Evaluating Incompatible Third-party Library API Usage in LLM-based Code Completion

LI LIN, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
YAORUI FEI, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
YUNFENG SHEN, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
LINGFENG BAO*, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China
ZHONGXIN LIU, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
DAVID LO, Singapore Management University, Singapore

Large Language Models (LLMs) have advanced code completion, but their ability to generate API usages compatible with evolving third-party libraries (TPLs) remains uncertain. As TPL APIs frequently change, LLMs risk producing code incompatible with installed library versions, causing build failures or incorrect behaviors. We define such issues as Incompatible Third-party Library API Usage (ITAU) and conduct a systematic study to evaluate how state-of-the-art LLMs handle this challenge.

To this end, we propose an automated framework that builds a versioned TPL API Knowledge Base and a large-scale benchmark with 10,867 realistic code completion tasks. Through comprehensive evaluation of six state-of-the-art LLMs, we find that even top-performing models frequently generate incompatible completions. We further propose two lightweight solutions, Real-time Detection and Lightweight Repair, to mitigate ITAUs. This framework and benchmark provide a foundation for more compatibility-aware code generation in evolving software ecosystems.

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

Additional Key Words and Phrases: Large Language Models, Code Completion, Third-party Libraries, API Compatibility

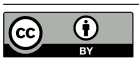
ACM Reference Format:
Li Lin, Yaorui Fei, Yunfeng Shen, Lingfeng Bao, Zhongxin Liu, and David Lo. 2026. Evaluating Incompatible Third-party Library API Usage in LLM-based Code Completion. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2026), 29 pages. https://doi.org/10.1145/3830085

1 introduction

Large Language Models (LLMs) [1–5] have transformed code completion, shifting from traditional statistical methods [6, 7] to context-aware generation [8, 9]. Pre-trained or fine-tuned on large-scale

*Corresponding Author

Authors' Contact Information: Li Lin, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, linli1210@zju.edu.cn; Yaorui Fei, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, fyr031109@outlook.com; Yunfeng Shen, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, shenyunfeng@zju.edu.cn; Lingfeng Bao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China, lingfengbao@zju.edu.cn; Zhongxin Liu, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, liu_zx@zju.edu.cn; David Lo, Singapore Management University, Singapore, davidlo@smu.edu.sg.



ACM 1557-7392/2026/1-ART
https://doi.org/10.1145/3830085

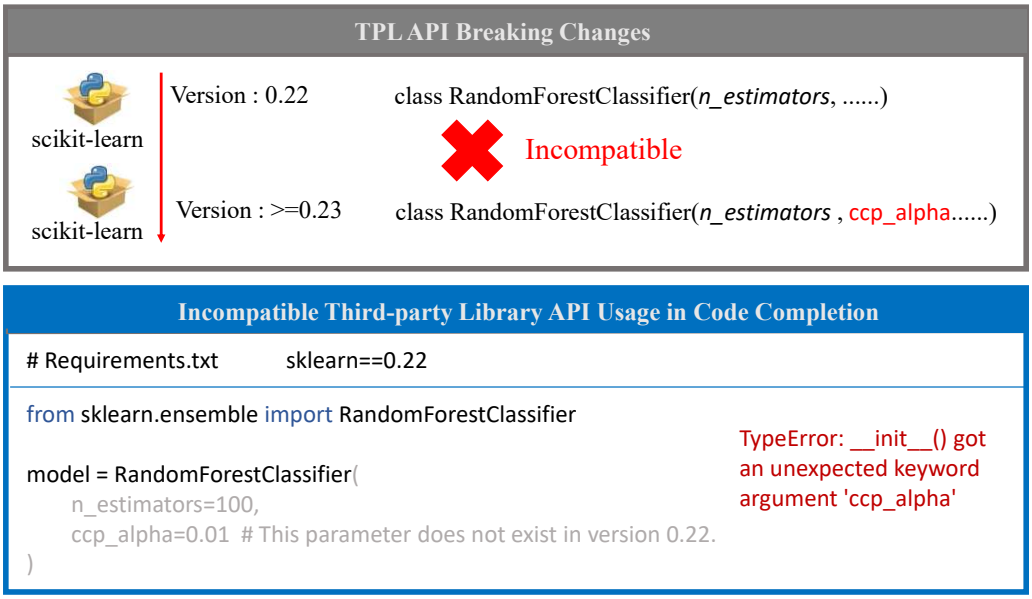


Fig. 1. **An Example of Incompatible Third-party Library API Usage (ITAU) Caused by Library Evolution**

code corpora, these models can generate syntactically and semantically relevant suggestions, from tokens and variables to methods and higher-level code structures.

Problem. To streamline development, modern software engineering practices extensively leverage third-party libraries (TPLs) via Application Programming Interfaces (APIs). However, this dependency introduces new challenges for code completion systems. In particular, the software ecosystem is inherently dynamic—libraries evolve rapidly, leading to frequent updates, deprecations, or modifications of API definitions. When a code completion tool suggests APIs incompatible with the current library version, it may introduce signature-level compatibility risks and, depending on the downstream execution context, lead to build failures or incorrect behaviors. We refer to such version-mismatched API usages as **ITAU** (**I**ncompatible **T**hird-party Library **A**PI **U**sage), a common phenomenon in fast-evolving and library-rich ecosystems. [1] For example, as shown in Figure 1, the `RandomForestClassifier` class in `scikit-learn` underwent a breaking change between versions 0.22 and 0.23. In version 0.23 and above, a new parameter `ccp_alpha` was introduced into the class constructor, which did not exist in version 0.22. If a code completion model suggests this parameter without awareness of the target environment's library version, the resulting code may fail to run, triggering a `TypeError`. Such errors exemplify ITAU, where LLM-generated completions are incompatible with the API of the installed library version. This raises an important question:

> *"To what extent are LLM-based code completion systems capable of adapting to the evolving and version-specific nature of third-party libraries?"*

Literature Review. To the best of our knowledge, the capability of LLM-based code completion tools to handle ITAUs in realistic development environments remains underexplored. Existing

[1] **ITAU** refers to a single instance of **Incompatible Third-party Library API Usage**, where a generated API call mismatches the installed library version. **ITAUs** denotes multiple such instances. We use the singular to describe the problem type and the plural to describe concrete occurrences.

studies that consider the sensitivity of LLMs to library changes often limit their evaluation to static code generation tasks or controlled benchmarks. They rely on manually constructed or LLM-fabricated datasets, where code snippets are deliberately curated and labeled for a specific library version [10–14]. While these datasets can reveal whether models are sensitive to simple version-specific signals (e.g., the introduction of deprecated APIs [10]), they do not provide an automated or comprehensive method for assessing ITAUs in real-world code completion scenarios. As a result, they may not fully capture the complexity and dynamism of real-world usage environments. For example, Wang et al. [10] conduct an empirical study showing that LLMs frequently suggest deprecated Python APIs in code completion tasks, underscoring LLMs' difficulty in consistently adopting up-to-date APIs. Similarly, Zheng et al. [12] and Wu et al. [13] examine how well LLMs handle API evolution in the Rust and PyPI ecosystems, respectively, by explicitly specifying package version numbers and analyzing whether the models generate APIs that are compatible with those specific versions. While these efforts provide valuable insights into the general limitations of LLMs in handling API evolution, they fall short in several key aspects when applied to realistic code completion scenarios:

Gap 1: Development-time code completion lacks fully resolved dependency context. Existing benchmarks [10, 12, 13] typically provide the model with explicit dependency specifications needed to generate the target code. However, this assumption does not fully match development-time code completion. Although dependency versions become deterministic after installation, CI, or deployment, code completion usually occurs while developers are writing code. At this stage, an LLM-based assistant typically observes the current editing context, such as the active file, surrounding code, imported symbols, and project-level dependency declarations including `requirements.txt`, `setup.py`, or `pyproject.toml`, rather than the complete resolved dependency tree in the underlying environment. Developers typically rely on dependency specifications declared in the project (often defined by previous contributors), which are commonly expressed as version ranges (e.g., "`pandas ≥1.3, <2.0`") or even left unconstrained [15, 16]. The exact versions are then resolved by package managers at install time. As a result, the dependency information available to the completion model can be incomplete or under-specified, even though the concrete versions are eventually resolved during build or deployment. Moreover, developers are often unaware of the actual versions in use, particularly in the presence of transitive dependencies or automated upgrades [17]. Representative completion workflows, such as GitHub Copilot or chat-based code assistants, often generate suggestions from visible code context and selected project files, rather than invoking a full dependency-resolution process or querying the installed environment before every completion request. Therefore, the key challenge is not that dependency versions are permanently unknowable, but whether LLM-based code completion can generate version-compatible API usages under the dependency information realistically available during code writing.

Gap 2: Lack of fine-grained understanding of compatibility issues. Prior evaluations of LLM-based code completion under API evolution largely adopt a coarse-grained perspective, focusing on high-level changes such as additions, deprecations, and deletions [10, 13]. While these categories capture broad evolutionary trends, they overlook fine-grained breaking changes–such as modified default values, parameter type shifts, or altered return types–that frequently occur in practice and can create signature-level incompatibility risks; some of these risks may later manifest as runtime errors such as `AttributeError` or `TypeError` when the affected code path is executed. These subtle changes, especially prevalent in actively maintained libraries [18], often lead to ITAU when LLM-based code completion suggests API usages misaligned with the installed library versions. Although Zheng et al. [12] explicitly consider fine-grained API changes (e.g. parameter additions and removals) in the Rust ecosystem, their study covers only 15 TPLs and relies on synthetic tasks

rather than real usage sites in actual projects, limiting the generality of their findings. Consequently, existing benchmarks rarely capture such nuanced cases at scale, leaving open whether LLMs can effectively detect and adapt to fine-grained API incompatibilities in realistic development environments.

This Study. To address these gaps, we conduct a study to uncover ITAUs of LLM-based code completion systems in realistic development environments. While the challenges of third-party library evolution apply broadly across software ecosystems, our empirical study, automated framework, and TPL API Knowledge Base construction are explicitly centered on the Python ecosystem. We select Python as our primary target because its dynamic nature, loosely enforced versioning conventions, and rapidly evolving libraries make it inherently susceptible to fine-grained compatibility issues. Our study has two key characteristics: 1) **To address Gap 1**, we collect high-starred GitHub projects with complex and realistic dependency configurations to simulate real-world environments where library versions are implicitly specified and dynamically resolved. 2) **To address Gap 2**, we automatically analyze fine-grained API signature changes across library versions to evaluate real-world compatibility issues arising from TPL API usage in evolving environments. Although runtime execution can eventually expose some ITAUs, relying on post-generation validation is costly and often unsuitable for code completion. It requires constructing executable environments, resolving transitive dependencies, preparing valid inputs, and running project-specific tests, which introduce substantial overhead into an interactive development workflow. Moreover, runtime failures do not always provide fine-grained evidence that the failure is caused by a specific version-incompatible API usage, since errors may also arise from missing inputs, incomplete context, project-specific assumptions, or unrelated runtime conditions. Therefore, we adopt a proactive shift-left strategy grounded in static API-signature analysis. By validating generated API calls against version-specific signatures at completion time, our mechanism can directly identify signature-level compatibility risks before they are integrated into the code.

Building on these insights, we propose the first automated framework to systematically evaluate ITAUs introduced by LLM-based code completion systems in realistic development environments. Our prototype focuses on the Python ecosystem, which is one of the most widely used and rapidly evolving TPL ecosystems. Our framework first constructs a versioned TPL API Knowledge Base by collecting multiple versions of widely used TPLs and extracting their API signatures via abstract syntax tree (AST) analysis. By encoding fine-grained evolution patterns (e.g., parameter additions or removals), this knowledge base reflects the actual dynamics of API changes in real-world projects and facilitates the systematic identification of potential compatibility-breaking usages. In total, our TPL API Knowledge Base consists of 2,178 libraries spanning 129,970 versions, covering 496,369,872 unique API signatures. We then mine high-starred, actively maintained GitHub repositories to locate TPL API call sites that are likely affected by such changes, and transform them into code completion prompts by masking the target API invocation while preserving its surrounding context. Finally, we generate 10,867 code completion tasks and evaluate seven state-of-the-art LLM-based code completion systems by prompting them to complete each task. For evaluation, we design two new metrics (See Section 5.6) that go beyond surface-level textual similarity, including Compatibility Rate (CR) and Breaking Completion Rate (BCR) to assess whether the generated completions conform to the correct API usage under the actual library versions. We further analyze how prompt configurations and API evolution types affect the models' ability to avoid ITAU.

Results and Findings. Our study yields the following findings: **Finding 1:** State-of-the-art LLMs frequently generate ITAUs under realistic project settings. This is largely due to their lack of awareness of version-specific API signatures. **Finding 2:** The way dependency information is expressed in prompts significantly affects the model's ability to avoid compatibility-breaking

completions. Explicit and precise version constraints substantially improve compatibility, whereas the absence of such information leads to a higher rate of errors due to version ambiguity. **Finding 3:** LLMs are also vulnerable to fine-grained API breaking changes. Both parameter updates and return type modifications lead to consistently high incompatibility rates. This indicates a lack of fine-grained sensitivity to semantic shifts introduced by evolving library APIs.

Real-time Detection and Lightweight Repair. We also propose two lightweight approaches to proactively prevent ITAUs during LLM-based code completion. **Real-time Detection** continuously monitors generated TPL API calls and checks their compatibility against version-specific knowledge from the TPL API Knowledge Base. When a potential incompatibility is identified, **Lightweight Repair** augments the prompt with a brief description of the issue and the correct usage pattern from the TPL API Knowledge Base, then instructs the LLM to regenerate only the relevant code segment while preserving surrounding context. This strategy avoids model fine-tuning, adds negligible runtime cost, and effectively steers the model toward producing version-consistent API calls. Our evaluation in Section 7 demonstrates substantial reductions in ITAUs and improvements in compatibility-aware metrics.

Contribution. Our contributions are summarized as follows:

- We conduct the first systematic study on ITAU in LLM-based code completion, highlighting how TPL API evolution causes compatibility-breaking completions.
- We develop an evaluation framework and create a large-scale benchmark comprising 10,867 real-world completion tasks for compatibility testing.
- We propose two lightweight approaches, Real-time Detection and Lightweight Repair, for mitigating ITAU in LLM-based code completion.

Paper Organization. The remainder of this paper is organized as follows. Section 2 introduces background and preliminaries on dependency versioning and fine-grained API evolution. Section 3 presents an overview of the study. Section 4 describes the construction of the versioned TPL API Knowledge Base. Section 5 details the experimental setup, including task construction, evaluation metrics, and model configurations. Section 6 reports and analyzes the empirical results from multiple perspectives. Section 7 introduces two lightweight mitigation techniques and evaluates their effectiveness. Section 8 discusses broader implications, limitations, and directions for future work. Section 9 discusses related work. Finally, Section 10 concludes the paper.

2 preliminaries

2.1 Version Specification and Resolution of Python Dependencies

In modern Python software development, TPLs are widely adopted to accelerate implementation, encapsulate complex functionalities, and promote code reuse. To manage these dependencies, developers typically declare them in configuration files such as `requirements.txt` or `setup.py`. Each declared dependency may be accompanied by version constraints that define which versions of the library are acceptable for the project. According to Python Enhancement Proposals (PEPs) standards [19], dependency version specifications can be broadly categorized into three types:

- **Pinned** (exact) versions (e.g., "`numpy==1.22.0`") require the installation of a specific version and are commonly used in deployment or testing environments to ensure full reproducibility.
- **Constrained** (range-based) versions (e.g., "`pandas≥1.3, <2.0`") allow flexibility within a defined range, balancing stability and compatibility with newer features.
- **Unconstrained** versions (e.g., "`elasticsearch-dsl`") do not specify any version, leaving the package manager to install the latest available release at install time.

While pinned dependencies facilitate reproducible builds, they are relatively uncommon in development settings due to the need for library updates, security patches, or compatibility with

other tools. Instead, range-based versions are widely used in practice [15], with less than 30% of dependencies relying on pinned versions. As a result, the exact versions of libraries are often implicitly determined by the package manager at install time and are frequently overlooked by developers [17]. This inherent uncertainty introduces a fundamental challenge for LLM-based code completion systems: they typically do not have access to the specific versions of libraries installed in the current environment. This disconnection between declared dependencies and actual libraries used gives rise to Gap 1 described in Section 1.

2.2 Fine-Grained Compatibility Issues in API Evolution

APIs in TPLs evolve frequently due to bug fixes, new feature additions, refactoring, and interface redesigns [20]. While such evolution is essential for library maintenance and improvement, it often introduces backward-incompatible changes that pose serious risks to client code. Based on prior taxonomy studies of API evolution [18, 20], we summarize common API evolution patterns in Table 1. These patterns are categorized by the affected code element (e.g., class, function, attribute, parameter), and we further provide their corresponding matching rules and compatibility symptoms. These compatibility-breaking changes may manifest in various forms and lead to runtime errors such as `TypeError`, `AttributeError`, or `ImportError`. The impact is particularly critical in the context of LLM-based code completion systems, where ITAU can be silently introduced into generated code.

Table 1. **Refined API Evolution Patterns, Matching Rules, and Compatibility Symptoms**

Idx	Element	Pattern	Matching Rule	Potential API Usage Symptoms
1	Class	AddClass	$c \notin C,\ c \in C'$	Likely safe, unless reflection or type check is used
2		RemoveClass	$c \in C,\ c \notin C'$	`ImportError`, `NameError`, `AttributeError`
3		ChangeInheritance	$c \in C \cap C',\ base(c) \neq base(c')$	`TypeError`, behavioral inconsistency, MRO issues
4	Function	AddFunction	$f \notin M,\ f \in M'$	Not available in earlier versions, may trigger `AttributeError`
5		RemoveFunction	$f \in M,\ f \notin M'$	`AttributeError`, `NameError`
6		ChangeReturnType	$f \in M \cap M',\ ret(f) \neq ret(f')$	Logical errors, type mismatches, unexpected behavior
7	Attribute	AddAttribute	$a \notin A,\ a \in A'$	`AttributeError`, `NameError` in older versions
8		RemoveAttribute	$a \in A,\ a \notin A'$	`AttributeError` in updated versions
9		ChangeAttributeType	$a \in A \cap A',\ type(a) \neq type(a')$	`TypeError`, downstream logic bugs
10	Parameter	AddParameter	$p \notin P,\ p \in P'$	`TypeError: missing required positional argument`
11		RemoveParameter	$p \in P,\ p \notin P'$	`TypeError: got unexpected keyword argument`
12		ChangeParameter	$p \in P \cap P',\ sig(p) \neq sig(p')$	`TypeError`, `ValueError`, `unexpected keyword`

Notation: c, f, a, p denote class, function, attribute, and parameter respectively. C, M, A, P denote their respective sets in the original version, and C', M', A', P' in the new version. $base(c)$ denotes base classes of c; $ret(f)$ is the return type of f; $sig(p)$ denotes the full parameter signature.

Despite the growing adoption of LLMs for code generation and completion, current evaluation efforts have paid limited attention to ITAU. Prior studies [10, 13] primarily focus on coarse-grained evolution patterns, such as detecting removed APIs (e.g., *RemoveFunction*). For instance, Wang et al. [10] report that GPT-3.5 suggests deprecated functions with an 85% rate, revealing the model's limited awareness of obsolete APIs. However, many real-world ITAUs arise not from complete symbol removal, but from fine-grained changes—such as the addition of a required parameter, a shift in argument order, or a subtle change in return types—that existing benchmarks fail to capture. To bridge this gap, we propose a fine-grained categorization of API evolution patterns and systematically analyze their effects on LLM-based code completion systems.

3 Study Overview

This study aims to evaluate the capability of LLM-based code completion systems in handling ITAUs arising from API evolution in realistic development environments. We chose Python, the

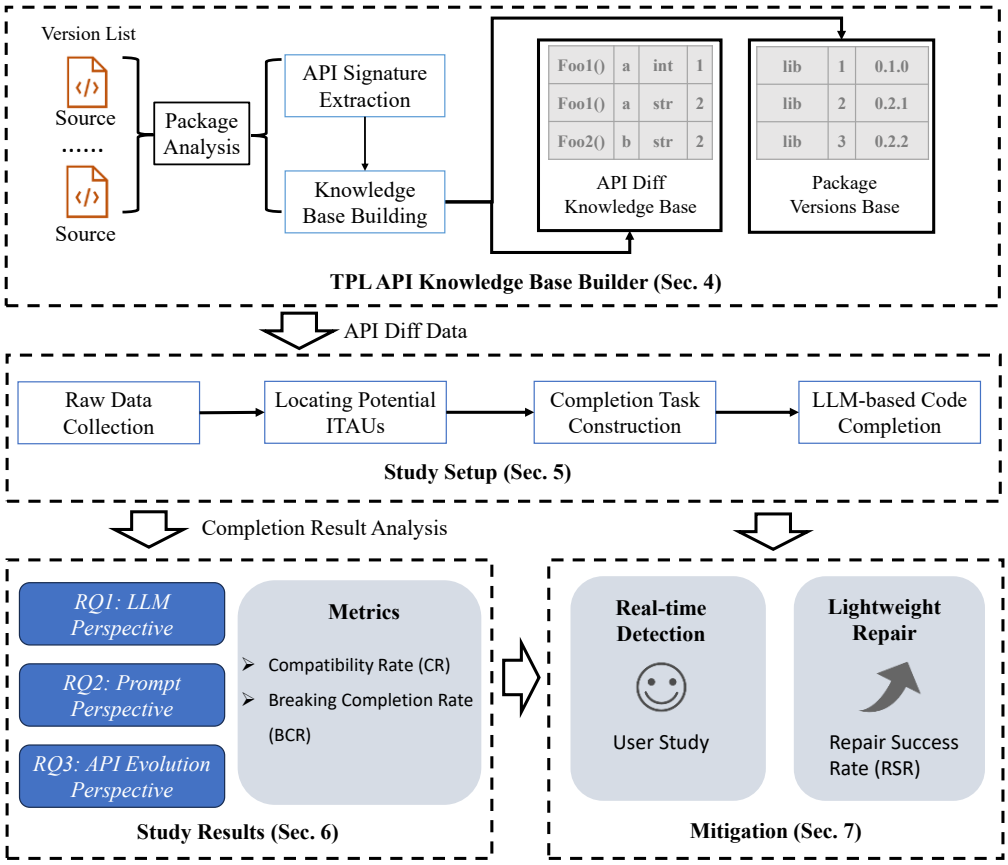


Fig. 2. **Overview of Our Study.** We first construct a versioned TPL API Knowledge Base by extracting API signatures and fine-grained API evolution information from multiple library versions. Based on real-world open-source projects, we then locate potential ITAUs and transform them into realistic code completion tasks. The generated completions from LLMs are evaluated using compatibility-aware metrics from multiple perspectives, followed by two lightweight mitigation techniques for detecting and repairing ITAUs.

most popular programming language in the recent ranking [21]. The overview of our study is presented in Figure 2. It includes four phases:

TPL API Knowledge Base Builder (Section 4). The study begins with the construction of a comprehensive TPL API Knowledge Base, which contains detailed information about the evolution of APIs across different versions of libraries. This knowledge base is built by extracting and analyzing API data from multiple versions of selected TPLs, focusing on identifying significant changes such as modifications to API names, parameters, and return types. The data is organized into an API Diff Knowledge Base, which tracks the differences between API versions and allows for the detection of potential compatibility issues.

Study Setup (Section 5). In this phase, we simulate realistic developer code completion scenarios by leveraging real-world open-source projects from GitHub. We identify the TPL API usages and dependency configurations within these projects, and use the API Diff Data to recognize potential ITAUs that LLM-based code completion systems may encounter. Based on these identified issues,

we construct code completion prompts, which are then processed by various LLMs to generate code completions. The generated completions are automatically annotated, and relevant metrics are calculated to evaluate their quality and correctness.

Study Results (Section 6). The results of the code completions are analyzed to address key research questions: to what extent are LLMs capable of adapting to evolving APIs (RQ1), how does the dependency configuration of the prompt affect performance (RQ2), and how do fine-grained API changes impact code completion accuracy (RQ3). This analysis evaluates the Compatibility Rate (CR) and Breaking Completion Rate (BCR), and further examines how sensitive LLMs are to version-specific changes.

Mitigation (Section 7). Leveraging the TPL API Knowledge Base, we develop two lightweight approaches to mitigate ITAU in LLM-based code completion. Real-time Detection analyzes generated code to identify TPL API calls and checks their compatibility with the resolved library versions. Lightweight Repair leverages the detected incompatibility symptoms and the corresponding version-specific API knowledge to construct informative prompts, guiding the LLM to regenerate completions that align with the correct API usage.

4 TPL API Knowledge Base Builder

To enable version-aware compatibility analysis, we construct a fine-grained TPL API Knowledge Base that captures the evolution of TPL APIs across versions. This section describes how we analyze package versions from PyPI, extract structured API signatures, and organize them into a versioned knowledge base to support downstream code completion analysis.

4.1 API Model Definition

To support fine-grained compatibility analysis across library versions, we define a unified API model that captures essential structural information of TPL APIs. This model enables the systematic detection of breaking changes, including modifications to API names, parameter types, and return types, as summarized in Table 1. We formalize an API signature as a tuple:

$$\text{API}_{\text{signature}} = (\text{Name}, \text{Param}, \text{Return}, \text{Scope}) \tag{1}$$

where:

- **Name** denotes the fully qualified name (FQN) of the API, including module path and class context (e.g., "`sklearn.ensemble.RandomForestClassifier.__init__`");
- **Param** is an ordered list of parameter names (e.g., "`n_estimators`", "`ccp_alpha`"). Parameters with default values are also included;
- **Return** denotes the return type or symbolic return expression if statically known. For example, in "`numpy.mean`", the return is typically "`float`" or "`ndarray`", depending on the input. While Python is dynamically typed, return types can often be approximated using annotations or docstrings.
- **Scope** distinguishes between top-level functions (e.g., "`math.sqrt`"), class methods (e.g., "`pandas.DataFrame.head`"), or class attributes (e.g., "`sklearn.SVC.support_`"). This distinction is essential for detecting changes involving inheritance, visibility, or attribute semantics.

By explicitly tracking these elements across versions, we enable systematic diffing and facilitate the identification of potential compatibility issues in LLM-generated code completion under realistic TPL evolution.

4.2 Package Analysis from PyPI

To support compatibility analysis in realistic development environments, we construct our TPL API Knowledge Base from the source code of widely-used TPLs published on PyPI—the central package repository for the Python ecosystem. We focus on the top 2,500 most downloaded or depended-upon TPLs, covering major libraries across domains such as data science, web development, and machine learning. To ensure fast and reliable access, we maintain a local PyPI mirror and periodically synchronize it with the upstream registry. This design supports incremental updates and allows us to efficiently track library evolution over time without overloading the central infrastructure. Since versioning conventions in Python are often loosely enforced (e.g., pre-release tags, inconsistent increments), we normalize library versions using their official release timestamps to establish a total chronological order. This normalization ensures consistency when comparing adjacent versions and detecting API-level changes.

4.3 API Signature Extraction

While serving as the primary human-facing reference, official API documentation is fundamentally inadequate for automated compatibility analysis. Its heterogeneous formatting and informal natural language across libraries make high-precision parsing infeasible. Furthermore, documentation routinely omits internal or undocumented APIs that remain functionally accessible and are frequently memorized by LLMs. Therefore, rather than relying on brittle documentation, we extract API signatures directly from the source code to capture the precise execution ground truth. To build a structurally accurate and versioned API knowledge base, we extract fine-grained API signatures from the source code of each TPL version. This process begins by parsing all Python source files in each package version using static analysis. For each module, we construct its AST and traverse the tree to identify function and method definitions corresponding to API entry points, including `FunctionDef` and `AsyncFunctionDef` nodes, as well as relevant `ClassDef` contexts. For each discovered API, we extract its FQN (composed of the module path and class hierarchy), its ordered parameter list including both positional and keyword arguments, and its structural scope, such as top-level function, class method, or class attribute, as formalized in Section 4.1.

However, a unique challenge arises from Python's flexible and dynamic import mechanism, which allows APIs to be accessed through multiple alias paths. Due to transitive imports and symbol re-exports across intermediate modules, an API defined deep in the package hierarchy (e.g., `tensorflow.python.ops.nn_ops.relu`) may also be referenced via simplified and user-facing paths such as `tensorflow.nn.relu`. To resolve such aliasing behavior and capture the full set of valid access paths, we augment the extraction process with a static import-flow analysis [22] that traces how symbols are re-imported, re-exported, and propagated across modules. Concretely, we construct a directed graph whose nodes represent modules and whose edges represent import relationships, and then compute the transitive closure to obtain all re-exported alias paths for each API. This enables our knowledge base to record both the canonical definition site and all valid access paths observed within the package.

For example, as shown in Figure 3, the API `relu` is physically defined in the internal module `tensorflow.python.ops.nn_ops`. It is subsequently re-exported in `tensorflow.math.nn` via a `from ... import ...` statement, and further exposed at a higher-level developer-facing namespace `tensorflow.nn.relu`. This propagation yields the following symbolic flow chain:

$$\texttt{tensorflow.python.ops.nn\_ops.relu} \xrightarrow{\texttt{import}} \texttt{tensorflow.math.nn.relu}$$

$$\texttt{tensorflow.math.nn.relu} \xrightarrow{\texttt{import}} \texttt{tensorflow.nn.relu}$$

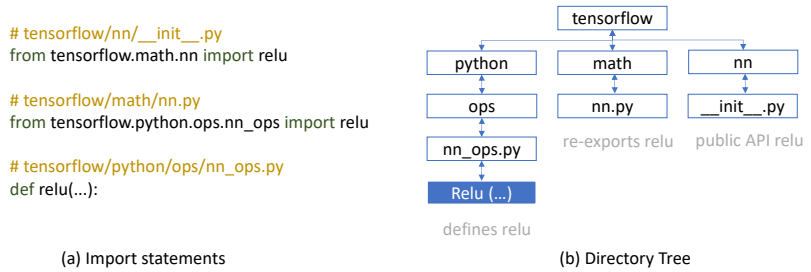


Fig. 3. **An Example of Transitive Imports**

By computing the transitive closure of such chains, we enumerate the full set of effective access points for each API. Incorporating this import-flow information enriches the versioned TPL API knowledge base with both canonical signatures and their exposed alias paths, allowing us to detect breaking changes not only at definition sites but also across the user-facing access namespaces through which developers actually invoke the APIs.

4.4 Knowledge Base Construction

For each package, we aggregate the extracted API signatures across all available versions and perform version-to-version differencing to identify structural API changes. Each API is indexed by its FQN and signature within a version, enabling us to track additions, removals, and signature-level modifications over time. Based on this process, we construct two core artifacts: (1) a *Package Versions Base*, which records the version history of each package and supports version alignment and lookup; and (2) an *API Diff Knowledge Base*, which captures fine-grained compatibility changes between adjacent versions, such as parameter additions, removals, type changes, or attribute modifications. To address namespace refactorings during version-to-version differencing, our import-flow analysis resolves backward-compatible aliases to their canonical signatures. In cases of unaliased API migrations, the framework indexes the change strictly as a structural removal at the obsolete FQN and an addition at the new FQN, without performing cross-module semantic alignment. This structural representation aligns with our evaluation methodology: since invoking an unaliased, outdated FQN inherently triggers an `ImportError` or `AttributeError` in Python, recording such migrations as explicit path removals faithfully captures the resultant compatibility breakage at the client usage site. Currently, our knowledge base covers 2,178 packages spanning 129,970 versions and includes 496,369,872 unique API signatures across all versions. Although we initially targeted the top 2,500 most downloaded or depended-upon TPLs, not all packages could be incorporated: some contain no analyzable APIs (e.g., data-only packages), some have missing or unavailable historical versions on PyPI, and others fail to build due to dependency or compilation issues. These records form the structural foundation for compatibility evaluation, allowing us to trace when and how each API evolves and whether an API usage is valid under a specific library version.

5 Study Setup

5.1 Raw Data Collection

To simulate realistic LLM-based code completion scenarios, we collect a curated set of actively maintained Python projects from GitHub that make extensive use of TPLs. **(1) Repository Selection.** We select non-fork repositories created between January 1, 2025 and June 31, 2025, with data crawled on July 1, 2025. The time span ensures sufficient data collected with no overlap with the training data of many existing code LLMs released before 2025, no matter whether the data is

publicly available or not. To ensure quality and activity, we retain projects with at least 50 stars, 10 forks, and updates within the past 6 months. We further require the presence of explicit dependency declarations in `requirements.txt` or `setup.py`, allowing us to resolve version constraints reliably. **(2) Dependency Coverage.** To guarantee sufficient interaction with TPLs, we exclude minimalistic or single-purpose projects with fewer than five declared dependencies. Projects with richer dependency graphs are more likely to exhibit complex API usage patterns and face compatibility risks arising from version changes.

Algorithm 1: Locating Potential ITAUs in TPL API Usages

Input: Project p, TPL API Knowledge Base $\mathcal{KB}$, Dependency configurations $\mathcal{D}_p$ of project p
Output: List of potential ITAUs in TPL API usages $\mathcal{P}$

Step ① TPL API Usage Detection
Extract all API usages from project p;
for *each API usage u in project p* **do**
    Standardize u to fully qualified name (FQN) in the form *A.B.C.API_Name*;
    if *u is object-oriented* **then**
        Resolve object type and infer the FQN;
    if *u is aliased or imported with shortened name* **then**
        Resolve to FQN by analyzing aliases and imports;
    if *FQN(u) is in $\mathcal{KB}$* **then**
        Store u in $\mathcal{S}_{API}$;
Step ②: Dependency Configuration Validation and API Stability Check
for *each api_i in $\mathcal{S}_{API}$* **do**
    Retrieve the version range v_i of library l_i from $\mathcal{D}_p$;
    Extract the TPL and version information for api_i;
    Compare api_i with the API signatures in $\mathcal{KB}$ for the corresponding version range v_i;
    if *signature matches in all versions within v_i* **then**
        if *signature has evolved outside v_i* **then**
            Mark api_i as potential ITAU and insert into $\mathcal{P}$;
Step ③: Usage Isolation
for *each identified potential ITAU api_i in $\mathcal{P}$* **do**
    Construct an intra-project call graph for api_i;
    if *cross-file dependencies are detected* **then**
        Remove api_i from $\mathcal{P}$;
Return the list of potential ITAUs $\mathcal{P}$.

5.2 Locating Potential ITAUs

This step aims to identify potential ITAUs in project p that may arise from code completion systems and may not be compatible with the target version v of library l. TPLs evolve rapidly, and their APIs may change across versions. LLM-based code completion systems can introduce these risks by generating code that references outdated API usages, which may not be compatible with the version of the TPL currently in use. As shown in Algorithm 1, we identify potential ITAUs in code completions through a three-step process. We will describe these three steps in detail.

Step ①: TPL API Usage Detection. In this step, we aim to extract all TPL API usages in the code of project p. We perform static analysis on project p to identify all API usages (Line 2). These calls may use both FQNs, such as `img.img_to_graph`, and shortened names like `img_to_graph`. To ensure compatibility with our TPL API knowledge base, all usages are standardized to FQNs in the form of `A.B.C.API_Name`, specifying the full path, module, class, and API name (Line 4).

To resolve API calls into FQNs, we first examine the Python source code to identify all function calls and method invocations. We then traverse the ASTs to find any method calls or API references. The next step is resolving the object types in object-oriented code (Lines 5-6), which ensures that method invocations are fully qualified by their class and module. For instance, if a method `loc()` is called on an object `dt`, we ensure that `dt.loc()` is resolved as `pandas.DataFrame.loc()`, assuming that `dt` is an instance of `pandas.DataFrame`. Another common challenge in FQNs arises from the use of aliases for packages, classes, or functions (Lines 7-8). Python's `import-as` feature allows developers to create shorter names for packages or functions (e.g., `import pandas as pd` or `from pandas import DataFrame`), leading to API calls like `pd.DataFrame.loc()` that must be resolved to their full name `pandas.DataFrame.loc()`. We resolve these aliases by analyzing the import statements. For example, when `pandas` is imported as `pd`, we map `pd.DataFrame.loc()` back to `pandas.DataFrame.loc()`. Similarly, when using `from-import`, we resolve shortened API names to their respective full paths. For instance, if `from torch.linalg import lstsq` is used, the API call `lstsq()` is resolved to `torch.linalg.lstsq()`.

After the lightweight static analysis, we obtained the corresponding FQN for each function call. We checked whether the corresponding FQN matched the APIs in our TPL API Knowledge Base, thereby identifying the TPL API usages (Lines 9-10).

② **Dependency Configuration Validation and API Stability Check.** The purpose of Step ② is to validate the compatibility of the extracted TPL API usages by checking the project's dependency configuration for correct version alignment and identifying TPL APIs that have undergone significant changes, which are more likely to cause potential ITAUs in code completion systems.

We begin by retrieving the version range v_i of each library l_i from the project's dependency configuration $\mathcal{D}_p$ (Line 13). This range defines the versions of the library under which the project is expected to function correctly. Next, for each API usage api_i in $\mathcal{S}_{API}$, we extract the version information and compare the extracted API usage with the corresponding API signatures from the TPL API Knowledge Base for the identified version range v_i (Lines 14-15). This comparison ensures that the API usages are compatible with the specified version range v_i, without any changes that could introduce incompatibilities (Line 16). For example, if a library's version range is constrained (e.g., between versions v_1 and v_2), the API signature should remain stable within that range, and no changes should have occurred in this version range that might lead to incompatibility. In terms of version constraints, we handle three types. For two types of version constraints — pinned constraints (exact versions) and constrained constraints (range versions) — we analyze the API signatures within the defined constraint range to ensure they have not changed. For unconstrained dependencies, we do not assume that the project has been explicitly tested against, or semantically guaranteed to support, the latest library release. Instead, we model the default package-manager resolution scenario: when no version constraint is specified, package managers such as `pip` typically resolve the dependency to the latest available version at installation time. Accordingly, we use the project's commit timestamp to identify the latest library version available at that time and treat it as the implied target version for signature-level validation. The resulting labels should therefore be interpreted as compatibility risks under default dependency resolution, rather than as claims of developer-validated compatibility with the latest version. We then repeat the verification process as described for constrained versions. Additionally, we prioritize TPL API usages that have undergone

significant changes outside the specified version range v_i, as these evolved API signatures in other ranges are the most likely sources of potential ITAUs for the project (Lines 17-18).

③ **Usage Isolation.** Our goal is to evaluate LLMs' ability to generate version-compatible API usages rather than their capacity to recover cross-file project structure. However, many API usages in real projects rely on objects or helper functions defined in other files. If such usages are included, an incorrect completion could stem either from genuine API incompatibility or simply from the absence of cross-file context, making it difficult to attribute the error to version-related issues. To avoid this confounding factor, we perform a lightweight intra-file dependency check. Concretely, we construct a lightweight call graph centered around the TPL API usages identified in Step ②. Instead of analyzing the entire project or building full library-level call graphs, we perform a localized traversal of the AST to capture the intra-file call chains in which these TPL APIs are invoked (Line 21). If an API usage depends on definitions originating outside the current file, we mark it as cross-file dependent and exclude it from prompt construction (Lines 22-23). This isolation ensures that subsequent code completion evaluation focuses strictly on the LLM's version-awareness under realistic file-local completion settings, without interference from repository-level context that the model cannot reasonably access.

5.3 Completion Task Construction

In typical code completion scenarios, developers often begin writing a few lines of code and then pause mid-way, relying on LLMs to generate the next portion of code based on the preceding context. Inspired by this practical workflow, we construct our completion tasks using the code segments surrounding real-world API calls that may lead to ITAUs. Once a potential ITAU is located, we extract the entire file-local context preceding the API invocation as the prompt. To evaluate the LLM's behavior under different granularities of code generation, we design two levels of code completion: **(1) API-level Completion.** This setting targets a specific API call. We mask the line containing the invocation of the target TPL API and prompt the model to generate only that line. **(2) Function-level Completion.** In this setting, we provide the function header (i.e., the function definition line containing the name and parameters) and prompt the model to generate the full body of the function. This setting better reflects real-world development scenarios where developers often rely on LLMs to synthesize full function logic based on intent. Figure 4 illustrates an example of prompt construction. The dependency configuration is explicitly provided (e.g., "`scikit-learn<=1.1.3`") alongside the code snippet. The model is required to complete the given code in a manner compatible with the declared version. In total, we construct 10,867 code completion tasks for evaluation.

5.4 LLM-based Code Completion

We leverage multiple LLMs, both open-source and closed-source, to evaluate their performance on the code completion task. The LLMs included in this evaluation are listed in Table 2. These models consist of five state-of-the-art open-source LLMs: CodeGen [3], StarCoder [4], CodeLlama [23], Deepseek-Coder [5], and Deepseek-R1 [24], alongside two closed-source models, GPT-4o [25] and Gemini-2.5-flash [26]. The open-source models evaluated include base models, code models fine-tuned on code data from the base models, and models trained from scratch on code data. Among them, Gemini-2.5-flash is distinct in that it is not limited to prompt-based inference but also supports tool-using functionalities, such as automatically searching API documentation, which enhances its capability to generate more context-aware code completions. To deploy these models, we use a local API server based on vLLM [27], which provides a unified platform for LLM serving and inference. For the closed-source GPT-4o and Gemini-2.5-flash, we queried the models via their official online APIs [28]. It is important to note that the training data for all models in this

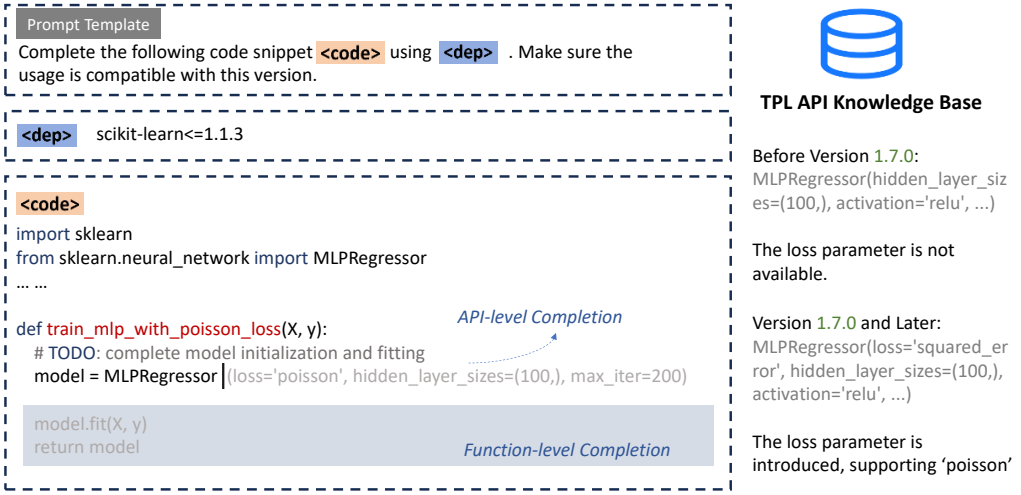


Fig. 4. **Illustration of Prompt Construction for Code Completion.** Each prompt consists of a file-local code snippet, an explicit dependency specification, and a natural-language instruction requiring version-compatible completion. This prompt illustrates two completion granularities: API-level completion and Function-level completion.

evaluation were collected prior to January 1, 2025, ensuring that the data used does not overlap with newer releases or models.

Table 2. **Evaluated Large Language Models**

Base Model	Model	Size	Open-Source
CodeGen [3]	CodeGen-6B [29]	6B	✓
StarCoder [4]	StarCoder2-7B [30]	7B	✓
CodeLlama [23]	CodeLlama-7B-Instruct [31]	7B	✓
Deepseek [24]	Deepseek-Coder-6.7B-Instruct [32]	6.7B	✓
Deepseek [24]	DeepSeek-R1-Distill-Llama-8B [33]	8B	✓
–	GPT-4o [25]	–	✗
–	Gemini-2.5-flash [26]	–	✗

For the completion task, the models were prompted to generate the next line of code in Python, following a specific input provided in the prompt. The models were evaluated using greedy decoding, meaning that the token with the highest probability was selected at each decoding step, ensuring a deterministic output for consistency in results. The output token limit was set to 100 tokens to constrain the completion length. For GPT-4o and Gemini-2.5-flash, the temperature parameter was set to 0 to enforce greedy decoding, ensuring consistent and high-probability completions.

5.5 Completion Result Annotation

For each LLM-generated completion, we determine whether the predicted API usage is compatible with the target TPL version. To facilitate this, we first extract the FQN of the invoked API using the same resolution strategy described in Section 5.2. Beyond the FQN, we analyze the corresponding AST node to extract the list of parameters passed to the API, their associated values, as well as any

relevant return value usage. These elements are critical for assessing the structural correctness of the API call and determining whether it aligns with the expected signature under the declared version. The extracted information forms the basis for compatibility analysis, which we detail in the next section.

5.6 Metrics

To evaluate the compatibility awareness of LLMs in real-world TPL evolution scenarios, we introduce two compatibility-oriented metrics: **Compatibility Rate (CR)** and **Breaking Completion Rate (BCR)**. These metrics specifically assess whether generated API usages conform to the current TPL version, emphasizing signature-level compatibility rather than surface-level textual similarity.

Compatibility Rate (CR). CR measures the proportion of completions that are valid under the actual API signatures of the TPL version used. Given the inherent flexibility in API usage (e.g., optional parameters), we design five matching rules described in Table 3 to determine whether a predicted API usage is compatible with the corresponding API signature.

$$\mathrm{CR} = \frac{1}{|P|} \sum_{p \in P} \mathbb{I}(\mathrm{LLM}(p) \in \mathrm{Compat}) \tag{2}$$

Breaking Completion Rate (BCR). BCR quantifies the proportion of completions that contain ITAUs, i.e., version-specific signature-level incompatibility risks under the current TPL version. We design four breaking rules shown in Table 3. These rules should be interpreted as static compatibility checks against the target API signature, rather than as guarantees of immediate runtime failure. Missing APIs, excessive arguments, and invalid keywords usually correspond to directly executable failures once the affected call is reached, whereas return-value mismatches depend on how the returned object is subsequently consumed.

$$\mathrm{BCR} = \frac{\sum_{p \in P} \mathbb{I}(\mathrm{LLM}(p) \in \mathrm{ITAU})}{\sum_{p \in P} \mathbb{I}(\mathrm{LLM}(p) \in \{\mathrm{Compat}, \mathrm{ITAU}\})} \tag{3}$$

In Equation 2 and Equation 3, we use the following notations:

- P: the set of evaluated prompts.
- $\mathrm{LLM}(p)$: the code completion generated by the LLM for prompt p.
- Compat: the set of completions that are signature-compatible with the current TPL version.
- ITAU: the set of completions that violate the expected version-specific API signature and are therefore treated as signature-level compatibility risks.
- $\mathbb{I}(\cdot)$: the indicator function, which returns 1 if the condition holds and 0 otherwise.

Unlike other similarity-based metrics (e.g., Exact Match) that offer only a coarse-grained view of output fidelity, CR and BCR provide a fine-grained lens on the model's awareness of library versioning. A desirable model should exhibit a high CR and a low BCR. To maintain interpretability, the following three cases are labeled as *Uncertain* where static resolution is intractable: dynamic argument unpacking, reliance on abstract or polymorphic types, and indirect invocations involving reflection or higher-order callbacks. Inclusion of *Uncertain* and *Others* would artificially dilute the breaking rate. Therefore, to ensure reliability, completions labeled as *Uncertain* or *Others*–e.g., those with unresolved signatures or ambiguous calls–are excluded from BCR computations. For return-value-related cases, we apply a stricter criterion: a completion is labeled as a return-value mismatch only when the versioned API signature records an explicit return-structure or return-type change and the generated code immediately uses the returned object in a statically observable way that conflicts with the target-version return contract, such as incompatible tuple unpacking or

downstream attribute/method access. If the return value is merely assigned, passed through, or otherwise used in a way whose runtime impact cannot be inferred statically, we do not group it with directly executable errors such as missing APIs or invalid keywords; instead, we label the case as *Uncertain* and exclude it from BCR.

Table 3. **API Usage Compatibility and Breaking Rules**

Category	Rule Name	Rule Description
Compatible	C1: Method Name Match	Fully-qualified name (FQN) matches the expected API signature.
	C2: Argument Count	Number of provided arguments ≤ specified in signature; all required arguments are present; optional arguments may be omitted.
	C3: Argument Sequence	All positional arguments appear before keyword arguments; the order of keyword arguments can be arbitrary.
	C4: Keyword Validity	All provided keyword arguments exist in the signature; keywords may be reordered.
	C5: Return Value Match	The return value's type and structure are compatible with those specified in the API signature.
Breaking	B1: Nonexistent/Removed API	Calling API that is unavailable in the current TPL version
	B2: Argument Count Error	Provided arguments > allowed, or missing required arguments.
	B3: Invalid Keyword	Usage of keywords not present in the current signature.
	B4: Return Value Mismatch	The generated code immediately consumes the returned object in a way that is statically inconsistent with the target-version return type or return structure recorded in the API signature.

6 Results

Our study aims to answer the following research questions (RQs):

- **RQ1 (LLM Perspective): To what extent do state-of-the-art LLM-based code completion systems generate ITAUs under different completion granularities?**
  We evaluate the compatibility performance of various LLMs by measuring the CR and BCR of their generated API usages under actual TPL versions extracted from open-source repositories, in both API-level and function-level code completion settings. For this comprehensive baseline evaluation, the reported metrics are calculated by aggregating the results across the *Pinned*, *Range*, and *Unconstrained* dependency configurations, excluding the *Omitted* setting.
- **RQ2 (Prompt Perspective): How do different dependency configurations affect the model's ability to avoid ITAU errors?**
  We conduct experiments at the API-level completion granularity under varying prompt settings—including *Omitted* (no dependency information provided), *Pinned*, *Constrained*, and *Unconstrained*—to assess how explicitly providing version information influences the model's compatibility awareness and its tendency to generate incompatible API usages.
- **RQ3 (API Evolution Perspective): How does the type of API evolution affect the likelihood of ITAU in LLM-generated completions?**
  We analyze the impact of different API evolution types—including changes in *names*, *parameters*, and *return types*—on the frequency of incompatible completions.

6.1 RQ1: LLM Perspective

We compare models on both API-level and Function-level completion using our compatibility-aware metrics: **CR** (higher is better) and **BCR** (lower is better). The results in Table 4 provide a comprehensive view of how different LLMs handle compatibility across varying granularities of code completion. Overall, the compatibility of current LLMs in code completion remains limited. At the API-level, CR values are consistently low, below 33% for all models, while BCR values remain high, ranging from 45.99% to 68.50%. The situation worsens at the Function-level, where CR drops

below 7% for every model and BCR often exceeds 80%. These results indicate that, despite recent advances, generating version-consistent code remains a major challenge.

Among all evaluated models, GPT-4o achieves the best relative performance with 31.78% CR and 45.99% BCR at the API-level, and 5.91% CR with 73.35% BCR at the Function-level. Although still far from ideal, these results demonstrate GPT-4o's stronger ability to produce more version-consistent completions. By contrast, open-source models such as CodeGen and StarCoder lag significantly behind, with CR values as low as 5.86% and 12.02% at the API-level, and dropping to 1.04% and 5.38% at the Function-level. CodeLlama and DeepSeek-Coder show moderate API-level performance (around 28% CR) but still suffer steep declines at the Function-level. Although Gemini-2.5-flash integrates tool-using features like automatic API documentation search, its performance at the Function-level still lags behind GPT-4o, suggesting that while tool usage may improve contextual understanding, it is not sufficient to generate version-specific API usage code or fully address the challenges of maintaining compatibility across evolving library versions.

The results indicate that the Function-level setting is significantly more challenging than the API-level setting. For every model, CR drops sharply and BCR values rise substantially compared to the API-level setting. This performance degradation reflects the greater difficulty of generating longer, context-dependent code segments, where models are more likely to overlook version-specific constraints and thus produce compatibility-breaking completions.

To provide a comprehensive view, we also quantified the proportion of completions excluded from the BCR computation. In our experiments, these results account for a moderate fraction of the total generations, ranging from 30.88% (GPT-4o) to 36.24% (CodeGen). This excluded subset predominantly consists of severe syntax errors, unresolved parsing cases, and model refusals, which are categorized as *Others*. A negligible fraction is due to dynamic usages, represented by the *Uncertain* category.

Table 4. **RQ1: Comparison of CR and BCR at API and Function-level Completion**

Metric	CodeGen	StarCoder	CodeLlama	DeepSeek-Coder	DeepSeek-R1	GPT-4o	Gemini
API-level (CR)	5.857%	12.017%	28.628%	28.708%	28.628%	31.778%	32.845%
API-level (BCR)	68.495%	56.790%	56.039%	47.911%	51.435%	45.994%	56.211%
Function-level (CR)	1.037%	5.383%	6.743%	3.527%	1.867%	5.913%	3.216%
Function-level (BCR)	96.575%	83.007%	79.062%	91.169%	93.750%	73.349%	81.576%

Note. Higher CR is better, while lower BCR is better. Cells are color-coded for readability: for CR (green), darker shades indicate better performance, whereas for BCR (red), lighter shades indicate better performance.

Finding 1: Current LLMs exhibit limited compatibility awareness in code completion, with substantial room for improvement even for state-of-the-art models. Even the best-performing model (GPT-4o) achieves only 31.78% CR at the API level and 5.91% CR at the Function level, with corresponding BCR values of 45.99% and 73.35%.

6.2 RQ2: Prompt Perspective

We evaluate how different dependency configurations in prompts influence compatibility-aware performance across LLMs. As shown in Table 5, providing more precise dependency information improves model performance. For example, GPT-4o attains its highest CR of 32.99% and lowest BCR of 50.34% under the pinned setting, compared to only 23.24% CR and 66.25% BCR when dependency information is omitted. A similar trend holds for CodeLlama and DeepSeek-Coder, where pinned prompts reduce BCR by more than 10 percentage points relative to omission. One might argue

that exposing exact resolved versions via advanced IDE tooling could completely resolve ITAU. However, it is critical to note that even under this ideal *Pinned* setting, the absolute error rates remain remarkably high. The fact that top-tier models like GPT-4o still generate compatibility-breaking completions over 50% of the time underscores that merely providing explicit version constraints is insufficient to eliminate ITAUs. This exposes a fundamental deficit in LLMs' intrinsic version-awareness: they struggle to dynamically align their generated API signatures with explicit version boundaries, regardless of the provided context. In contrast, *Omitted* dependencies yield the weakest results, with low CR (below 10% for CodeGen and StarCoder) and high BCR (above 65% for several models). This suggests that without dependency hints, models rely on pretraining priors, which often produce outdated or incompatible API calls. This further demonstrates that LLMs lack the reasoning capacity to translate broad environmental constraints into precise, version-compatible API usage logic.

The *Constrained* and *Unconstrained* settings yield intermediate performance. While they improve compatibility relative to omission, they remain less effective than pinned prompts. For instance, DeepSeek-Coder exhibits a higher BCR under constrained prompts (48.88%) than under pinned dependencies (45.91%). Our evaluation reveals a counterintuitive phenomenon: in certain cases, the "Unconstrained" setting marginally outperformed the "Constrained" setting. We interpret this as a plausible alignment effect rather than a demonstrated causal mechanism. One possible explanation is that LLMs tend to favor frequent and mainstream API signatures observed during pretraining, which often correspond to relatively recent library versions. In an unconstrained dependency setting, package managers commonly resolve dependencies to recent available versions at installation time. As a result, model-generated mainstream API usages may occasionally align with the default resolved versions more often than expected, even without explicit version reasoning. Under explicit range constraints, the model may face a harder contextual reasoning problem: it must condition on version boundaries and sometimes generate older or less frequent API signatures. Our results suggest that current models do not consistently perform this alignment. Therefore, the relatively better compatibility performance under the unconstrained setting should not be read as evidence of superior version reasoning. Rather, it is a plausible symptom of reliance on distributional API priors, and it highlights the need for stronger evidence before attributing these differences to a specific causal mechanism.

Table 5. **RQ2: Impact of Different Dependency Configurations on Compatibility**

Metric	CodeGen	StarCoder	CodeLlama	DeepSeek-Coder	DeepSeek-R1	GPT-4o	Gemini
Omitted (CR)	7.441%	8.199%	17.641%	18.073%	9.064%	23.235%	25.244%
Omitted (BCR)	66.547%	68.404%	58.172%	66.132%	58.421%	66.248%	65.678%
Pinned (CR)	9.337%	13.978%	29.088%	29.162%	15.616%	32.987%	29.855%
Pinned (BCR)	63.598%	54.946%	42.961%	45.911%	47.452%	50.338%	59.357%
Constrained (CR)	5.289%	10.032%	28.502%	27.695%	12.677%	31.490%	29.295%
Constrained (BCR)	70.661%	61.020%	43.293%	48.882%	49.800%	52.133%	58.284%
Unconstrained (CR)	7.634%	12.470%	28.537%	27.858%	13.509%	27.538%	26.005%
Unconstrained (BCR)	61.336%	59.794%	45.161%	48.712%	53.762%	60.897%	60.216%

Note. Higher CR is better, while lower BCR is better. Cells are color-coded for readability: for CR (green), darker shades indicate better performance, whereas for BCR (red), lighter shades indicate better performance.

Finding 2: While providing precise dependency information improves relative performance compared to omission, it does not eliminate ITAUs, with BCR remaining above 50% for top-tier models like GPT-4o. This highlights that ITAU is not merely a transient engineering or tooling issue; it also reflects a persistent limitation in current LLMs' ability to use version-specific API information.

6.3 RQ3: API Evolution Perspective

Table 6 examines how different types of API evolution affect the likelihood of ITAU. Overall, the results show that *name changes* are the most disruptive form of evolution for most models, while *parameter changes* and *return type changes* also introduce substantial compatibility risks.

Name changes yield the lowest CR and some of the highest BCR across nearly all models. For example, CR under name changes is only 2.46% for CodeGen and 11.89% for StarCoder, with BCR reaching 80.65% and 76.33%, respectively. Even stronger models struggle: GPT-4o achieves 26.31% CR with a BCR of 66.29%, while Gemini-2.5-flash reaches 20.08% CR and 69.57% BCR. These results suggest that LLMs tend to reproduce identifiers memorized during pretraining and have difficulty adapting when API names are renamed; once the identifier shifts, the learned associations often become incompatible with the current library version. However, *parameter changes* and *return type changes* also introduce substantial challenges. For *parameter changes*, CR remains very low across models (3.57% for CodeGen, 10.00% for StarCoder, and 24.81% for GPT-4o), while BCR frequently exceeds 55%. *Return type changes* exhibit a similar pattern: although CR is somewhat higher (e.g., 32.74% for CodeLlama and 30.97% for StarCoder), BCR remains substantial, reaching 72.45% for GPT-4o.

These observations highlight that, while name changes represent the most severe signature/path-level evolution type in our benchmark, fine-grained changes in parameters and return types expose a deeper limitation of current LLM-based code completion systems. Such changes alter API contracts beyond surface identifiers; as a result, generated completions may appear syntactically correct yet still violate version-specific signature expectations. Return-value mismatches should be interpreted especially carefully: in Python, a changed return type does not necessarily cause an immediate failure unless the downstream code consumes the returned object incompatibly. We therefore treat return-value results as statically observable compatibility risks tied to explicit downstream use, rather than as guaranteed runtime failures. These subtle yet impactful changes are both frequent and realistic in modern software ecosystems, and they expose critical weaknesses in current LLM-based code completion systems.

Table 6. **RQ3: Impact of API Evolution Types on Compatibility**

Metric	CodeGen	StarCoder	CodeLlama	DeepSeek-Coder	DeepSeek-R1	GPT-4o	Gemini
Names (CR)	2.459%	11.885%	17.213%	15.574%	9.836%	26.311%	20.082%
Names (BCR)	80.645%	76.331%	66.667%	68.852%	64.179%	66.286%	69.565%
Parameters (CR)	3.571%	10.000%	20.952%	22.143%	13.333%	24.810%	36.667%
Parameters (BCR)	63.415%	75.041%	52.941%	53.266%	57.895%	68.586%	52.322%
Return Types (CR)	13.274%	30.973%	32.743%	26.549%	19.469%	23.894%	42.478%
Return Types (BCR)	59.459%	68.456%	44.776%	56.522%	53.191%	72.449%	50.515%

Note. Higher CR is better, while lower BCR is better. Cells are color-coded for readability: for CR (green), darker shades indicate better performance, whereas for BCR (red), lighter shades indicate better performance.

Finding 3: API name changes constitute the most disruptive form of evolution for LLM-based code completion. Across models, name changes yield extremely low compatibility, with CR dropping to 2.46%–26.31% and BCR rising to 64.18%–80.65%. Meanwhile, fine-grained changes in parameters and return types also introduce substantial risks: CR remains below 25% for most models under parameter changes and below 33% under return type changes, while BCR frequently exceeds 50%. These results indicate that both surface-level and semantic-level API evolution pose significant challenges to current LLMs.

7 Mitigation

For ITAU, mitigation cannot rely on retraining LLMs with perfectly up-to-date corpora, as library evolution is continuous and fine-grained API changes occur frequently. Instead, effective solutions should focus on model inference and prompt-level interventions, enabling real-time detection and correction of incompatibilities without modifying the underlying model. Based on the empirical findings, we develop lightweight approaches to mitigate Incompatible Third-party API Usages (ITAU) in LLM-based code completion. To clearly distinguish between human-dimension assistance and model-dimension improvement, we explicitly separate our mitigation strategies into two independent yet complementary paths: (1) Real-time Detection (Human-facing, Section 7.1), positioned as a safety net during the coding phase to intercept generated errors on the fly, thereby enhancing human development efficiency and code reliability; and (2) Lightweight Repair (LLM-facing, Section 7.2), positioned as an automated, closed-loop repair mechanism designed to objectively and end-to-end improve the LLM's final code output.

7.1 Real-time Detection

7.1.1 Approach. We propose the Real-time Detection approach to continuously monitor API calls as they are generated. When a TPL API invocation is detected (detailed in Section 5.2), the system queries the TPL API Knowledge Base to retrieve the canonical, version-specific API signature. This lookup is highly efficient, with most queries completing in under 0.03 ms, introducing negligible overhead to interactive code completion. The generated call is then validated against this signature using the compatibility rules in Table 3. If any breaking rule is violated, the detector flags the call as potentially incompatible and records it for subsequent repair. In parallel, regardless of whether an incompatibility is detected, the retrieved API signature is presented to the developer as an authoritative reference, enabling them to verify the correctness of their usage in real time. This dual process both provides immediate compatibility validation and delivers just-in-time guidance, reducing the likelihood of ITAU instances propagating into the final codebase.

7.1.2 Evaluation. To assess the practical usefulness of the Real-time Detection approach from a developer's perspective, we conducted a user study focusing on two aspects: (i) the perceived value of providing version-specific API signatures during development, and (ii) the perceived importance of detecting ITAUs in real time. Eight participants—graduate students or professional developers with at least two years of Python experience and regular exposure to TPL-heavy projects—were recruited through internal mailing lists. The study protocol was reviewed and approved by the Institutional Review Board (IRB) of Zhejiang University (ZJU). Participants took part in the study either on-site or remotely under supervised screen sharing. Each participant worked in a customized VS Code environment where our detection module was integrated into the LLM-based completion interface. The system automatically monitored every TPL API invocation and, depending on compatibility status, displayed either unobtrusive version-specific API signatures or highlighted warnings paired with corrected signatures. Participants were asked to complete three development

tasks extracted from real-world open-source repositories, each designed to represent a distinct category of breaking API evolution. Table 7 summarizes the tasks, which cover parameter changes (`requests`), return type changes (`pandas`), and API name changes (`scikit-learn`), with code contexts ranging from 10 to 150 LOC to reflect low-, medium-, and high-complexity scenarios. After completing the tasks, participants filled out a questionnaire consisting of the questions listed in Table 8, rating the perceived usefulness of version-specific API signatures and real-time detection of ITAU on a 5-point Likert scale (1 = not useful, 5 = very useful).

Table 7. **User Study Tasks Covering Different API Evolution Types**

Task	Difficulty	TPL	Evolution Type	Task Description
T1	Low	`requests`	Parameter change	Update deprecated SSL-related parameter in `requests.get()` (10–20 LOC).
T2	Medium	`pandas`	Return type change	Adjust downstream logic after Series-to-DataFrame API change (50–80 LOC).
T3	High	`scikit-learn`	API name change	Fix broken imports and calls due to moved/renamed classes (100–150 LOC).

Table 8. **Likert-scale Questions Organized by Evaluation Objective**

Evaluation Objective	ID	Question
(i) Value of version-specific API signatures	Q1	The version-specific API signatures reduced my need to consult external documentation.
	Q2	The API signatures increased my confidence in correct API usage.
	Q3	The real-time API signature reference improved my overall development efficiency.
(ii) Importance of real-time incompatibility detection	Q4	Real-time detection of incompatible API usages is important for avoiding hard-to-debug runtime errors.
	Q5	Real-time incompatibility detection should be an essential feature in LLM-based code completion tools.

Across all tasks, participants consistently reported that Real-time Detection offered clear practical benefits. For objective (i), the perceived usefulness of version-specific API signatures received an average score of 4.63. For objective (ii), the perceived importance of real-time incompatibility detection received an even higher average score of 4.75. Participants reported that having access to authoritative API signatures not only reduced lookup time in external documentation but also increased confidence in the correctness of API usage. They also highlighted that real-time incompatibility detection prevented subtle runtime errors that are often hard to trace in large projects. These results suggest that Real-time Detection can meaningfully improve both development efficiency and code reliability with negligible runtime overhead.

Finding 4: Real-time Detection introduces negligible overhead (API signature lookup $<$ 0.03 ms) while delivering strong practical benefits. In a user study, participants rated version-specific API signatures at 4.63/5 and real-time incompatibility detection at 4.75/5, indicating that the approach effectively improves developer confidence and prevents ITAU during code completion.

7.2 Lightweight Repair

7.2.1 Approach. Building on the detection results, the Lightweight Repair approach intervenes at the prompt level to steer the LLM toward producing compatible API calls without model retraining. For each detected incompatibility, the repair approach augments the original prompt with two key elements: (i) a concise description of the incompatibility and its cause under the current TPL version, and (ii) the correct version-specific API signature from the TPL API Knowledge Base. The

LLM is instructed to regenerate only the minimal code span surrounding the API call, preserving the surrounding context. This signature injection approach incurs negligible runtime overhead and is compatible with any LLM capable of accepting natural language and code prompts.

7.2.2 Evaluation. To evaluate the effectiveness of the Lightweight Repair approach, we used the subset of prompts from RQ1 where the LLM-generated completions contained ITAUs. We focused only on API-level completions in which the incompatibility could be directly linked to a specific TPL API invocation. We report the **Repair Success Rate (RSR)**, defined as the proportion of incompatible completions that are successfully transformed into compatible ones after repair according compatibility rules in Table 3. In addition to RSR, we also report the pre- and post-repair CR/BCR to evaluate the end-to-end impact of Lightweight Repair. RSR is a local metric computed only over the initially incompatible completions, measuring how many detected ITAU cases can be converted into compatible API usages. In contrast, CR and BCR are global metrics computed using the same definitions as Section 5.6: CR is computed over all evaluated prompts, while BCR is computed over completions that can be confidently classified as either *Compat* or *ITAU*. For the post-repair results, we replace each initially incompatible completion with its repaired output and then recompute CR and BCR on the same evaluation set. Therefore, RSR reflects local repair effectiveness, whereas CR/BCR before and after repair reflect the overall change in final completion quality.

Table 9 summarizes the evaluation results. Across all tested LLMs, Lightweight Repair achieved RSR values ranging from 2.96% to 46.34%. Among them, Deepseek-Coder attained the highest repair success rate of 46.34%, demonstrating strong adaptability to prompt-level signature injection. Note that we apply repair only to the cases where each model actually produced ITAU instances, so the repaired subsets differ across models; therefore, we do not compare the absolute RSR values between models as a measure of relative performance. From the global perspective, Lightweight Repair improves CR for all evaluated models, with the largest gains observed for Gemini (+16.54 percentage points), DeepSeek-Coder (+12.24 points), and GPT-4o (+8.80 points). BCR also decreases for most models, especially DeepSeek-Coder (from 52.09% to 25.71%), CodeLlama (from 56.04% to 26.92%), and Gemini (from 56.21% to 33.58%). These results indicate that the repair mechanism improves not only the local repair subset measured by RSR, but also the overall compatibility of final completions under the original CR/BCR metrics.

Models achieving higher repair success rates, such as DeepSeek-Coder, exhibit a strong proficiency in resolving localized structural mappings, which correspond to fine-grained categories like `ChangeParameter` and `ChangeReturnType`. Upon receiving the version-specific API signature from our knowledge base, these models reliably execute precise syntactic substitutions. For example, considering a `ChangeParameter` instance in the `numpy` library (version `1.24.0`) where a parameter signature was modified (e.g., the obsolete `normed` argument in `numpy.histogram` being replaced by `density`), DeepSeek-Coder successfully leveraged the injected prompt context to update the target keyword argument while strictly preserving the surrounding file-local logic. Incompatibilities associated with absolute structural removals, corresponding to categories such as `RemoveFunction` and `RemoveClass`, remain highly challenging and lead to persistent repair failures across models. A representative failure case occurs within the `RemoveFunction` category in the `scikit-learn` library, where the function `sklearn.datasets.load_boston()` was completely deprecated and removed. Resolving this specific ITAU requires significantly more than a straightforward signature substitution; it necessitates a transition to an entirely different implementation mechanism. In such scenarios, even when provided with explicit context regarding the absence of the obsolete API, models severely struggle to synthesize the multi-step logic required for the new paradigm. Instead, they frequently exhibit hallucinations by either inventing fictitious backward-compatibility

wrappers or continually regenerating the deprecated usage. This highlights a critical structural limitation: while lightweight repair prompts effectively guide LLMs through minor signature alignments for categories like `ChangeParameter`, they are often insufficient for bridging the severe semantic gaps introduced by `RemoveFunction` or `RemoveClass` instances. These results suggest that the proposed approach can guide LLMs toward more version-consistent API usages, although its effectiveness still depends on the type of signature-level incompatibility and the model's ability to use the injected signature context.

Table 9. **Repair Success Rate (RSR) of Lightweight Repair across Evaluated LLMs.**

Metric	CodeGen	StarCoder	CodeLlama	Deepseek-Coder	DeepSeek-R1	GPT-4o	Gemini
RSR (%)	24.127%	5.032%	38.776%	46.341%	2.963%	23.565%	39.845%
CR (Before)	5.857%	12.017%	28.628%	28.708%	28.628%	31.778%	32.85%
CR (After)	8.927%	12.814%	37.332%	40.947%	29.062%	40.574%	49.394%
BCR (Before)	68.495%	56.790%	56.039%	52.089%	48.565%	45.995%	56.21%
BCR (After)	51.983%	53.922%	26.922%	25.705%	49.911%	41.274%	33.582%

Finding 5: Lightweight Repair mitigates ITAUs by injecting concise error explanations and version-specific API signatures into the prompt, without requiring model retraining. Across evaluated LLMs, the approach successfully repairs 2.96%–46.34% of incompatible completions, with the highest repair success rate achieved by DeepSeek-Coder (46.34%). These results demonstrate that prompt-level signature injection can effectively recover a substantial fraction of ITAU cases, providing a lightweight yet practical mitigation strategy.

8 discussion

8.1 Implications

For LLM Researchers. Our findings suggest that incompatibility in LLM-based code completion is not merely a form of hallucination, but reflects a deeper structural issue: version boundaries are not encoded in the model's internal representation. LLMs tend to conflate APIs across library releases into a unified semantic space, leading to systematic over-generalization when the same symbol evolves across versions. This indicates that future LLM designs should incorporate mechanisms that explicitly represent the temporal dimension of API evolution, rather than relying solely on static pretraining knowledge. Beyond improving prompt- or decoding-level strategies, researchers may explore model editing techniques—such as parameter-level edits, memory-based knowledge injection, or contrastive learning over API diffs—to update outdated API knowledge directly within the model. Finally, the strong impact of fine-grained signature changes highlights the need for evaluation benchmarks that reflect realistic, version-varying environments, enabling researchers to measure a model's compatibility robustness rather than relying solely on static correctness metrics.

For Code Developers. From a practical perspective, ITAU can silently introduce runtime errors or subtle functional bugs. Our study indicates that these risks can be significantly mitigated by integrating automatic compatibility validation into development workflows. In particular, real-time detection of version-specific mismatches—combined with inline warnings—allows developers to catch erroneous completions before code execution. Moreover, developers often do not need full API documentation to avoid incompatibilities; instead, they benefit most from version-differential signature knowledge that describes only the API changes relevant to their installed environment.

This insight helps explain the effectiveness of lightweight repair strategies and suggests practical opportunities for tooling: IDE extensions or autonomous agents can continuously monitor dependency drift, retrieve up-to-date API signatures, and suggest compatible replacements as libraries evolve.

8.2 Threats to Validity

Internal Validity. The primary threat to internal validity lies in the limitations of our compatibility assessment methodology. Although our TPL Knowledge Base effectively captures API evolution across versions, Python's dynamic typing and flexible usage patterns make it inherently difficult to determine whether an API usage precisely matches its signature. For instance, argument unpacking, dynamic imports, or runtime reflection can obscure static analysis results. To mitigate this threat, we define a set of structured matching rules to guide the compatibility evaluation process. Moreover, as discussed in Section 5.6, we exclude ambiguous or unresolved cases labeled as *Uncertain* or *Others* from metric computation to reduce noise and improve annotation reliability. Another limitation is that our validation focuses on version-specific API signature compatibility rather than full dynamic execution or complete semantic equivalence. Dynamic execution could provide stronger end-to-end evidence of actual runtime failures, but it requires constructing precise environments, resolving transitive dependencies, preparing valid inputs, and executing project-specific tests, which is difficult to scale to large-scale and real-time code completion scenarios. Meanwhile, a purely signature-level oracle cannot fully reason about complex backward-compatible API evolutions, such as broadening accepted parameter types (e.g., from `int` to `Union[int, float]`), changing return types to more abstract but behaviorally compatible interfaces (e.g., from `list` to `Iterable`), or introducing overloads and wrapper aliases. These cases may preserve semantic compatibility even when the surface signature changes, and thus may be conservatively flagged as potential risks by static analysis. Therefore, our reported ITAU rates should be interpreted as version-specific signature-level compatibility risks rather than exhaustive runtime failure rates or exact semantic incompatibility rates. To better understand the precision of this oracle, we manually inspected 100 cases uniformly sampled from the completions labeled as ITAU by our static analysis. During inspection, we recorded the corresponding evolution category, including removed or renamed FQNs, argument-count changes, invalid keywords, and return-value mismatches. A case was treated as a likely false positive only when the generated usage could plausibly remain backward-compatible under the target version despite the surface signature difference, for example because the change added an optional argument with a default value, broadened an accepted type, preserved the old access path through an alias or wrapper, or changed the return type to a behaviorally compatible abstraction whose downstream use remained valid. Under these criteria, 5 of the 100 sampled cases involved potentially backward-compatible signature changes. This manual check does not replace execution-based validation, but it suggests that such false positives exist and should be acknowledged while not dominating the observed trends. In future work, we plan to incorporate semantic constraint solving, type hierarchy reasoning, and lightweight execution feedback to reduce such false positives.

External Validity. Our evaluation focuses on a selected subset of LLMs due to computational resource constraints. As a result, the findings may not fully generalize to all code generation models available in the ecosystem. We believe, however, that the evaluated models are representative of current state-of-the-art LLMs and cover diverse families of architectures and training settings. Additionally, our study exclusively targets Python, which, while being one of the most widely used languages in software development, may not capture API evolution and compatibility patterns present in statically typed or compiled languages. Regarding generalizability, the core methodology of our framework (extracting API signatures across versions, masking file-local contexts for prompts,

and LLMs' version-awareness evaluation) is language-agnostic and can be generalized to other ecosystems like Java (Maven) or JavaScript (npm). However, certain structural designs depend heavily on Python-specific semantics. For instance, our dependency resolution relies on parsing `requirements.txt` and `setup.py`, and our static analysis is deeply tailored to Python's dynamic import aliasing and AST structures. Adapting this framework to statically typed languages would require replacing these Python-specific components with ecosystem-native dependency solvers and build-time type checkers. Extending our framework to support additional programming languages is our future work to enhance the generality of our findings.

9 Related work

9.1 API Evolution

API evolution has long been a central topic in software engineering research, with early studies primarily focusing on the Java ecosystem and systematically characterizing the types, causes, and impacts of API changes. Des Rivières [34] provided one of the earliest classification taxonomies for Java API changes and discussed how APIs can evolve while preserving backward compatibility for client code. From a refactoring perspective, Dig et al. [20] analyzed API evolution in Java libraries and found that nearly 80% of breaking changes originate from API refactorings, highlighting the structural roots of incompatibility. Subsequent large-scale empirical studies further quantified the prevalence and impact of API evolution in practice. For example, Xavier et al. [35] studied 317 real-world Java libraries and reported that up to 27.99% of API changes are breaking, affecting approximately 2.54% of client projects. Dietrich et al. [36] investigated binary compatibility issues in OSGi-based systems and demonstrated that even minor API changes can trigger severe runtime failures in modular environments. Beyond Java, prior work has examined API evolution in specific ecosystems and application domains. Wu et al. [37] analyzed the frequency and impact of API evolution patterns in the Apache and Eclipse ecosystems, while Li et al. [38] investigated the evolution of Web service APIs and its implications for client applications. In the mobile domain, McDonnell et al. [39] and Wei et al. [40] studied API stability and evolution in the Android framework, revealing strong correlations between API changes and application defects. More recently, Zhang et al. [41] showed that API evolution is a fundamental root cause of a wide range of defects in deep learning programs, underscoring its growing importance in data-intensive and intelligent software systems.

Recently, a growing body of research has examined API changes in Python libraries, motivated by Python's widespread adoption in scientific computing, data analysis, and machine learning ecosystems. Existing studies have investigated Python API evolution from multiple complementary perspectives, including the characterization of evolution patterns [42–44], automated detection of compatibility-breaking changes [18, 20], and empirical evaluation of their impact on downstream projects [45–47]. These works collectively reveal that Python APIs evolve rapidly and frequently introduce breaking changes, often due to refactorings, signature modifications, or semantic shifts in core abstractions. For instance, Zhang et al. [44] conducted the first large-scale and fine-grained empirical study of Python library evolution, systematically categorizing breaking changes and showing that they are pervasive across popular libraries. Building on this line of work, Du et al. [18] proposed AexPy, an API-model-based framework that leverages static analysis and historical information to detect compatibility-breaking changes across library versions, significantly outperforming prior rule-based and heuristic approaches. Complementary studies [18, 48] further demonstrated that such breaking changes can propagate widely through dependency chains, forcing downstream projects to lag behind or undertake costly migration efforts.

Despite these advances, prior research has largely focused on the effects of API evolution on *human-written* client code and traditional software maintenance workflows. The impact of API evolution on *automatically generated code*, particularly code produced by LLMs, remains underexplored. Unlike human developers, LLMs rely heavily on pretraining distributions and limited prompt context, making them especially vulnerable to both surface-level changes (e.g., API renaming) and semantic-level changes (e.g., parameter and return-type evolution). In this paper, we aim to fill this gap by systematically investigating the prevalence of incompatible API usages in LLM-generated code completions and proposing lightweight detection and remediation strategies tailored for real-world development environments.

9.2 Code Completion Evaluation

Code completion is a fundamental feature in modern Integrated Development Environments (IDEs) and code editors. With the advancement of natural language processing, deep learning models have been increasingly adopted for code completion tasks [49, 50], due to the structural similarity between code completion and token-level language prediction. To evaluate the capabilities of LLM-based code completion tools [9, 51, 52], prior studies have proposed systematic benchmarks and evaluation methodologies. For example, Ciniselli et al. [53] conducted large-scale empirical analyses of state-of-the-art Transformer models across multiple completion granularities, ranging from single-token prediction to full code snippets. Zeng et al. [54] showed that pretrained models significantly outperform non-pretrained approaches on program understanding tasks, while also observing that no single pretrained model consistently dominates across all tasks. Xu et al. [55] evaluated multiple LLMs on the HumanEval benchmark, while Ding et al. [56] identified common error patterns in Python code completion, such as undefined names and unused variables. Using real-world autocomplete logs spanning 12 programming languages, Izadi et al. [57] further reported that function name prediction errors are particularly prevalent, accounting for approximately 23% of all token-level errors. In addition, Liu et al. [58] proposed EvaluPlus, which assesses the functional correctness of LLM-generated code using enhanced test suites.

In addition to accuracy concerns, LLM-based code generation methods face a range of challenges including security vulnerabilities and hallucinations. One particularly critical yet underexplored issue is the incompatibility caused by TPL API evolution. These version-sensitive changes may trigger runtime errors or inconsistent behaviors. Due to the subtle and version-specific nature of these errors, existing evaluation benchmarks fall short in assessing LLMs' awareness of real-world dependencies. To address this gap, we propose a novel evaluation framework that systematically benchmarks LLM-based code completion systems against compatibility-breaking TPL API usages, offering deeper insights into their robustness in evolving software ecosystems.

10 Conclusion and Future Work

This work investigates Incompatible Third-party Library API Usage (ITAU) in LLM-based code completion, with a focus on the challenges introduced by dynamic dependency constraints and fine-grained API evolution. Our evaluation shows that all tested LLMs consistently fail to maintain version consistency under these conditions. To mitigate this, we introduce two lightweight strategies—Real-time Detection and Lightweight Repair—that identify such inconsistencies and regenerate only the affected call sites. These findings offer practical insights into enhancing compatibility in LLM-driven software development and point to promising directions for integrating library evolution awareness into future code generation systems. The source code and datasets are available at https://github.com/VerComFix/VerComFix.

Future work can extend this study beyond signature-level compatibility to semantic API changes, where behavioral differences arise even without explicit signature modifications. Incorporating

semantic information from documentation, changelogs, or historical usage patterns may enable more fine-grained detection of such subtle incompatibilities. Additionally, although this work focuses on Python, the underlying methodology is language-agnostic. Extending our framework to other programming languages and ecosystems would enable a broader understanding of how LLM-based code completion systems cope with API evolution under diverse language features and library ecosystems.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable and insightful feedback. This work is supported by the Xinmiao Project of ZheJiang Provincial Applied Basic Research Program (Grant No. 2026XMZD032), the Zhejiang Pioneer (Jianbing) Project (2025C01198), the National Science Foundation of China (No. 62372398, No. 72342025), Zhejiang Provincial Science Foundation of China (No. LQK26F020002), and the Fundamental Research Funds for the Central Universities (No. 226-2025-00067). Lingfeng Bao is the corresponding author, and is also affiliated with the Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

References

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

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

[3] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

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

[5] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[6] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 858–868.

[7] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, 2014, pp. 419–428.

[8] G. Pinna, D. Ravalico, L. Rovito, L. Manzoni, and A. De Lorenzo, "Enhancing large language models-based code generation by leveraging genetic improvement," in *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 2024, pp. 108–124.

[9] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[10] C. Wang, K. Huang, J. Zhang, Y. Feng, L. Zhang, Y. Liu, and X. Peng, "Llms meet library evolution: Evaluating deprecated api usage in llm-based code completion," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 781–781.

[11] D. Zheng, Y. Wang, E. Shi, R. Zhang, Y. Ma, H. Zhang, and Z. Zheng, "Humanevo: An evolution-aware benchmark for more realistic evaluation of repository-level code generation," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 764–764.

[12] L. Liang, J. Gong, M. Liu, C. Wang, G. Ou, Y. Wang, X. Peng, and Z. Zheng, "Rustevoˆ 2: An evolving benchmark for api evolution in llm-based rust code generation," *arXiv preprint arXiv:2503.16922*, 2025.

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

[14] N. Islah, J. Gehring, D. Misra, E. Muller, I. Rish, T. Y. Zhuo, and M. Caccia, "Gitchameleon: Unmasking the version-switching capabilities of code generation models," *arXiv preprint arXiv:2411.05830*, 2024.

[15] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 102–112.

[16] C. Wang, R. Wu, H. Song, J. Shu, and G. Li, "smartpip: A smart approach to resolving python dependency conflict issues," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[17] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX security symposium (USENIX Security 19)*, 2019, pp. 995–1010.
[18] X. Du and J. Ma, "Aexpy: Detecting api breaking changes in python packages," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 470–481.
[19] "Pep specifications," https://peps.python.org/, 2023, accessed: 2023-08-24.
[20] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring," *Journal of software maintenance and evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.
[21] O. Software, "Most popular programming languages," 2023, accessed: 2025-07-19. [Online]. Available: https://www.orientsoftware.com/blog/most-popular-programming-languages/
[22] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of jupyter notebooks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1622–1633.
[23] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.
[24] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.
[25] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.
[26] Google, "Google gemini," https://gemini.google.com/, accessed: 2025-09-15.
[27] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.
[28] Sourcegraph, "Code search and an ai assistant with the context of the code graph," 2024, accessed: 2024-07-31. [Online]. Available: https://sourcegraph.com
[29] Salesforce, "Codegen-6b-nl," 2025, accessed: 2025-07-28. [Online]. Available: https://huggingface.co/Salesforce/codegen-6B-nl
[30] BigCode, "Starcoder2: A family of open llms for code," https://huggingface.co/bigcode/starcoder2-7b, 2024, accessed: 2024-07-29.
[31] CodeLlama, "Codellama-7b-instruct-hf," 2025, accessed: 2025-07-28. [Online]. Available: https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf
[32] DeepSeek-AI, "Deepseek-coder 6.7b instruct," https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct, 2024, hugging Face model card; accessed 2025-09-06.
[33] D. AI, "Deepseek-r1-distill-llama-8b," 2025, accessed: 2025-07-28. [Online]. Available: https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Llama-8B
[34] J. Des Rivières, "Evolving java-based apis," http://wiki.eclipse.org/Evolving_Java-based_APIs, Oct. 2007, accessed: 2007.
[35] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 138–147.
[36] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 64–73.
[37] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2366–2412, 2016.
[38] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How does web service api evolution affect clients?" in *2013 IEEE 20th International Conference on Web Services*. IEEE, 2013, pp. 300–307.
[39] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 70–79.
[40] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 226–237.
[41] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 129–140.
[42] H. Quan, J. Wang, B. Li, X. Du, K. Liu, and L. Li, "Characterizing python method evolution with pymevol: An essential step towards enabling reliable software systems," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2022, pp. 81–86.
[43] Z. Zhang, Y. Yang, X. Xia, D. Lo, X. Ren, and J. Grundy, "Unveiling the mystery of api evolution in deep learning frameworks: a case study of tensorflow 2," in *2021 IEEE/ACM 43rd International Conference on Software Engineering:*

Software Engineering in Practice (ICSE-SEIP). IEEE, 2021, pp. 238–247.

[44] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do python framework apis evolve? an exploratory study," in *2020 ieee 27th international conference on software analysis, evolution and reengineering (saner)*. IEEE, 2020, pp. 81–92.

[45] S. A. Haryono, F. Thung, D. Lo, J. Lawall, and L. Jiang, "Mlcatchup: Automated update of deprecated machine-learning apis in python," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 584–588.

[46] A. Vadlamani, R. Kalicheti, and S. Chimalakonda, "Apiscanner-towards automated detection of deprecated apis in python libraries," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021, pp. 5–8.

[47] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library apis are (not) handled," in *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 233–244.

[48] H. Wang, S. Liu, L. Zhang, and C. Xu, "Automatically resolving dependency-conflict building failures via behavior-consistent loosening of library version constraints," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 198–210.

[49] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1433–1443.

[50] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

[51] S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh, "Improving llm code generation with grammar augmentation," *CoRR*, 2024.

[52] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the capabilities of llms for code-change-related tasks," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 6, pp. 1–36, 2025.

[53] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4818–4837, 2021.

[54] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 39–51.

[55] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, 2022, pp. 1–10.

[56] H. Ding, V. Kumar, Y. Tian, Z. Wang, R. Kwiatkowski, X. Li, M. K. Ramanathan, B. Ray, P. Bhatia, S. Sengupta *et al.*, "A static evaluation of code completion by large language models," *arXiv preprint arXiv:2306.03203*, 2023.

[57] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, "Language models for code completion: A practical evaluation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[58] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.