Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Moving Fast and Breaking Clients? An Empirical Study of API Breaking Changes in Python LLM SDKs and Agent Frameworks

Revanth Krishna Maddula

revanth802@gmail.com

San Jose State University https://orcid.org/0009-0006-0596-6180

Research Article

Keywords: API evolution, breaking changes, semantic versioning, large language models, software ecosystems, dependency management

Posted Date: November 4th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11603167/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.

Moving Fast and Breaking Clients? An Empirical Study of API Breaking Changes in Python LLM SDKs and Agent Frameworks

Revanth Krishna Maddula
Independent Researcher, San Jose, CA, USA
revanth802@gmail.com

Abstract

Context. LLM-based applications depend on a young layer of libraries: provider SDKs such as OpenAI's and Anthropic's, and frameworks such as LangChain and CrewAI. Practitioners say these libraries change constantly, but this has not been measured. **Objective.** We measure how often LLM libraries break their public APIs, whether version numbers and deprecations warn users, and how dependent packages respond, compared with mature Python libraries. **Method.** We statically compared consecutive releases of 24 LLM libraries and 16 mature libraries (5,468 releases, January 2023 to October 2026), validated 160 detected breaking changes, and analyzed 8,380 dependency declarations from the 15,000 most downloaded PyPI packages. **Results.** LLM libraries released 2.3 times as often (median 31.4 vs. 13.7 releases per year) and broke their API in more releases (35.5% vs. 21.7%), four times as often relative to API size. Most breaking changes arrived in releases that semantic versioning does not permit to break, and only 2.9% of objects removed by LLM libraries were previously marked deprecated (8.5% for mature libraries). About 70% of detected breaking changes were real in both groups. Dependents bounded or pinned 50.8% of dependencies on LLM libraries (30.7% for mature ones), and 23.0% excluded the latest release (5.7%). **Conclusions.** The instability is real but concentrated in frameworks rather than provider SDKs, and in versioning and deprecation practices as much as in the rate of change.

Keywords: API evolution; breaking changes; semantic versioning; large language models; software ecosystems; dependency management

1 Introduction

Since 2023, a new layer of software has formed between LLM-based applications and the models they call. At its base are the providers' own SDKs, the official client libraries of OpenAI, Anthropic, Google, Mistral and others. Above them sit frameworks that promise to hide provider differences and add retrieval, tool use, memory and multi-agent orchestration: LangChain, LlamaIndex, CrewAI, Haystack, DSPy, Pydantic AI and many more. These libraries are among the most downloaded on the Python Package Index (PyPI), and thousands of other packages depend on them.

They also have a reputation. Practitioners routinely describe LLM frameworks as moving fast and breaking user code, and some teams have publicly abandoned frameworks for raw provider SDKs for that reason. The underlying technology changes quickly: new model capabilities, such as tool calling, structured output and multimodal input, appear every few months, and libraries race to expose them. Whether that pace translates into unusual instability for library users, however, is an empirical question that has not been answered. Prior research on LLM-related software has

studied how applications migrate when providers retire models [1], how LLM code generators use deprecated APIs [2], and which issues developers of agent frameworks and multi-agent systems report [3, 4], but not how the libraries' own APIs evolve.

The question matters for at least three groups. Application developers need to know how much to invest in pinning, testing and upgrading their dependencies. Library maintainers need evidence on whether their versioning and deprecation practices protect users. And because AI coding assistants are trained on snapshots of these same APIs, frequent breaking changes may make generated code obsolete faster [2].

We present a large-scale empirical study of API evolution in the Python LLM library ecosystem. We analyze 24 LLM provider SDKs and frameworks and, as a baseline, 16 mature and widely used Python libraries, over the same period from January 2023 to October 2026. For each library we compare the public API of every release with that of its predecessor, 5,468 releases in total, and we examine the dependency declarations that the 15,000 most downloaded PyPI packages make on these libraries. We ask four research questions:

- **RQ1 (Release practices):** How often do LLM libraries release, and how do they use version numbers?
- **RQ2 (Breaking changes):** How often do releases break the public API, and do version numbers signal it?
- **RQ3 (Deprecation):** Are removed APIs deprecated before removal?
- **RQ4 (Downstream response):** How do dependent packages constrain their dependencies on these libraries?

Our contributions are: (1) the first measurement of API breaking changes in the LLM library ecosystem, compared against a baseline of mature libraries and validated manually; (2) evidence that the instability is concentrated in frameworks rather than provider SDKs and in versioning and deprecation practices as much as in the rate of change; (3) evidence that the wider ecosystem has already priced this instability in, through far more restrictive dependency constraints; and (4) a replication package with all data and scripts.

2 Background

Semantic versioning in Python. Semantic versioning (SemVer) asks that a library increment its major version for incompatible API changes, its minor version for backward-compatible additions and its patch version for backward-compatible fixes; versions below 1.0.0 are considered initial development, in which "anything may change at any time" [6]. Python's packaging standard (PEP 440) defines version syntax and comparison but does not mandate SemVer [7], and many Python projects use other schemes. Clients express their expectations through version specifiers, such as a lower bound (`openai>=1.40`), an upper bound (`langchain-core<0.4`) or an exact pin (`==0.2.5`). Upper bounds protect against future breaking releases at the cost of blocking upgrades and fixes, and are debated in the Python community.

The LLM library layer. We distinguish three kinds of LLM libraries. *Provider SDKs* are official clients for one provider's API (for example `openai`, `anthropic`, `google-genai`, `mistralai`, `cohere`, `groq`, `ollama`). Several of them are generated from API specifications, so their public surface tracks the provider's API closely. *Frameworks* provide abstractions for building LLM applications and agents (for example `langchain`, `llama-index-core`, `crewai`, `haystack-ai`, `dspy`).

Infrastructure libraries sit in between, such as the LiteLLM gateway, which exposes many providers behind one interface, and the Model Context Protocol (MCP) SDK, which implements a protocol connecting models to tools.

3 Related Work

API breaking changes and semantic versioning. Raemaekers et al. analyzed the Maven repository and found that about one third of releases introduced at least one breaking change, including many releases that semantic versioning would not permit to break [8]. Xavier et al. found that 14.8% of API changes in popular Java libraries were breaking [9], and Brito et al. built APIDiff to detect such changes [10]. Decan and Mens compared how four packaging ecosystems follow SemVer, as revealed by dependency constraints [11], and Bogart et al. showed that ecosystems negotiate the costs of breaking changes through different values and policies [12]. For Python, Zhang et al. studied how framework APIs evolve and found frequent incompatible changes [13], and Du and Ma proposed AexPy to detect breaking changes in Python packages [14].

Deprecation and client reaction. Studies of deprecation found that deprecated APIs often lack replacement guidance [15], and that clients react to API evolution and deprecation slowly or not at all [16, 17]. Studies of dependency updates and technical lag found that clients frequently keep outdated dependencies [18, 19] and that ecosystems differ in their dependency network dynamics [20].

LLM software ecosystems. Hasan et al. studied the security and maintainability of MCP servers [5]. Wang et al. studied developer practices across AI agent frameworks [3], and Liu et al. analyzed the development and issues of multi-agent AI systems, finding that bugs are the most common issue category [4]. Kim studied how open-source applications migrate when providers retire models and endpoints, noting that, unlike library deprecation, "the deprecated artifact is a hosted service" [1]. Wang et al. showed that LLM-based code completion frequently uses deprecated library APIs [2]. None of these studies measures how the LLM libraries' own APIs change, which is our focus.

4 Study Design

4.1 Subject libraries

We selected 24 LLM libraries covering the three kinds described in Section 2: eight provider SDKs (`openai`, `anthropic`, `google-genai`, `google-generativeai`, `mistralai`, `cohere`, `groq`, `ollama`), fourteen frameworks (`langchain`, `langchain-core`, `langchain-community`, `langchain-openai`, `langgraph`, `llama-index-core`, `crewai`, `autogen-agentchat`, `pydantic-ai-slim`, `openai-agents`, `haystack-ai`, `dspy`, `semantic-kernel`, `instructor`), the `litellm` gateway and the MCP SDK (`mcp`). We chose them as the most prominent libraries of each kind by downloads and community visibility, and analyzed the distribution that contains the code (`pydantic-ai-slim` rather than the `pydantic-ai` meta-package).

As a baseline, we selected 16 mature, widely used Python libraries of comparable roles: four SDKs for commercial web APIs (`stripe`, `twilio`, `slack-sdk`, `google-cloud-storage`) and twelve general-purpose libraries and frameworks (`requests`, `httpx`, `aiohttp`, `pydantic`, `fastapi`, `flask`, `django`, `sqlalchemy`, `celery`, `click`, `rich`, `typer`). Comparing the two groups over the same period controls for changes in Python tooling and practices over time.

4.2 Releases

For each library we retrieved the full release history from the PyPI JSON API. We kept final releases (excluding pre-releases, development releases and yanked releases) published between 1 January 2023 and 29 October 2026, ordered them by version number, and compared each release with its predecessor in version order; the last release before the window served as the predecessor of the first release in it. Ordering by version rather than date compares maintenance releases of older branches with their own predecessors. This yielded 5,468 releases: 4,476 from LLM libraries and 992 from the baseline.

4.3 Detecting breaking changes

We downloaded each release (a pure-Python wheel, or the source distribution when no wheel existed) and analyzed it statically with Griffe [21], which builds a model of a package's modules, classes, functions and attributes from source and can report breaking changes between two versions. We counted a change only if it affected the public API, which we defined as every object reachable from the package's top-level module through names that do not start with an underscore, following re-exports. This definition includes objects defined in private modules but re-exported publicly, which is a common pattern in SDKs, and excludes internal helpers in private modules.

Griffe reports twelve kinds of change. We treat eight as *hard* breaking changes, which make previously valid client code fail: removal of a public object, removal of a parameter, a moved positional parameter, a changed parameter kind (for example, positional to keyword-only), a parameter that became required, a new required parameter, an object that changed kind, and a removed base class. We ignore the four *soft* kinds (changed defaults, incompatible return or attribute types, and changed attribute values), which are often invisible to callers. For each release we count the number of distinct public objects with at least one hard breaking change. After the validation described below, we also excluded objects in test modules and pydantic configuration artefacts (inner `Config` classes and `model_config` attributes), which the validation showed were consistently false positives.

4.4 Versioning and deprecation

We classified each release as a major, minor or patch increment relative to its predecessor. Following SemVer, a hard breaking change is *permitted* in a major increment and, for versions below 1.0.0, also in a minor increment; any other release with a hard breaking change counts as a SemVer *violation*. We apply this rule to all libraries as a common yardstick, while recognizing that not every project claims to follow SemVer. For every removed public object, we checked whether it carried a deprecation marker in the preceding release: a deprecation decorator, a docstring mentioning deprecation, or a `DeprecationWarning`, `PendingDeprecationWarning` or `FutureWarning` in its body.

4.5 Downstream dependencies

To study how clients respond, we took the 15,000 most downloaded PyPI packages over the 30 days before 1 October 2026 [22], retrieved the dependency declarations of each package's latest release, and kept those naming one of our 40 libraries (excluding self-dependencies). This yielded 8,380 package–dependency pairs: 1,251 on LLM libraries, from 579 packages, and 7,129 on baseline libraries, from 4,211 packages. For each pair we recorded whether the dependency is optional (declared under an extra), whether any declaration carries an upper bound or exact pin, and

whether the declared constraints exclude the dependency's latest final release as of 29 October 2026.

4.6 Validation

Static API comparison can misreport changes, for example when an object moves but remains importable through a lazy import or a deprecation shim. We therefore drew a stratified random sample of 160 detected hard breaking changes, 80 from each group and spread across libraries, and had each judged against a written codebook by inspecting the source of both releases. Each item was labeled *real* if code that correctly used the object in the old release would fail or need changes in the new one, *false positive* if such code would still work or the object was clearly not public API, or *unclear*. The judgments were made by four AI agents (Claude, Anthropic), each working blind on a quarter of the sample from the package sources alone, without access to our results; this tests the precision of our detector against an independent reading of the code, but it is not a human expert assessment. The author reviewed the reasons given for all false positives.

4.7 Use of AI tools in the research process

Data-collection and analysis scripts were written with the assistance of an AI model (Claude, Anthropic), and the same model, run as separate blind agents, performed the validation judgments described above. The author designed the study, reviewed all scripts, outputs and judgments, and takes responsibility for the results.

4.8 Statistical analysis

Our unit of comparison is the library. We compare per-library metrics between groups with two-sided Mann–Whitney U tests and report rank-biserial correlations as effect sizes. At the release level, we fit a logistic regression for whether a release contains a hard breaking change, with group, increment type and pre-1.0 status as predictors and standard errors clustered by library. For downstream dependencies, we compare proportions with χ^2 tests and fit logistic regressions with group and optionality as predictors, clustering by dependency.

5 Results

Table 1 summarizes all 40 libraries, and Table 2 compares the two groups. We analyzed 5,463 of the 5,468 releases; five could not be loaded or compared.

5.1 Validation of the detector

Of the 160 sampled breaking changes, 106 were judged real and 54 false positives; none was unclear. The eight sampled items in test modules or pydantic configuration artefacts were all false positives, so we excluded these categories from the analysis. On the remaining items, the precision of the detector was 70.3% for LLM libraries (52 of 74; 95% Wilson interval 59.1–79.5%) and 69.2% for the baseline (54 of 78; 58.3–78.4%). The remaining false positives had four main causes: objects still importable through lazy imports or deprecation shims (for example, module-level `__getattr__` functions), objects with public names that are internal in practice (such as execution-engine or telemetry classes), hooks that the framework rather than the user calls (such as pydantic validators and command-line callbacks), and changes visible only in type annotations. Because precision is almost identical in the two groups, false positives inflate absolute counts but do not explain

Table 1: Per-library results, January 2023 to October 2026. *Rel.*: releases in the window; */yr*: releases per year; *0.x*: % of releases below version 1.0; *Hard*: % of releases with at least one hard breaking change; *Viol.*: % of releases with a hard breaking change in an increment that SemVer does not permit to break; */1k*: hard breaking changes per thousand public API elements per year; *Rem.*: removed public objects; *Dep.*: % of removed objects deprecated in the preceding release. Upper block: LLM libraries; lower block: baseline.

Library	Kind	Rel.	/yr	0.x	Hard	Viol.	/1k	Rem.	Dep.
autogen-agentchat	framework	30	7.8	100	13	7	7.1	5	0
crewai	framework	212	55.4	69	49	24	169.3	551	1
dspy	framework	87	22.7	6	61	59	68.9	313	1
haystack-ai	framework	72	18.8	0	51	50	35.6	220	7
instructor	framework	97	25.4	33	34	29	191.4	193	2
langchain	framework	422	110.3	86	50	49	100.9	1201	2
langchain-community	framework	89	23.3	100	63	61	7.5	287	2
langchain-core	framework	273	71.4	70	18	18	20.8	139	34
langchain-openai	framework	130	34.0	62	3	2	13.0	4	25
langgraph	framework	251	65.6	84	31	28	178.2	318	1
llama-index-core	framework	235	61.4	100	46	44	27.0	261	2
openai-agents	framework	122	31.9	100	35	25	11.0	16	0
pydantic-ai-slim	framework	335	87.6	35	36	33	75.6	300	17
semantic-kernel	framework	89	23.3	0	56	56	22.6	281	2
litellm	gateway	794	207.6	45	32	31	263.0	855	1
mcp	protocol	64	16.7	0	47	45	73.9	318	4
anthropic	provider	233	60.9	90	18	5	29.8	127	5
cohere	provider	205	53.6	0	42	41	375.3	725	0
google-genai	provider	118	30.9	1	25	22	5.8	116	4
google-generativeai	provider	24	6.3	100	62	38	14.9	35	0
groq	provider	53	13.9	83	21	0	18.6	43	2
mistralai	provider	104	27.2	14	57	53	68.7	939	2
ollama	provider	35	9.2	100	20	11	52.7	5	0
openai	provider	402	105.1	5	16	16	50.5	564	0
celery	framework	14	3.7	0	29	29	1.7	15	0
django	framework	112	29.3	0	24	21	10.9	428	18
fastapi	framework	201	52.6	100	15	6	42.4	90	7
flask	framework	15	3.9	0	20	13	69.8	109	18
aiohttp	library	63	16.5	0	22	22	4.8	23	0
click	library	15	3.9	0	33	33	16.4	23	0
httpx	library	13	3.4	100	8	0	9.3	0	–
pydantic	library	85	22.2	0	24	22	72.9	531	1
requests	library	13	3.4	0	0	0	0.0	0	–
rich	library	35	9.2	0	23	23	3.1	4	0
sqlalchemy	library	72	18.8	0	54	53	27.4	671	9
typer	library	62	16.2	100	21	8	15.7	14	0
google-cloud-storage	provider	40	10.5	0	10	8	4.8	1	0
slack-sdk	provider	43	11.2	0	7	7	0.3	3	33
stripe	provider	123	32.2	0	10	3	1.0	35	0
twilio	provider	86	22.5	0	85	83	18.2	915	8

Table 2: LLM libraries (24) versus baseline libraries (16): per-library medians, two-sided Mann–Whitney U tests and rank-biserial correlations (r; positive values mean higher in LLM libraries).

Metric	LLM	Baseline	p	r
Releases per year	31.4	13.7	<0.001	0.62
Releases below version 1.0 (%)	65.5	0.0	0.003	0.54
Releases with a hard breaking change (%)	35.5	21.7	0.020	0.44
precision-adjusted (%)	30.0	19.1	0.022	0.43
Non-major releases with a hard breaking change (%)	35.3	21.6	0.018	0.45
Releases violating SemVer (%)	29.9	17.3	0.055	0.36
Hard breaking changes per year	102.0	10.0	0.026	0.42
Hard breaking changes per 1k API elements per year	43.1	10.1	0.002	0.59
Removed objects deprecated beforehand (%)	2.0	0.5	0.65	0.09

differences between groups. We report raw counts and, where relevant, precision-adjusted estimates, in which a release with k detected breaking changes counts as breaking with probability $1-(1-p)^k$.

5.2 RQ1: Release practices

LLM libraries release far more often than mature libraries: a median of 31.4 releases per year against 13.7 ($p < 0.001$, $r = 0.62$). The most active, LiteLLM, published 794 releases in under four years, more than four per week; LangChain published 422 and OpenAI's SDK 402. Most of these releases are small: 73.0% of LLM library releases were patch increments, against 65.8% for the baseline. LLM libraries also make extensive use of the pre-1.0 version range, which SemVer reserves for initial development: the median LLM library published 65.5% of its releases below version 1.0, against none for the median baseline library ($p = 0.003$). Pre-1.0 versions are not unique to AI (FastAPI, HTTPX and Typer in the baseline also remain below 1.0), but in the LLM ecosystem they are the norm, including for frameworks with tens of millions of monthly downloads. Major increments were rare in both groups: 29 of 4,472 LLM releases (0.6%) and 23 of 991 baseline releases (2.3%).

5.3 RQ2: Breaking changes

LLM libraries broke their public API in a larger share of releases (Figure 1). The median LLM library introduced at least one hard breaking change in 35.5% of its releases, against 21.7% for the median baseline library ($p = 0.020$, $r = 0.44$); after adjusting for detector precision, the medians are 30.0% and 19.1%. Sixteen of the 24 LLM libraries broke their API in at least a quarter of their releases, against four of the 16 baseline libraries. Normalized by API size, LLM libraries introduced 43.1 hard breaking changes per thousand public API elements per year, four times the baseline's 10.1 ($p = 0.002$, $r = 0.59$). Removals of public objects dominate in both groups (63.3% of hard breaking changes in LLM libraries and 54.5% in the baseline), followed by removed parameters, moved positional parameters and new required parameters.

The ecosystem is far from uniform. Frameworks are the least stable: the median framework broke its API in 40.7% of releases, with the highest rates in `langchain-community` (63%), `dspy` (61%), `semantic-kernel` (56%), `haystack-ai` (51%) and `langchain` (50%). Provider SDKs are much closer to the baseline, with a median of 22.7%: the official OpenAI and Anthropic SDKs broke their API in 16% and 18% of releases, although the Mistral (57%), Cohere (42%) and legacy Google (62%) SDKs were considerably less stable. Some LLM libraries are as stable as the best

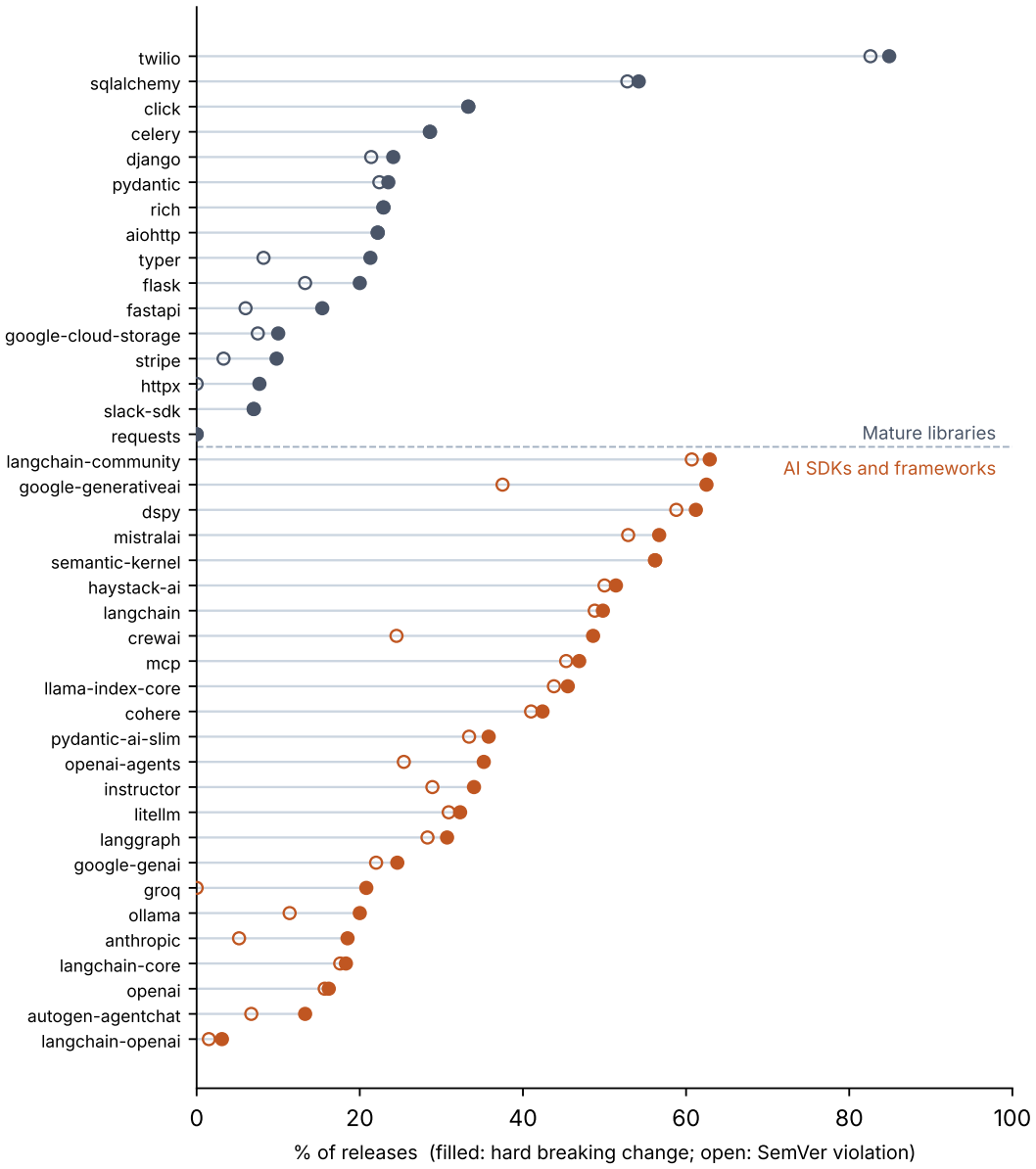


Figure 1: Share of releases with at least one hard breaking change (filled markers) and with a hard breaking change in an increment that SemVer does not permit to break (open markers), per library.

baseline libraries, notably `langchain-openai` (3%) and `autogen-agentchat` (13%). The baseline also has outliers: Twilio's generated SDK broke its API in 85% of releases and SQLAlchemy, during its 2.0 transition, in 54%.

At the release level, the difference between groups is smaller and less certain. Pooled across libraries, 34.6% of LLM releases and 25.9% of baseline releases contained a hard breaking change. In a logistic regression controlling for increment type and pre-1.0 status, with standard errors clustered by library, the odds ratio for LLM libraries was 1.57 (95% CI 0.70–3.55, $p = 0.28$), while major increments (odds ratio 10.3) and minor increments (1.74) were significantly more likely to break than patch increments. The large variation between libraries within each group means that membership in the LLM group alone does not reliably predict whether a given release breaks; what distinguishes the groups is the combination of a higher breaking rate per library with a much higher release rate.

Version numbers rarely signal breaking changes. In both groups, most breaking releases were increments that SemVer does not permit to break: 88.6% of LLM releases with a hard breaking change and 82.5% of baseline ones. Two thirds of breaking LLM releases (65.8%) were patch increments, against 56.8% in the baseline. The median per-library share of SemVer-violating releases was 29.9% for LLM libraries and 17.3% for the baseline, a difference that falls just short of conventional significance ($p = 0.055$). Two libraries illustrate the alternatives. Anthropic's SDK made most of its breaking changes while below version 1.0 and in minor increments, as SemVer permits, giving it a violation share of 5%. In the baseline, Stripe's SDK published 12 major versions in the window, roughly two per year, and confined almost all breaking changes to them (violation share 3%): frequent breaking changes need not mean unannounced ones.

5.4 RQ3: Deprecation before removal

Deprecation before removal was rare in both groups but rarer among LLM libraries. Of 7,816 public objects removed by LLM libraries, 223 (2.9%) carried a deprecation marker in the preceding release, against 243 of 2,862 (8.5%) in the baseline ($\chi^2 = 158.2$, $p < 0.001$). The difference is driven by a few libraries in each group that deprecate systematically, such as Django and Flask in the baseline and `pydantic-ai-slim` and `langchain-core` among LLM libraries; per-library medians are low in both groups and do not differ significantly (2.0% vs. 0.5%, $p = 0.65$). Our measure captures deprecation only in the release immediately before removal and only through markers in code and docstrings, so it is a lower bound; release notes and documentation may announce removals that the code does not mark.

5.5 RQ4: Downstream response

Dependent packages treat LLM libraries very differently from mature ones (Figure 2). Among the 15,000 most downloaded PyPI packages, 50.8% of dependencies on LLM libraries carried an upper bound or exact pin, against 30.7% of dependencies on baseline libraries ($\chi^2 = 191.2$, $p < 0.001$), and 23.0% excluded the library's latest release, against 5.7% ($\chi^2 = 417.1$, $p < 0.001$). Dependencies on LLM libraries were also far more often optional (56.3% vs. 22.3%), reflecting their role as pluggable backends. Controlling for optionality, with standard errors clustered by library, the odds of an upper bound or pin were 2.76 times higher for LLM libraries (95% CI 1.75–4.37) and the odds of excluding the latest release 4.28 times higher (2.37–7.75). The pattern holds for the 1,098 LLM and 5,113 baseline dependencies declared by packages released in the last six months (52.6% vs. 33.3% bounded; 23.3% vs. 5.5% excluding the latest release), so it does not merely reflect stale packages.

Constraints differ by library in ways that mirror their history. Frameworks attract the

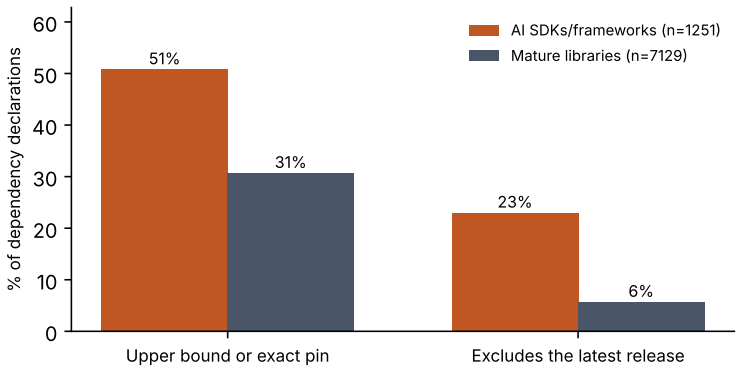


Figure 2: Dependency declarations on LLM and baseline libraries among the 15,000 most downloaded PyPI packages.

most upper bounds: 92.6% of dependencies on `llama-index-core`, 69.6% on `mcp`, 66.4% on `langchain-core` and 61.1% on `langgraph` are bounded. Provider SDKs attract fewer bounds (33.0% for `openai`, 40.4% for `anthropic`) but are more often left behind by their latest release (20.3% and 39.4% of dependencies exclude it), because both SDKs published new major versions in 2026 that bounded dependents do not admit. Overall, the most downloaded Python packages have already priced the instability of the LLM library layer into their dependency declarations.

6 Discussion

6.1 How unstable is the LLM library layer?

Our results support the practitioners' perception, with qualifications. Per library, LLM libraries break their public API in a larger share of releases, four times as often relative to their API size, and release more than twice as often, so their users meet breaking changes far more frequently in absolute terms. The instability is concentrated in frameworks; the official SDKs of the largest providers are much closer to mature libraries. And the release-level comparison shows that group membership alone does not predict whether a release breaks: some LLM libraries are as stable as the best baseline libraries, and some baseline libraries break as often as the least stable LLM frameworks. "LLM libraries are unstable" is therefore too coarse; "LLM frameworks change faster than their versioning and deprecation practices can signal" fits the evidence better.

6.2 Versioning and deprecation, not only the rate of change

The more actionable finding concerns how change is communicated. In both groups most breaking changes arrive in releases that SemVer does not permit to break, and in LLM libraries two thirds of them arrive in patch increments. Deprecation markers preceded fewer than 3% of removals. Taken together with the heavy use of pre-1.0 versions, this means that version numbers in the LLM ecosystem carry little information about compatibility. Stripe's SDK shows that frequent breaking change is compatible with honest versioning: it published a major version roughly every six months and kept breaking changes out of other releases. For maintainers of LLM libraries, the lowest-cost improvements are to reserve breaking changes for major (or, before 1.0, minor) increments, to keep deprecated objects importable with a warning for at least one release, and to leave the 0.x range once a library has a large user base.

6.3 The ecosystem has priced instability in

Downstream packages have responded rationally: they bound and pin dependencies on LLM libraries far more often and are more often left behind by their latest releases. This defensive posture has costs. Upper bounds block bug fixes and security fixes, and when many packages bound the same library differently, users cannot install them together. With a quarter of dependencies on LLM libraries already excluding the latest release, dependency conflicts are a predictable consequence for applications that combine several LLM-related packages. Better versioning upstream would let downstream packages relax these bounds.

6.4 Implications for AI coding tools

AI coding assistants generate code against APIs they learned from training snapshots, and they already use deprecated APIs frequently [2]. The APIs they are most often asked to write against, those of LLM frameworks, are also the ones that change fastest and warn least. Retrieval of current documentation, version-aware code generation and checks against the installed version of a dependency are likely to matter more for LLM application code than for code using mature libraries.

6.5 Implications for research

Our study suggests three directions. First, static detection of breaking changes in Python remains imprecise: about 30% of detected changes were false positives, mostly from lazy imports, internal objects with public names and framework-invoked hooks, which better tools could recognize. Second, the consequences for clients deserve study: how often applications actually break, how long fixes take, and whether bounded dependencies lead to conflicts. Third, the interplay between generated provider SDKs and the APIs they wrap could explain why some provider SDKs are stable and others are not.

7 Threats to Validity

Construct validity. We measure breaking changes statically, from the public API as reachable through non-underscore names. This misses behavioral breaking changes that leave signatures intact, and it cannot see APIs defined dynamically, so it underestimates breakage in some respects; it also reports false positives, which our validation estimates at about 30% in both groups. The validation judgments were made by AI agents working blind from package sources, which tests the detector's precision against an independent reading of the code but is not a human expert assessment; the author reviewed every false-positive judgment. Applying SemVer's rules to libraries that do not claim to follow SemVer measures how much information version numbers carry, not whether projects keep their own promises. Our deprecation measure covers only markers in code and docstrings in the release immediately preceding a removal.

Internal validity. The two groups differ in age, size and purpose, and we do not claim that being an LLM library causes instability. Libraries that are young, fast-growing and tracking a rapidly changing upstream API would plausibly behave similarly in any domain; our baseline includes some such libraries (for example FastAPI and Typer), but not many.

External validity. We studied 24 prominent Python LLM libraries and 16 baseline libraries; results may differ for less popular libraries and for other languages, notably the JavaScript and TypeScript ecosystem, in which many LLM applications are also built. The selection of baseline

libraries affects the comparison; we chose widely used libraries with roles comparable to the LLM libraries, and we report every library individually so that readers can form their own comparisons. Downstream results describe the 15,000 most downloaded PyPI packages, not applications, which usually do not publish to PyPI.

Reliability. All data come from public sources, and we provide the complete dataset and analysis scripts.

8 Conclusion

LLM SDKs and agent frameworks have become core dependencies of a growing share of Python software. Measured over nearly four years and 5,468 releases, they release more than twice as often as mature libraries, break their public API in a larger share of releases and four times as often relative to their size, rarely deprecate before removing, and seldom signal breaking changes through their version numbers. The instability is concentrated in frameworks; the largest providers' official SDKs come close to mature libraries. The wider ecosystem has responded by bounding and pinning these dependencies far more often than others. Better versioning and deprecation practices would let the LLM library layer keep its pace of innovation while sparing its users much of the cost.

Declarations

Funding. This research received no specific grant from any funding agency in the public, commercial or not-for-profit sectors.

Conflict of interest. The author declares no competing interests. No organization whose libraries are studied funded, reviewed or approved this work.

Data availability. The release-level and dependency-level datasets, the validation sample with codebook and judgments, and all collection and analysis scripts are provided as supplementary material and will be deposited in a public archive with a DOI upon acceptance.

Ethics approval. Not applicable: the study analyzes publicly available software packages and metadata and involves no human participants.

Author contributions. R.K.M. conceived and designed the study, collected and analyzed the data, and wrote the manuscript.

Use of AI tools. Claude (Anthropic) was used to write data-collection and analysis scripts, to perform the blind validation judgments described in Section 4, and to draft and edit text. The author designed the study, reviewed all results and judgments, and takes full responsibility for the content.

References

[1] H. L. Kim, When the model retires: An empirical study of LLM migration in open-source applications, arXiv:2609.31288 (2026).

[2] C. Wang, K. Huang, J. Zhang, Y. Feng, L. Zhang, Y. Liu, X. Peng, LLMs meet library evolution: Evaluating deprecated API usage in LLM-based code completion, in: Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE), 2025.

[3] Y. Wang, X. Xu, J. Chen, T. Bi, W. Gu, Z. Zheng, An empirical study of agent developer practices in AI agent frameworks, arXiv:2512.01939 (2025).

[4] D. Liu, K. Upadhyay, V. Chhetri, A. B. Siddique, U. Farooq, A large-scale study on the development and issues of multi-agent AI systems, arXiv:2601.07136 (2026).

[5] M. M. Hasan, H. Li, E. Fallahzadeh, G. K. Rajbahadur, B. Adams, A. E. Hassan, Model Context Protocol (MCP) at first glance: Studying the security and maintainability of MCP servers, arXiv:2506.13538 (2025).

[6] T. Preston-Werner, Semantic Versioning 2.0.0, `https://semver.org/`.

[7] N. Coghlan, D. Stufft, PEP 440 – Version identification and dependency specification, Python Enhancement Proposals (2013). `https://peps.python.org/pep-0440/`.

[8] S. Raemaekers, A. van Deursen, J. Visser, Semantic versioning and impact of breaking changes in the Maven repository, Journal of Systems and Software 129 (2017) 140–158.

[9] L. Xavier, A. Brito, A. Hora, M. T. Valente, Historical and impact analysis of API breaking changes: A large-scale study, in: Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 138–147.

[10] A. Brito, L. Xavier, A. Hora, M. T. Valente, APIDiff: Detecting API breaking changes, in: Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track, 2018, pp. 507–511.

[11] A. Decan, T. Mens, What do package dependencies tell us about semantic versioning?, IEEE Transactions on Software Engineering 47 (6) (2021) 1226–1240.

[12] C. Bogart, C. Kästner, J. Herbsleb, F. Thung, When and how to make breaking changes: Policies and practices in 18 open source software ecosystems, ACM Transactions on Software Engineering and Methodology 30 (4) (2021) Article 42.

[13] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, Y. Xiong, How do Python framework APIs evolve? An exploratory study, in: Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 81–92.

[14] X. Du, J. Ma, AexPy: Detecting API breaking changes in Python packages, in: Proceedings of the 33rd IEEE International Symposium on Software Reliability Engineering (ISSRE), 2022, pp. 470–481.

[15] G. Brito, A. Hora, M. T. Valente, R. Robbes, Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems, in: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2016, pp. 360–369.

[16] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, M. T. Valente, How do developers react to API evolution? The Pharo ecosystem case, in: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 251–260.

[17] A. A. Sawant, R. Robbes, A. Bacchelli, On the reaction to deprecation of clients of 4+1 popular Java APIs and the JDK, Empirical Software Engineering 23 (4) (2018) 2158–2197.

[18] R. G. Kula, D. M. German, A. Ouni, T. Ishio, K. Inoue, Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration, Empirical Software Engineering 23 (1) (2018) 384–417.

[19] A. Zerouali, E. Constantinou, T. Mens, G. Robles, J. González-Barahona, An empirical analysis of technical lag in npm package dependencies, in: New Opportunities for Software Reuse (ICSR 2018), Lecture Notes in Computer Science, vol. 10826, Springer, 2018, pp. 95–110.

[20] A. Decan, T. Mens, P. Grosjean, An empirical comparison of dependency network evolution in seven software packaging ecosystems, Empirical Software Engineering 24 (1) (2019) 381–416.

[21] T. Mazzucotelli, Griffe: Signatures for entire Python programs, `https://mkdocstrings.github.io/griffe/` (accessed October 2026).

[22] H. van Kemenade, Top PyPI Packages: A monthly dump of the 15,000 most-downloaded packages from PyPI, `https://hugovk.github.io/top-pypi-packages/` (snapshot of 1 October 2026).