

PyPI API Drift: A Release-Level Record of Public API Changes in 1,872 Popular Python Libraries, with a Checker and a Benchmark for Version-Aware Code Generation

Pushkar Devanahalli Gopalakrishna

pushkar87dg@gmail.com

Independent Researcher

Research Article

Keywords: Python, PyPI, API evolution, breaking changes, semantic versioning, code generation, large language models, benchmark, MCP

Posted Date: October 27th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11503453/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: The authors declare no competing interests.

PyPI API Drift: A Release-Level Record of Public API Changes in 1,872 Popular Python Libraries, with a Checker and a Benchmark for Version-Aware Code Generation

Pushkar Devanahalli Gopalakrishna
Independent Researcher, USA

Abstract

Ask a code assistant to append a row to a pandas DataFrame and it may write `df.append(row)`, a method removed in pandas 2.0. Models trained on code from many releases cannot tell which API exists in the version a user has installed, and benchmarks of this problem are small and frozen. We present *PyPI API Drift*, a release-level record of the public APIs of the 1,872 most-downloaded Python libraries: up to 80 recent final releases each (55,565 releases, 2021–October 2026), with signatures, 15,258,652 release-to-release changes and 210,535 deduplicated breaking changes. Extraction is static and import-free, reads only the needed bytes of each wheel, separates names that were truly removed from names that merely left the documented API, and marks with confidence labels and a coverage layer where static analysis cannot see. It is a high-precision, low-recall record: in AI-agent runtime audits of libraries never used for tuning, 91.9% of high-confidence breaking changes were confirmed, while recall over all runtime-visible changes is limited (16 of 99). On top of it we release an offline checker, lookup tools with an MCP server for coding agents, and VersionProbe, a benchmark of 10,958 valid/invalid snippet pairs. In a preregistered code-repair experiment, repeated across 4 models from one model family, API history from the dataset raised the share of API-compatible, intent-preserving migrations (under our evaluation protocol) to 68.5%–69.9% for every model, from 7.0%–56.6% unaided and 46.2%–62.2% with the new release’s documentation. 13.4% of patch releases contain at least one breaking change.

1 Introduction

Python libraries evolve quickly. Functions are deprecated and removed, parameters are renamed or made keyword-only, and modules are reorganized. For developers this causes upgrade failures; for LLM-based code assistants it causes a subtler failure: models trained on a mix of old and new code confidently produce calls that are valid for some version of a library but not the one installed. Typical examples include `DataFrame.append` (removed in pandas 2.0), `read_csv(error_bad_lines=...)`, `Retry(method_whitelist=...)` (removed in urllib3 2.0) and `from flask import escape` (removed

in Flask 3.0).

Recent benchmarks measure this weakness [1–4], but they are small, hand-curated or frozen at a point in time. What is missing is *infrastructure*: a large, accurate, continuously refreshable record of which public API existed in which release of which library. This paper contributes:

- **A dataset** of public-API snapshots and changes for 1,872 libraries and 55,565 releases, with signature-level detail, deduplication across inheritance and re-exports, and confidence labels;
- **An extraction method** that is cheap (range reads of only the Python members of wheels), import-free, and conservative about what counts as a removal;
- **A validation protocol** of pipeline-isolated runtime audits (precision, benchmark labels, recall, real upgrade failures, comparison with two existing tools), reported in full with the patterns that shaped the method;
- **Tools**: `apidrift-check`, a version-aware checker for Python code (e.g., LLM output); `apidrift-lookup` and an MCP server that give coding agents the same record; and **VersionProbe**, a paired benchmark of 10,958 valid/invalid snippets;
- **Evidence of use**: a preregistered code-repair experiment, repeated across 4 models from one family;
- **Findings** on how often semantic-versioning expectations are violated in the Python ecosystem.

2 Related Work

Version-aware code generation. VersiCode [1] covers over 300 libraries and 2,000 versions up to 2023, labelling API changes at the name level. GitChameleon [2] and GitChameleon 2.0 [3] provide 116 and 328 executable, version-conditioned problems. Wang et al. [4] study deprecated API usage by LLMs over 145 API mappings in eight libraries. Our work is complementary: it supplies signature-level API records across far more libraries and recent releases, from which such benchmarks can be generated and refreshed.

API evolution and breaking changes. AexPy [6] detects breaking changes in Python packages by combining static and dynamic analysis; it imports each release. griffe [5] loads Python APIs statically and reports breaking changes between

two versions, for documentation and release checks. Section 7 compares both with this dataset on the same release pairs. In the Java ecosystem, Raemaekers et al. [7] reported that about one third of Maven releases introduce a breaking change, regardless of version increment, while Ochoa et al. [8] found higher semantic-versioning compliance in a later replication. We provide a large, open, release-level dataset for Python, together with measured precision.

3 Data Collection

3.1 Selection

We take the top 2,000 projects by recent downloads from the public top-PyPI-packages list (October 2026), and for each the final (non-pre-release, non-yanked) releases uploaded since 2021-01-01 that ship a wheel, capped at the 80 most recent releases per project. For every release we pick one wheel, preferring a pure-Python wheel, then a CPython `manylinux x86_64` wheel for the newest interpreter. This yields 55,839 releases of 1,872 projects (83.5% pure-Python wheels). Of these, 55,565 were analysed: 274 releases of packages with more than 8,000 modules were skipped, and 5 placeholder releases (version 0.0.0 uploaded after the project’s real releases) were removed (Section 5.4). 1,869 projects have at least one analysed release.

3.2 Range reads

Wheels are ZIP files. We read the end-of-central-directory record and the central directory with HTTP range requests, select `.py` and `.pyi` members (excluding `tests`, `test`, `conftest` and `metadata`), and fetch only members whose CRC changed since the previous release of the same project. We measured the transfer by replaying this fetch logic over the full histories of 300 randomly chosen projects (8,149 releases): it downloads 8.5% of the wheels’ bytes (95% bootstrap CI 5.7–14.7% over projects), which extrapolates to about 26 GB (17–44 GB) for the whole dataset instead of the 300 GB needed to download every wheel. The saving comes from wheels of 1 MB or more (95.3% of the bytes, of which 5.3% is fetched); small pure-Python wheels consist mostly of the files we need anyway (74.1% fetched). The median release costs 81 KB and two requests, and only 15.2% of `.py/.pyi` members have to be fetched at all, the rest being unchanged since the previous release.

3.3 Static extraction

Each module is parsed with Python’s `ast`; nothing is imported or executed. Per module we record definitions (functions with full signatures, classes with members, including instance attributes set through `self`), imports, star imports, a statically evaluable `__all__`, and properties of a module-level `__getattr__`. Several Python idioms required explicit handling, most of them discovered through the audit rounds of Section 5 and the corrections of Section 5.4:

conditional and `try/except` imports (fallback branches never override the primary binding); PEP 484 positional-only parameters written as `__x` (normalized to PEP 570 form); `@overload` declarations (merged, and treated as typing-level evidence); runtime source taking precedence over `.pyi` stubs, with stub-only class members ignored; classes created by calls (`TypedDict(...)`, `namedtuple(...)`) and names rebound to instances; typing Protocols (members excluded); namespaces populated dynamically (`globals()`, `exec`, `protobuf` builders), which are treated as unknown; and classes whose constructor a decorator generates or replaces (`@dataclass`, `attrs`, factories such as `Matplotlib’s make_norm_from_scale`), whose inherited constructor signature is discarded.

The public API of a release is the set of paths reachable from public modules (no underscore-prefixed segment) through `__all__` or, when absent, public definitions and (in packages) re-exports, resolved through relative imports, star imports and inheritance from classes defined in the same distribution.

3.4 Diffing and the accessibility test

Consecutive releases (by version order) are diffed per path. Signature changes are classified into `param_removed`, `param_renamed` (positional match), `required_param_added`, `became_keyword_only`, `became_positional_only`, `default_removed`, `vararg_removed`, `varkw_removed` and `positional_reordered` (breaking), and several non-breaking types. The key design decision concerns removals: a path that disappears from the public API is recorded as `removed` only if an *accessibility test* on the new release proves it unreachable by attribute access. If it is still reachable (e.g., dropped from `__all__`, now an alias of an external object, served lazily) or reachability cannot be decided statically (compiled modules, dynamic namespaces, class `__getattr__` for data attributes), it is recorded once as non-breaking `left_public_api` and carried forward. Changes are deduplicated across subclass and re-export copies (representative) and labelled `high` or `medium` confidence; `medium` covers members missing from classes with bases in other packages, stub-derived symbols, and removed `**kwargs`.

4 Dataset

Table 1 lists the released tables. The data covers 55,565 successfully analysed releases (274 releases of packages with more than 8,000 modules were skipped), 7,209,790 parsed modules (166 unparseable), and 799,705 distinct signatures. Every release covers at most the 80 most recent final releases of its project, so it is not a complete history of every release since 2021.

Coverage layer. Absence from the dataset is not proof that a name does not exist. A separate pass

Table 1: Released tables (Parquet).

File	Rows	Grain
releases	55,839	release
api_history	15,634,142	path $\times$ state $\times$ release range
changes	15,258,652	change between releases
signatures	799,705	distinct signature
module_coverage	289,735	module $\times$ coverage $\times$ range

Table 2: Deduplicated breaking changes by type (high confidence).

Change type	Count
Symbol removed	113,573
Parameter removed	21,068
Became keyword-only	14,443
Required parameter added	11,430
Positional parameter renamed	11,077
Default removed	6,584
Became positional-only	1,408
Positional parameters reordered	1,054
Kind changed	386
Total	181,023

over every analysed wheel records, per module and release range, whether the module was parsed from Python source, only a `.pyi` stub ships, the module serves names dynamically (an open module `__getattr__`, `globals()` injection, a runtime `__all__`), or it binds names from compiled extension modules (289,735 rows in `module_coverage`). Each `api_history` row carries the resulting `coverage_status`: 79.1% static, 0.8% compiled boundary, 17.7% dynamic module and 2.3% stub only. The coverage pass uses a deliberately broad notion of “dynamic” (any `globals()` use), so it errs toward warning.

Integrity. `validate_dataset.py` runs 51 checks on every build: key uniqueness, version order, interval consistency (no overlaps, indices matching versions, no interval spanning unanalysed releases), every signature hash resolving, change rows matching their releases, label domains, benchmark provenance (each pair traces to a breaking change between its two releases), and a recomputation of every headline count; the released build passes all of them. A regression suite of 158 tests covers each false-positive pattern found by the audits.

5 Validation

Because the dataset is meant to be used as a reference, we measured its precision with *pipeline-isolated, AI-agent-conducted runtime audits*: the auditor never saw the pipeline or the dataset, only the claims to check. The independence comes from that isolation, not from a separate human researcher; disputed cases were not adjudicated by humans. Claims were drawn by stratified random sampling (at most two per

Table 3: Pipeline-isolated runtime audits of claimed breaking changes.

Round	Sample	Outcome
1	60 claims, 32 libs	37% false positives
2	60 claims, 32 new libs	25% false positives
3 (final)	100 claims, 53 new libs	high: 88.8% [80.0–94.0%] medium: 45.0%
4 (v1.2)	160 high, 134 new libs	91.9% [86.6–95.2%]

Brackets: Wilson 95% interval.

library; round 3 additionally stratified by confidence tier and excluded every library used in rounds 1–2). Each claim was verified by an AI agent (an LLM with shell access, not a human annotator) that had no access to the pipeline or the dataset: it installed both releases in isolated virtual environments, imported the module, walked attributes and compared `inspect.signature`, and fell back to reading released sources when installation was impossible (18 of 100 claims in round 3). These audits are therefore a systematic, reproducible spot check rather than human-annotated ground truth. Table 3 summarizes the results.

Rounds 1 and 2 revealed recurring false-positive patterns: names still reachable after leaving `__all__`, re-exports of external objects, lazy module attributes, `try/except` fallbacks, the `__x` convention, stub/runtime disagreement, typing overloads, dynamically generated namespaces, and synthetic classes. Each pattern was fixed generally (Section 3) and checked against the earlier audit samples before the final run. Round 3 used libraries never seen during tuning, so its figures estimate the precision of the released data: 88.8% for high-confidence claims (Wilson 95% CI 80.0–94.0%, $n=80$) and 45.0% for medium (25.8–65.8%, $n=20$). Precision differs by category: 98.1% for removals ($n=53$, CI 90.1–99.7%) against 79.2% for signature changes ($n=24$, CI 59.5–90.8%); all 3 sampled kind changes were wrong. Two residual patterns in round 3, removed `*args` that were only forwarded to a base class and classes rebound as aliases (`Name = OtherClass`), were then reclassified as medium. Against the released tables, 94.5% of the remaining high-confidence round-3 claims are confirmed ($n=73$). Because that rule was chosen after seeing round 3, we report the as-audited 88.8% as the out-of-sample estimate. All samples, verdicts, evidence and the protocol are published with the dataset.

Fresh audit of the re-extracted release (round 4). Because version 1.2 changed the extraction rules (Section 5.4), we audited it again on 160 new high-confidence claims from 134 libraries never used in any earlier audit or study, stratified into claims that are new in v1.2 (80, drawn from 8,274) and claims unchanged since v1.1 (80); 125 were checked at runtime and the rest from source. Precision was 91.9% overall (CI 86.6–95.2%); 90.0% for the new claims (CI 81.5–94.8%) and 93.8% for the unchanged ones (CI 86.2–97.3%); weighted by the size of each stratum, 93.6% (CI 88.5–98.7%). Removals scored 91.4% ($n=128$) and signature changes 93.8% ($n=32$). By verification method, claims checked at runtime

were 92.0% correct (115/125) and claims checked from source 91.4% (32/35); in round 3 the split was 97.1% (67/69) at runtime and 36.4% (4/11) from source, the latter being packages that could not be installed. The 13 false positives came from names served by `__getattr__` or compiled star imports, circular wildcard imports, stub-only declarations, inheritance from another distribution and `**kwargs` that still accept a renamed keyword. Auditors also flagged confirmed removals that are not real public API (test modules and sample code shipped in wheels).

5.1 Benchmark labels

Two samples of VersionProbe pairs were verified by resolving every name and keyword in both releases at runtime (never executing the snippet). The first (200 pairs, one per library) found 172 correct (86.0%, CI 80.5–90.1%) and prompted two exclusions (instance-only attributes, type-checking-only names). The benchmark was then rebuilt, and a second, *fresh* sample of 400 pairs was drawn from it, restricted to 265 libraries never used in any earlier audit, study or fix. All 400 pairs were decided, and 366 were correct: 91.5% (CI 88.4–93.9%). By type: removals 90.8% (n=292), removed parameters 96.3% (n=54), renamed parameters 90.7% (n=54). The “invalid after” half held in 97.3% of pairs and the “valid before” half in 94.0%: almost every error is a name that looks present statically but does not exist at runtime (deleted with `del` after use, defined only in a Python-2 or `if False:` branch, present only in a stub, or a class attribute mistaken for a module attribute), plus one bogus “0.0.0” release sorted before real ones. The `del` pattern is fixed in the extractor and takes effect at the next refresh.

5.2 Recall and comparison with existing tools

An agent introspected both releases of 30 random release pairs at runtime and listed 99 breaking changes. Within the dataset’s definition of public API, recall was 87.5% (14/16, CI 64.0–96.5%). Over all runtime-visible changes it was 59.3% excluding one outlier (`trio`, whose 72 wrapper methods were generated at runtime) and 16.2% including it; most other misses are helper names imported into modules for internal use. Table 4 scores `griffe` and `AexPy` on the same pairs with the same matching rule. `AexPy`, which imports every release, finds most runtime-generated changes; within the shared public-API definition the three methods are close. The runtime reference set itself is incomplete (some changes found by both this dataset and `AexPy` are absent from it), so we report agreement rather than precision for the tools’ extra claims.

5.3 Real upgrade failures

Pilot. An agent ran the test suites of 12 real projects (9 breaking dependency upgrades, 3 controls) before and after the upgrade and found 12 API-caused failures. `apidrift-check` produced 7 findings, all real, with none on the old environments or controls, but predicted only 4 of the failures. Fixing

Table 4: Breaking changes detected on the 30 recall pairs (runtime reference set).

	PyPI API Drift	griffe	AexPy
Within API-Drift definition	14/16	12/16	13/16
Runtime-visible, w/o trio	16/27	12/27	20/27
All runtime-visible	16/99	12/99	90/99
Pairs analysed	30	30	29
Median s / pair	2.1	1.3	62.7

Time per pair: extracting both releases (range reads; `griffe` downloads wheels; `AexPy` installs both releases). One `AexPy` pair ran out of memory.

the two extraction gaps it exposed raised this to 8 of 12 (in-sample).

Unseen upgrades. We then froze the checker and data and had two agents, who never ran the checker, collect 35 new cases (25 expected breaks, 10 controls, 27 libraries, including compiled packages and lazily loaded APIs). 32 were scored (the other library versions were never analysed, and the checker reports that instead of guessing). Of 68 findings, 63 pointed at real incompatible code (92.6%, CI 83.9–96.8%), including code that the tests never exercise. Per failure, it predicted 9 of 18 API-caused test failures (50.0%, CI 29.0–71.0%); per upgrade, 8 of 18 truly breaking upgrades were predicted completely, and it raised findings on 3 of 14 upgrades with no API failure (two of them real but untested incompatibilities). Misses fall into: names removed behind module `__getattr__` deprecation shims (`NumPy`, `NetworkX`), names only ever served through such shims, a module re-exporting an external package (`lxml.html.clean`), static misreads, dunder names (skipped by design), a call through a subclass, and one checker bug (modules removed in later releases were not mapped to their distribution, e.g. `pkg_resources` in `setuptools` 82), now fixed. With the fix and the data corrections of Section 5.4, it predicts 14 of 18 (in-sample); 93 of its 106 findings match what the agents observed at runtime, and most of the rest are uses of modules removed in the new release (e.g. `pandas.util.testing` in `pandas` 2.0) in test code the agents did not list.

5.4 Corrections found by the second review

The integrity checks, tests and audits above found four defects in the first release, all fixed in this one: 77 renames of positional-only parameters were labelled breaking; 369 breaking constructor-signature events concerned classes whose constructor a decorator builds (79,234 such class definitions were found), whose signature had been inherited from a base class (now medium, and the signature dropped from 9,373 history rows); API-history intervals of two projects extended over releases too large to analyse (342,126 rows); and module-to-distribution mapping used only the latest release.

Version 1.2. The unseen-upgrade misses traced to three over-cautious rules, now fixed: any use of `globals()`, even a read in `__dir__`, marked a module as dynamic and hid every removal in it (`NumPy` 2.0’s `bool`); a runtime-built

`__all__` blocked attribute checks although it only affects star exports (SciPy’s `sims`); and star-import chains were abandoned instead of followed (NetworkX). With these and the `del` fix, the full dataset was re-extracted: high-confidence breaking changes rose from 175,221 to 181,950 (6,743 more removals), 82 VersionProbe pairs whose change no longer holds were dropped (including 4 that the fresh audit had judged wrong), and the checker now also checks `super()` calls against library base classes. Re-scored against the released tables, the round-3 audit keeps 94.5% precision ($n=73$); these after-fix figures are in-sample.

Version 1.3 (third review). Two further defects were fixed. 5 placeholder releases (`0.0.0` uploaded after real releases, e.g. `pytest-xdist`) had been sorted first and diffed against unrelated builds; they are removed (1,761 change events and 6 VersionProbe pairs dropped), and selection and integrity checks now exclude them. Second, names served by module `__getattr__` from module-level containers were invisible; deprecation shims and lazy-import maps are now followed when their filling is static, counting a name as served only where the lookup returns an object and as removed only where `__getattr__` explicitly rejects it (NumPy’s `__former_attrs__`). This changed 56 libraries: public symbol paths rose by 27,360. Following them exposed a dating problem: a name that becomes statically undecidable is carried forward, and its removal is recorded only where it becomes provable, which can be later than the real removal (NumPy dropped `float` in 1.24.0 and rejects it explicitly from 1.24.2). The 1,123 distinct high-confidence removals with this history are now medium confidence, with the earliest possible release in their detail, and 66 VersionProbe pairs built on them were dropped. High-confidence deduplicated breaking changes went from 181,935 to 181,023. Re-scored on v1.3, the unseen-upgrade study is unchanged (14 of 18): the NumPy case it was meant to catch is exactly such an undecidable release.

5.5 A model baseline, with and without API history

To show what the dataset adds for code models we built *VersionProbe-verified*: the 492 pairs (984 items, 375 libraries) whose labels were confirmed at runtime by the two benchmark audits. Items were shuffled and asked one at a time (“is this snippet valid in exactly version v ?”). A Claude model (`claude-opus-5-5`), run as an agent allowed only to read the questions and write answers, scored 58.4% item accuracy (CI 55.3–61.5%) and 26.6% pair accuracy from its own knowledge, against 50% and 0% for a constant answer. It accepted most valid snippets (72.8%) but recognized only 44.1% of invalid ones, and was weaker on changes released after its training cutoff (52.1%, $n=96$, against 59.1%). Given API history retrieved from the dataset for *every* name and keyword in the snippet (the releases in which each existed or was accepted; no label and no marker of what changed), it scored 100.0% and 100.0% pair accuracy (409 items fixed, 0 broken). This is essentially an oracle condition: the evidence answers the

Table 5: Code repair on 143 real breaking changes (one model).

	Closed book	Docs	API Drift
API-compatible	58.7%	69.9%	80.4%
Intent (AI judge)	93.0%	86.0%	88.1%
Both	56.6%	60.1%	68.5%
removed parameter	52.3%	56.8%	70.5%
renamed parameter	57.1%	73.8%	76.2%
moved/removed name	59.7%	52.6%	61.4%

question directly, and the items are ones where the dataset is known to be right. It shows only that the lookup works (the answer is in the record when retrieved), not that models benefit in general; the repair experiment below tests that. Prompts, raw answers and the scoring script are released.

5.6 Code repair: does the dataset help a model fix real breaks?

The classification baseline above shows what the history contains, not whether it helps a model write code. We therefore ran a preregistered repair experiment (task list hashed before any model output). 143 real breaking changes from 109 libraries (removals with a recorded new location, renamed and removed parameters; 7 more tasks were excluded because the original snippet already resolved and bound in the new release) were given to the same model as code written for the old release, with the instruction to update it for the new release while keeping its purpose. Three conditions: closed book; *documentation*, an API reference page for the imported module(s) generated with griffe from the new release’s wheel; and *PyPI API Drift*, the dataset’s history, recorded change and new signatures for the snippet’s names. Each answer was checked at runtime in an environment with only the new release installed (every name resolves and every call’s arguments bind to its signature; the call is not executed, and missing required arguments are not checked), and a separate model, blind to the condition, judged whether it preserved the snippet’s intent rather than deleting or substituting the call. “Success” below means both checks passed under this protocol; it is not evidence of behavioural equivalence, and the intent check is an AI judgement, not an independent human verification.

With API history the model succeeded on 68.5% of tasks (CI 60.5–75.6%), against 56.6% closed book (exact McNemar $p = 0.0115$; 29 tasks fixed only with the history, 12 only without) and 60.1% with documentation ($p = 0.0428$; 21 vs 9). The gain comes from validity: answers written against the dataset’s record were API-compatible far more often, while slightly more of them were judged unfaithful, mostly where a “still available as” hint pointed to a same-named but different object. Both preregistered hypotheses hold, and both survive a Holm correction for the two comparisons (adjusted $p = 0.023$ and 0.0428 ; under the more conservative Bonferroni correction the documentation comparison would not, $p = 0.0856$). The effect is moderate, the preregistered study used one model and an AI

Table 6: Replication: VersionProbe-verified (closed book) and code repair success per condition.

Model	VersionProbe		Repair success			p vs docs
	item	pair	closed	docs	API Drift	
opus-5-5*	58.4%	26.6%	56.6%	60.1%	68.5%	0.043
sonnet-5-5	54.3%	18.3%	42.0%	56.6%	69.2%	0.0029
haiku-5-5	49.6%	9.2%	7.0%	46.2%	68.5%	<0.0001
fable-5-1	56.5%	22.4%	49.7%	62.2%	69.9%	0.071

*preregistered run; p : exact McNemar, API Drift vs documentation.

judge, and runtime validity checks names and signatures, not behaviour. In a post-hoc sensitivity check (not preregistered) we re-verified every answer in fresh environments with full binding (`bind`, so a missing required argument fails); the re-run reproduced every preregistered verdict, and full binding removed 2 successes (1 closed book, 1 documentation, 0 API Drift), giving 55.9%, 59.4% and 68.5% ($p = 0.0064$ and $p = 0.0294$).

5.7 Replication across models

After the preregistered run we repeated both experiments, unchanged, with three more Claude models: the same 984 VersionProbe-verified items, and the same 150 repair tasks, runtime check and blind judge (`claude-opus-5-5`), with every answer re-verified in fresh environments under both the preregistered and the full-binding rule. Table 6 shows that the ordering closed book < documentation < API Drift holds for every model. The striking result is convergence: with API history all four models reached 68.5%–69.9% repair success, whatever their closed-book score (7.0%–56.6%; the smallest model left 88.7% of snippets unchanged without help). On VersionProbe the smaller models are close to the 50% of a constant answer. The models are of one family, so this replicates the protocol, not the result across model families.

API history beat documentation for all 4 models, significantly for 3 (exact McNemar; the exception has $p = 0.071$), and closed book for all 4 ($p < 0.0001$ for the three replications). Pooled over models, 107 task–model pairs were fixed only with the history and 34 only with documentation. Full binding removed 0 API Drift successes across the four models.

6 Findings

Semantic versioning. Among 53,696 release-to-release transitions, 64.4% of major, 29.3% of minor and 13.4% of patch increments include at least one breaking change. 840 of the 1,872 libraries shipped at least one breaking patch release. These rates count any public-API break, including rarely used members, and are therefore upper bounds on user-visible breakage, consistent with the Maven observation that most breaking changes affect code few clients use [8]. A version bump is not a promise, so we also split projects by the policy their PyPI description states (pattern match, every match reviewed by hand): 54 say they follow SemVer, 2 say they do

not, 61 use calendar versions, and 1,755 state nothing there. Among the SemVer projects’ stable (≥ 1.0) releases, 16.7% of 707 patch releases (17 of 40 projects) and 24.5% of minor releases contain a high-confidence breaking change; these are candidate SemVer violations, subject to the audited precision and to the caveat above.

Deprecation before removal. Only 4.8% of removed functions, classes and methods carried a statically detectable deprecation marker (decorator, docstring directive or unconditional warning) in an earlier analysed release. Deprecation is often communicated through changelogs or conditional warnings rather than markers that tools can see.

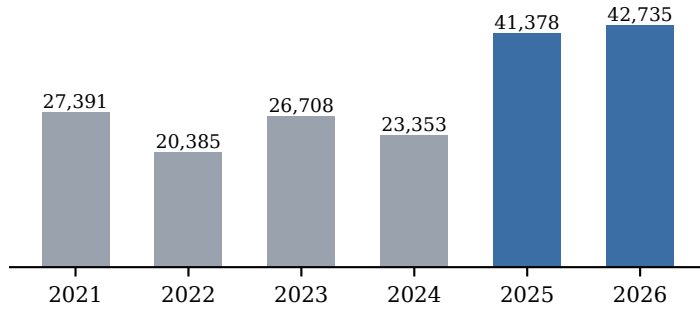
Recency. 83,846 high-confidence breaking changes were released from 2025 onward (Fig. 1; the per-year counts are affected by the 80-release cap). Code models whose training data predates them cannot know these changes without retrieval, which motivates checking generated code against version-specific API records.

7 Tools

apidrift-check parses Python code, resolves imports and simple constructor-based types (`df = pd.DataFrame(); df.append(...)`), and checks each referenced path and keyword argument against the API of the pinned release. By default it reports only names and keywords that were public in *some* analysed release but not the pinned one, each with a hint (“last present in 1.5.3”, “accepted up to 1.26.20”). On 1,823 source files of 14 widely used packages, checked against their own installed dependency versions, it raised 20 findings. Six point to APIs that really are absent in the installed versions, inside version-guarded or unused code paths. The rest are false alarms: parameters still accepted through compatibility decorators (e.g. SciPy’s `seed→rng` transition), attributes set dynamically, a method inherited from a compiled base class, and one name that was removed and later re-added outside the public API. The checker skips `if TYPE_CHECKING:` blocks and postponed annotations, which never run. A strict mode additionally flags names never seen in the API and checks arity. It reads only the Parquet row groups of the pinned libraries and the signatures they reference. A cold start (one file, one pin) takes 3.1 s of CPU and about 326 MB of memory, and checking all 1,421 files of pandas against its installed dependencies takes 45.7 s of CPU, on a shared two-core machine. It reads lockfiles (`uv.lock`, Poetry, `pylock.toml`, Pipfile), writes SARIF for GitHub code scanning, and ships as an experimental GitHub Action whose inputs reach the checker only as validated arguments.

apidrift-lookup and an MCP server. `apidrift-lookup` answers one-line queries offline: the history of a symbol (`history pandas.DataFrame.append`) and the breaking differences between any two analysed releases (`diff flask 2.3.3 3.0.0`), computed from the two API snapshots, each change listed once with its confidence label. `apidrift-mcp` exposes these lookups and the checker as read-only tools of a

High-confidence breaking changes by release year (2026: Jan-Oct)



Releases with at least one breaking change

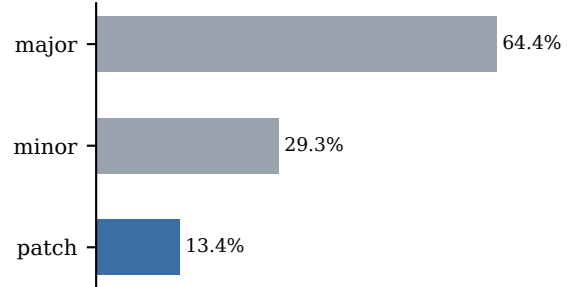


Figure 1: Left: deduplicated high-confidence breaking changes by year of the breaking release. Because each project’s history is capped at its 80 most recent releases, frequently releasing projects contribute only recent years, so the rise in 2025–2026 partly reflects coverage. Right: share of release transitions (calendar-versioned releases excluded) with at least one breaking change, by version increment.

Model Context Protocol server, so coding agents can consult the record before or after generating code. Every answer carries the coverage status of the rows it rests on, and a name that is not found is reported with the reminder that absence is weaker evidence than a recorded change.

VersionProbe contains 10,958 pairs from 583 libraries. Each pair is one snippet that is valid in release *A* and invalid in the next release *B*, derived from a high-confidence `removed`, `param_removed` or `param_renamed` event and verified with the checker in both directions. Items carry the release date of *B*, so evaluations can be split before and after a model’s training cutoff. A model is asked whether the snippet is valid for the stated version; a constant predictor scores 50%.

8 Limitations

PyPI API Drift is a high-precision, low-recall record, not complete ground truth. The analysis is static: compiled extensions are visible only through stubs, parameters altered by decorators are not modelled, and names created at runtime are often missing. This matters most for the libraries users try first: among the 100 most-downloaded libraries, 74.8% of API-history rows come from fully static modules (19.8% dynamic, 5.3% compiled boundary), against 86.3% for ranks 101–500; the `coverage_status` column marks every such row. Inheritance from classes in other distributions is not resolved (these cases are labelled medium). The history covers at most the 80 most recent final releases per project, so year-over-year counts are shaped by the cap, and release order follows version numbers (placeholder `0.0.0` uploads are now removed; Section 5.4). Behavioural changes with unchanged signatures are out of scope, and statically detectable deprecation markers cover only a small share of removals. All validation used samples checked by AI agents isolated from the pipeline, without human adjudication; we release a 200-item, double-labelling kit with an agreement script so this can be repeated by people. The clean out-of-sample precision estimates are 88.8% and

91.9%; figures labelled in-sample were measured after fixes informed by the same samples. Recall depends on the public-API definition: names imported into modules for internal use and runtime-generated objects are not tracked, and a tool that imports releases (AexPy) finds far more of the runtime-visible changes. The model experiments use 4 models of one family; the repair study checks names and signatures at runtime and intent with an AI judge, not program behaviour. The checker infers types only for objects constructed and assigned in the same scope, follows `super()` calls into library base classes but no other calls through subclasses, and never flags private or dunder names, which limits its recall on real upgrades.

9 Availability

Data (CC BY 4.0), code and tools (Apache-2.0), validation material and the benchmark are released with this paper. The pipeline is incremental and is intended to be re-run regularly, so new releases are added as they appear.

Acknowledgment

AI assistance was used for parts of the data-processing code and for editing this manuscript; the author reviewed all content and is responsible for it.

References

- [1] T. Wu *et al.*, “VersiCode: Towards version-controllable code generation,” arXiv:2406.07411, 2024.
- [2] 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:2411.05830, 2024.
- [3] D. Misra *et al.*, “GitChameleon 2.0: Evaluating AI code generation against Python library version incompatibilities,” arXiv:2507.12367, 2025.

- [4] 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,” arXiv:2406.09834, 2024.
- [5] T. Mazzucotelli, “griffe: Signatures for entire Python programs,” <https://mkdocstrings.github.io/griffe/>, 2026.
- [6] X. Du and J. Ma, “AexPy: Detecting API breaking changes in Python packages,” in *Proc. IEEE 33rd Int. Symp. Software Reliability Engineering (ISSRE)*, 2022, pp. 470–481.
- [7] S. Raemaekers, A. van Deursen, and J. Visser, “Semantic versioning versus breaking changes: A study of the Maven repository,” in *Proc. IEEE 14th Int. Working Conf. Source Code Analysis and Manipulation (SCAM)*, 2014, pp. 215–224.
- [8] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, “Breaking bad? Semantic versioning and impact of breaking changes in Maven Central,” *Empirical Software Engineering*, vol. 27, 2022.