent and functional adaptation, not just lexical change.

4.4 RQ4: Where in the pipeline do edits fail?

Editing success unfolds as a cascade of conditional achievements: the model must first produce compilable code, then exhibit consistent behavioral correctness, and finally integrate the target API without breaking unrelated functionality. We therefore ask: at which stage do current methods fail most often, and what patterns of degradation reveal about their internal representations? We examine this question through three sub-RQs corresponding to the taxonomy in Figure 2.

4.4.1 RQ 4.1: Where do edits fail syntactically? Syntactic validity is a necessary precondition for functional editing. Edits fail syntactically when the model either does not incorporate the target

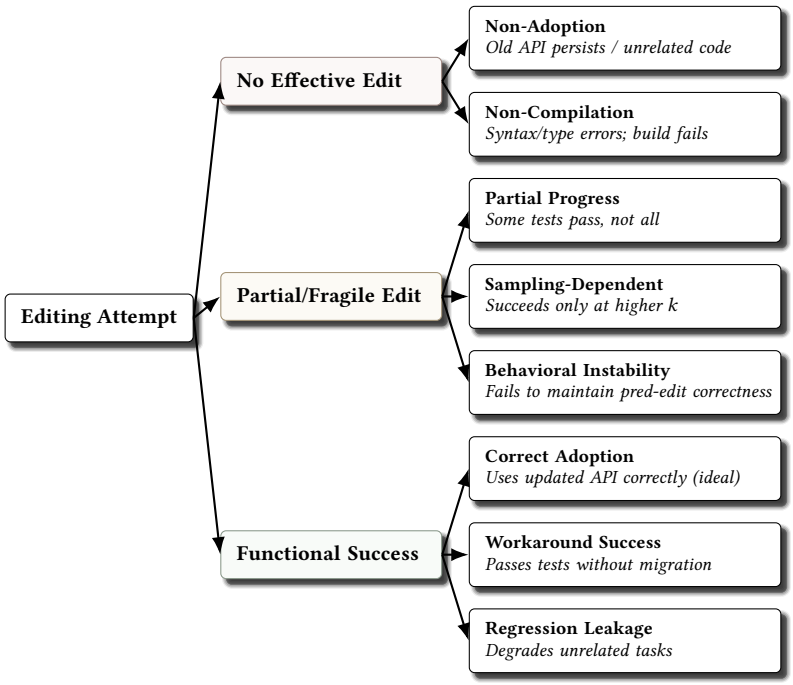

Fig. 2. Taxonomy of outcomes when editing code LMs for API evolution.

API (Non-Adoption) or generates code that fails to compile (Non-compilation). These failure modes correspond to the *No Effective Edit* branch in the taxonomy (Fig. 2).

(1) Non-Adoption. This remains the most common syntactic failure mode. CodeLlama, CodeQwen, and DeepSeek-Coder-6.7B account for 28.50 %, 54.34%, and 17.16% of such cases, respectively. Across methods, FT, GRACE-30, GRACE-90, GRACE-270, MEMIT, PMET, and ROME contribute 22.14%, 20.57%, 18.11%, 18.07%, 6.18%, 8.49%, and 6.45%. These failures arise when edits neither modify the relevant call sites nor insert the target API, indicating that the update failed to propagate to the generation.

(2) Non-Compilation. In these cases, produced code fails to compile. This includes common issues such as incorrect indentation, syntax errors, undefined variables, and type mismatches. These cases mark an early form of incorrect adaptation: the model recognizes the need to adopt but fails to implement it consistently. Across models, CodeLlama-7B contributes 36.29%, CodeQwen1.5-7B 15.28%, and DeepSeek-Coder-6.7B 48.42% of such errors. Across methods, FT, GRACE-(30/90/270), MEMIT, PMET, and ROME account for 6.73%, 8.44%, 8.13%, 8.24%, 23.99%, 21.52%, and 22.95%, respectively.

Table 14 shows that editing not only shifts existing error frequencies but also introduces new failure classes. While `IndentationError` and `NameError` decline after editing, the share of `SyntaxError` nearly doubles (28.9% → 52.5%). Newly emerging classes such as `ModuleNotFoundError`, `KeyError`, and `UnboundLocalError` appear exclusively after editing, suggesting that altered imports and bindings occasionally break dependency resolution. These shifts indicate that gradient-based edits (FT, GRACE) maintain syntactic integrity, whereas memory-based methods induce unstable structural rewrites that compromise compile-time reliability.

4.4.2 RQ 4.2: Where do edits fail behaviorally? After achieving syntactic validity, the next challenge is behavioral stability. This stage corresponds to Partial or Fragile Edit in the taxonomy, where

Table 15. Distribution of partial progress by editing method and model. Values show the percentage of cases with partial test pass.

Editing Method	CodeLlama-7B	CodeQwen1.5-7B	Deepseek-coder-6.7B
FT	6.5	4.5	6.5
GRACE-30	7.0	4.6	5.9
GRACE-90	6.5	4.6	6.6
GRACE-270	6.5	4.6	6.5
MEMIT	1.5	4.7	2.3
PMET	2.9	4.2	1.9
ROME	4.4	5.9	1.8

generations compile and run but do not pass all test cases. These failures reflect incomplete or unstable behavioral adaptation, where edits affect local syntax without consistently updating the underlying logic.

Approximately 26% of edited generations fall into this stage. CodeLlama-7B accounts for 35.34%, CodeQwen1.5-7B for 33.09%, and DeepSeek-Coder-6.7B for 31.57%. FT and GRACE variants dominate this group, contributing around 70% of these cases. This suggests that FT and GRACE often produce structurally correct code but fail to ensure consistent functional success.

(1) Partial Progress. These cases involve generations that pass only a subset of tests. The model typically updates the immediate call site but fails to apply changes across dependent logic. This indicates that the edit modifies local syntax without adjusting the broader control flow or program dependencies. As shown in Table 15, FT and GRACE exhibit the highest rates of partial progress, especially on CodeLlama-7B, where GRACE-30 reaches 7% and FT reaches 6.5%. CodeQwen1.5-7B and DeepSeek-Coder-6.7B show overall lower rates, with partial errors concentrated around 4–5% for most methods. In contrast, memory-based approaches such as MEMIT, PMET, and ROME produce lower partial progress rates, as they tend to either fail completely or produce brittle rewrites that do not execute successfully.

(2) Sampling Fragility. Behavioral outcomes are often sensitive to sampling. Increasing from Pass@1 to Pass@5 improves success rates by 8 to 10 percent (see RQ1). This instability suggests that edited behaviors lie in uncertain regions of the model's output distribution. GRACE yields more stable outputs across samples, while FT shows greater fluctuation, indicating that even gradient-based updates lack consistent behavioral alignment.

(3) Behavioral Instability. It reflects how often edits unintentionally disrupt functional outputs that were correct before modification. CodeLlama-7B accounts for 33.59% of all broken generations, CodeQwen1.5-7B for 26.73%, and DeepSeek-Coder-6.7B for 39.68%, measured over generations that were correct pre-edit. In terms of method contribution, these failures primarily stem from memory-based methods. MEMIT, PMET, and ROME together are responsible for 51.07% of all post-edit failures, despite handling the same number of edits as other methods. In contrast, FT and GRACE variants account for a smaller share, reflecting better behavioral preservation but still contributing a non-trivial portion of regressions.

4.4.3 RQ 4.3: Where do edits fail functionally or regress globally? The final stage, *Functional Success*, includes edits that pass all test cases. However, correctness alone does not guarantee robust or meaningful integration of the intended update. Even when outputs are fully correct, success may arise from superficial strategies or hidden side effects. This stage includes three distinct outcomes: Correct Adoption, Workaround Success, and Regression Leakage.

(1) Correct Adoption. This represents ideal editing behavior, the model adopts the new API, compiles, and passes all tests. Yet this outcome remains rare. Across instant edits, only about 11% of successful generations qualify as true migrations. Most of these come from CodeLlama-7B (96.18% of correct adoptions), with GRACE(90) and GRACE(270) producing the highest shares at 47.94%

and 48.19 %. DeepSeek-Coder-6.7B and CodeQwen1.5-7B contribute only 3.56% and 0.26%. These results show that the functional correctness of new APIs remains largely confined to models that already exhibit strong syntactic and behavioral stability.

(2) Workaround Success. It describes solutions that pass all tests without invoking the new API. Roughly 88% of fully correct generations fall into this category, indicating that models often achieve functional success by reverting to legacy code paths or using ad-hoc logic rather than adopting the intended abstraction. FT and GRACE variants account for most workarounds, while memory-based edits rarely produce such outcomes because they fail earlier in the pipeline. This behavior reflects partial functional alignment—models learn to satisfy the task objective but not to internalize the underlying update semantics.

(3) Regression Leakage. Functional success in the edited task may come at the cost of unrelated behaviors. This regression occurs when edits unintentionally degrade performance on tasks that were previously solved correctly. Across the specificity subset, 30.81 percent of generations exhibit such regressions. FT and GRACE-(30/90/270) each contribute approximately 19 percent of these failures, with MEMIT, PMET, and ROME accounting for the rest. By model, CodeQwen1.5-7B shows the highest leakage at 38.19 percent, followed by CodeLlama-7B at 34.89 percent and DeepSeek-Coder-6.7B at 26.92 percent. These regressions indicate that edits may overwrite shared internal representations, leading to unintended side effects beyond the intended scope.

> **Finding 4:** Most edits fail early in the pipeline. Memory-based methods (ROME, MEMIT, PMET) often break syntax or omit the new API entirely. FT and GRACE preserve compilation but produce fragile behaviors that pass tests inconsistently or only under certain samples. Even successful generations rely mainly on workarounds rather than true API migration, and about 30% trigger regressions on unrelated tasks. Overall, current methods stabilize structure but not semantics, achieving local correctness without global reliability.

Implications. Functional correctness is not sufficient to indicate successful model editing. Most models learn to satisfy task objectives without internalizing the intended change. Workarounds dominate, undermining the reliability and traceability of edits. Even correct solutions frequently degrade unrelated capabilities, highlighting the absence of internal constraints that isolate edits. These findings suggest that future editing techniques must go beyond local parameter shifts. Effective editing should enforce semantic adoption, resist fallback behavior, and explicitly preserve global model functionality.

5 Related Work

Model editing techniques have evolved rapidly, and can be classified along multiple dimensions, including whether they modify model parameters directly or rely on external or contextual mechanisms, the granularity of edits (single vs. batched), the scope of layers affected, and their stability under sequential edits [20]. Parameter-based editing methods alter internal weights to encode new knowledge. Among these, ROME [30] identifies and updates factual associations via rank-one modifications to transformer feed-forward layers, while MEMIT [31] extends this to perform batched edits at scale. MEND [32] uses a hypernetwork to predict low-rank updates, offering greater efficiency and stability. Conversely, parameter-free or indirect editing approaches, such as SERAC [33] and GRACE [20], avoid directly changing weights, relying instead on local fine-tuning, external memories, or causal graph reasoning to achieve consistent and generalizable edits. Retrieval- and context-based editing methods [4] represent another line of work, dynamically conditioning model outputs on edited contexts without altering underlying parameters. Recent unified frameworks [19]

further formalize model editing as a balance between preservation and memorization, highlighting trade-offs in locality, generalization, and retention across methods.

While most model editing research has focused on natural language LLMs, recent work extends these methods to LLMs for Code, addressing challenges arising from the structured and semantic nature of programming languages. For example, MINT [16] repairs next-token prediction errors in code generation by patching a small set of neurons, demonstrating fine-grained code behavior correction. CodeUpdateArena [27] introduces a benchmark for assessing model editing on API-function changes, simulating realistic software evolution scenarios. However, their evaluation is limited to fine-tuning approaches and does not systematically compare multiple editing techniques. Building on these efforts [26], systematically evaluates state-of-the-art editing techniques across multiple code tasks and introduces A-GRACE, an enhanced GRACE variant, but does not explore the compiler integration requirements necessary for realistic code editing. Collectively, these contributions highlight a growing research trend toward code-aware, benchmark-driven, and semantically robust model editing for large-scale code generation systems.

Our work differs in two main ways. First, we conduct a systematic evaluation of five state-of-the-art model editing methods (FT, ROME, MEMIT, PMET, and GRACE) on three code-oriented large language models (LLMs). Performing such an evaluation on real-world API changes is challenging because these models are trained on different data sources with different cutoff dates. Consequently, one model may have been exposed to a given API update during pretraining whereas another may not have, rendering direct comparisons unreliable. Synthetic APIs eliminate this data leakage, ensuring every model faces identical conditions and attributing outcomes to the editing method rather than training data differences. Second, we address unique code editing requirements that distinguish it from natural language: synthetic APIs must be integrated into the compiler environment while deprecated versions are removed, and edits must pass execution-based unit tests. Unlike natural language where edits can be validated at the surface level, code edits must survive compilation and runtime checks to be considered correct.

6 Conclusion

This study presents the first systematic evaluation of model editing for code language models and reveals that existing approaches offer only limited and unstable adaptation. Across five editing methods and three open-source models, syntactic integration of updates is slow and fragile, while semantic reliability declines sharply. Instant edits already reduce reliability and generalization by more than 80 percent points, and sequential edits amplify interference, often leading to functional collapse. Although compilation rates remain higher, correct API adoption remains rare, faulty adoptions dominate, and many passing outputs rely on workarounds that bypass the intended change. Even when edits succeed locally, they frequently introduce behavioral drift and regressions in unrelated tasks, showing that edits propagate globally rather than remaining isolated. By analyzing results across reliability, generalization, specificity, and adoption dimensions, we identify consistent failure patterns. Future work should focus on developing code-specific editing strategies that couple parameter updates with program-level reasoning. Such methods must preserve functional correctness across edits, manage interference in cumulative updates, and ensure stability as APIs and software ecosystems evolve.

References

[1] Amazon. 2023. Amazon CodeWhisperer: Build applications faster and more securely with your AI coding companion. https://aws.amazon.com/codewhisperer/.

[2] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma,

Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. *arXiv preprint arXiv:2309.16609* (2023).

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL] https://arxiv.org/abs/2005.14165

[4] Nicola De Cao, Wilker Aziz, and Ivan Titov. 2021. Editing Factual Knowledge in Language Models. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP2021).* https://arxiv.org/abs/2104.08164

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] OpenJS Foundation / Node.js contributors. 2025. Deprecations — Node.js API (latestv20.x). https://nodejs.org/docs/latest-v20.x/api/deprecations.html. Accessed: September 20, 2025.

[8] Oracle Corporation. 2025. Deprecated List — Java SE 23 API Documentation. https://docs.oracle.com/en/java/javase/23/docs/api/deprecated-list.html. Accessed: September 20, 2025.

[9] NumPy Developers. 2024. NumPy 2.0.0 Release Notes. https://numpy.org/doc/2.0/release/2.0.0-notes.html. Accessed: September 20, 2025.

[10] NumPy Developers. 2025. NumPy. https://numpy.org/. Accessed: September 20, 2025.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).* 4171–4186.

[12] 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. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[13] Node.js Foundation. 2025. Node.js. https://nodejs.org/. Accessed: September 20, 2025.

[14] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. 2021. Transformer Feed-Forward Layers Are Key-Value Memories. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 5484–5495. doi:10.18653/v1/2021.emnlp-main.446

[15] GitHub. 2021. GitHub Copilot: Your AI Pair Programmer. https://copilot.github.com/.

[16] 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

[17] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE] https://arxiv.org/abs/2401.14196

[18] Akshat Gupta, Anurag Rao, and Gopala Anumanchipalli. 2024. Model Editing at Scale leads to Gradual and Catastrophic Forgetting. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 15202–15232. doi:10.18653/v1/2024.findings-acl.902

[19] Akshat Gupta, Dev Sajnani, and Gopala Anumanchipalli. 2024. A Unified Framework for Model Editing. arXiv:2403.14236 [cs.LG] https://arxiv.org/abs/2403.14236

[20] Thomas Hartvigsen, Swami Sankaranarayanan, Hamid Palangi, Yoon Kim, and Marzyeh Ghassemi. 2023. Aging with GRACE: Lifelong Model Editing with Discrete Key-Value Adaptors. In *Advances in Neural Information Processing*

Systems.

[21] Peter Hase, Mohit Bansal, Been Kim, and Asma Ghandeharioun. 2023. Does Localization Inform Editing? Surprising Differences in Causality-Based Localization vs. Knowledge Editing in Language Models. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=EldbUlZtbd

[22] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 [cs.LG] https://arxiv.org/abs/1909.09436

[23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL] https://arxiv.org/abs/2305.06161

[24] Xiaopeng Li, Shasha Li, Shezheng Song, Huijun Liu, Bin Ji, Xi Wang, Jun Ma, Jie Yu, Xiaodong Liu, Jing Wang, and Weimin Zhang. 2025. SWEA: updating factual knowledge in large language models via subject word embedding altering. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'25/IAAI'25/EAAI'25)*. AAAI Press, Article 2730, 9 pages. doi:10.1609/aaai.v39i23.34628

[25] 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.

[26] 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?. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 937–949. doi:10.1109/ICSE55347.2025.00049

[27] Zeyu Leo Liu, Shrey Pandit, Xi Ye, Eunsol Choi, and Greg Durrett. 2025. CodeUpdateArena: Benchmarking Knowledge Editing on API Updates. arXiv:2407.06249 [cs.CL] https://arxiv.org/abs/2407.06249

[28] Google LLC. 2024. API Differences Between 34 and 35 — Android Developers. https://developer.android.com/sdk/api_diff/35/changes. Accessed: September 20, 2025.

[29] Google LLC. 2025. Android Developers. https://developer.android.com. Accessed: September 20, 2025.

[30] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and Editing Factual Associations in GPT. *Advances in Neural Information Processing Systems* 36 (2022). arXiv:2202.05262.

[31] Kevin Meng, Arnab Sen Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. 2023. Mass Editing Memory in a Transformer. *The Eleventh International Conference on Learning Representations (ICLR)* (2023).

[32] Eric Mitchell, Charles Lin, Antoine Bosselut, Chelsea Finn, and Christopher D Manning. 2022. Fast Model Editing at Scale. In *International Conference on Learning Representations*. https://openreview.net/pdf?id=0DcZxeWfOPt

[33] Eric Mitchell, Charles Lin, Antoine Bosselut, Chelsea Finn, and Christopher D. Manning. 2022. Memory-Based Model Editing at Scale. In *International Conference on Machine Learning*. https://arxiv.org/pdf/2206.06520.pdf

[34] Oracle. 2025. Java Platform, Standard Edition Documentation. https://docs.oracle.com/en/java/javase/. Accessed: September 20, 2025.

[35] The pandas development team. 2022. Deprecations — pandas 1.5.0. https://pandas.pydata.org/pandas-docs/version/1.5/whatsnew/v1.5.0.html#deprecations. Accessed: September 20, 2025.

[36] The pandas development team. 2022. pandas: pandas.concat. https://pandas.pydata.org/docs/reference/api/pandas.concat.html Accessed: 2025-09-20.

[37] The pandas development team. 2022. pandas: pandas.DataFrame.append. https://pandas.pydata.org/pandas-docs/version/1.4/reference/api/pandas.DataFrame.append.html Accessed: 2025-09-20.

[38] The pandas development team. 2025. pandas — Python Data Analysis Library. https://pandas.pydata.org/. Accessed: September 20, 2025.

[39] Google Research. 2025. mbpp: Mostly Basic Python Problems. https://github.com/google-research/google-research/tree/master/mbpp. Accessed: 2025-08-15.

[40] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[41] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton

Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[43] Jesse Vig, Sebastian Gehrmann, Yonatan Belinkov, Sharon Qian, Daniel Nevo, Yaron Singer, and Stuart Shieber. 2020. Investigating Gender Bias in Language Models Using Causal Mediation Analysis. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 12388–12401. https://proceedings.neurips.cc/paper_files/paper/2020/file/92650b2e92217715fe312e6fa7b90d82-Paper.pdf

[44] Peng Wang, Ningyu Zhang, Bozhong Tian, Zekun Xi, Yunzhi Yao, Ziwen Xu, Mengru Wang, Shengyu Mao, Xiaohan Wang, Siyuan Cheng, Kangwei Liu, Yuansheng Ni, Guozhou Zheng, and Huajun Chen. 2024. EasyEdit: An Easy-to-use Knowledge Editing Framework for Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Yixin Cao, Yang Feng, and Deyi Xiong (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 82–93. doi:10.18653/v1/2024.acl-demos.9

[45] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. doi:10.18653/v1/2021.emnlp-main.685

[46] 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* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 476–486. doi:10.1145/3196398.3196408

[47] Chen Zhu, Ankit Singh Rawat, Manzil Zaheer, Srinadh Bhojanapalli, Daliang Li, Felix Yu, and Sanjiv Kumar. 2020. Modifying Memories in Transformer Models. arXiv:2012.00363 [cs.CL] https://arxiv.org/abs/2012.00363