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.

Just Pin It: Can Coding Agents Be Trusted to Choose Dependency Versions?

Leonardo Leite

leo.leite@ic.ufal.br

Federal University of Alagoas (UFAL)

Caio Barbosa

Pontifical Catholic University of Rio de Janeiro (PUC-Rio)

Elvys Soares

Federal University of Alagoas (UFAL)

Baldoino Fonseca

Federal University of Alagoas (UFAL)

Research Article

Keywords: dependency management, large language models, coding agents, software supply-chain security, sycophancy, software vulnerabilities

Posted Date: November 25th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10549928/v1

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

Additional Declarations: No competing interests reported.

Just Pin It: Can Coding Agents Be Trusted to Choose Dependency Versions?

Leonardo Leite[1*], Caio Barbosa[2], Elvys Soares[1], Baldoino Fonseca[1]

[1*]Institute of Computing, Federal University of Alagoas (UFAL), Av. Lourival Melo Mota, S/N, Tabuleiro do Martins, Maceió, 57072-970, Alagoas, Brazil.

[2]Department of Informatics, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Marques de São Vicente, 225, Gávea, Rio de Janeiro, 22451-900, Rio de Janeiro, Brazil.

*Corresponding author(s). E-mail(s): leo.leite@ic.ufal.br;
Contributing authors: csilva@inf.puc-rio.br; elvys.soares@gmail.com; baldoino@ic.ufal.br;

Abstract

Large language model (LLM) coding agents increasingly manage software dependencies. Adding one means choosing a package and a specific version to pin, and because the agent answers from knowledge frozen at a training cutoff, it makes that security-relevant choice from memory rather than from a package index or a vulnerability database. To measure the decision where it occurs, we reconstruct 50 dependency-introducing commits from 29 open-source Python repositories at their parent state, ask six Claude models to perform the same task under tool and network isolation, and score each choice on a time-bounded, multidimensional correctness vector rather than a single pass or fail. We then challenge every choice in a second conversational turn, with a neutral prompt, a true defect, or a false alarm, separating genuine repair from sycophantic capitulation. Across 1,128 runs, newer models rarely name nonexistent versions but are no better at avoiding known-vulnerable ones, and they pin six to nine releases behind the newest release they could have known; their security-and-existence performance does not beat a random eligible draw. Under a false vulnerability alarm, agents abandoned a verified-secure pin in 98.5% of cases and never improved it, and even a contentless request to reconsider overturned 63.7% of good pins; naming a true defect did not measurably outperform that nudge. Suggesting a pin is worth

delegating only with downstream verification, and an agent's agreement under challenge carries no information about whether it is right.

Keywords: dependency management, large language models, coding agents, software supply-chain security, sycophancy, software vulnerabilities

1 Introduction

Large language models (LLMs) increasingly operate as coding agents. They read repositories, run tools, and resolve issues end to end, opening pull requests that human developers once wrote (Xi et al. 2023; Jimenez et al. 2024; Twist and Zhang 2025). That puts them in charge of a decision that shapes a system's stability and security. Dependency management asks which third-party libraries a project uses and at which versions (Singla et al. 2026); choosing a version weighs new features against breaking changes and the risk of a compromised package (Kula et al. 2018; He et al. 2025), and the agent making that call is often unreliable on matters of fact (Wang et al. 2024).

That decision carries a security boundary. Because an installed package runs with the full privileges of the application that imports it, one dependency can be enough to compromise a system. Open-source supply chains are now an actively exploited attack surface, as adversaries publish malicious packages and hijack popular or abandoned names, and taxonomies of these attacks catalog dozens of injection vectors (Ladisa et al. 2022; Ohm et al. 2020); PyPI[1] has been the target of large-scale malicious-package campaigns (Guo et al. 2023). Even without a deliberate attack, ordinary releases accumulate disclosed vulnerabilities over time, and one flawed version can propagate risk to every project that depends on it (Zerouali et al. 2022; Robinson et al. 2026). Selecting a dependency, and a version of it, is not a neutral convenience.

The concern is not hypothetical, code produced by LLMs agents frequently contains vulnerabilities and insecure dependencies (Basic and Giaretta 2026; Liu et al. 2025b), the same weaknesses recur across model outputs (Kordonsky et al. 2026), and generated projects are often not reproducible because their dependency specifications are incomplete or unstable (Vangala et al. 2025). The version an agent selects is a principal point at which this risk enters the codebase.

The version decision is acute for LLM agents because of how they make it. Adding a dependency means writing a specific version identifier next to a third-party import or in a dependency manifest file (Wang et al. 2026; Singla et al. 2026). An LLM answers from static parametric knowledge fixed at a time cutoff (Zhu et al. 2024; Pęzik et al. 2025), so it makes the choice without consulting a live package registry or vulnerability database, it pins from memory. Three failures recur: (i) models hallucinate package names that do not exist (Spracklen et al. 2025; Krishna et al. 2025; Haque et al. 2025); (ii) they lose track of API changes released after the cutoff (Ashik et al. 2026); and (iii) they pin versions with known vulnerabilities, the scale is not marginal. PinTrace,

[1] Python Package Index: https://pypi.org/.

a large-scale measurement of LLM-specified library versions, reports that 36.70%–55.70% of tasks made by LLMs contain at least one known Common Vulnerability and Exposure (CVE), most rated critical or high (Wang et al. 2026).

Current evaluations of that decision leave two gaps. Most security assessments of LLM-generated code look for weakness patterns inside isolated snippets, or check only whether the code runs (Singla et al. 2026). The work closest to the version decision does measure version-level risk, but scores isolated prompt–response pairs drawn from Stack Overflow tasks (Wang et al. 2026); a one-shot benchmark cannot observe the decision inside a real project, where the surrounding code and existing dependencies shape the choice (Hu et al. 2026). The second gap is conversational. Models often fail to fix their own mistakes (Olausson et al. 2024; Tsui 2025), and they can abandon a correct answer under social pressure (Sharma et al. 2025), but neither behavior has been measured on the dependency-version decision. We know of no prior work that characterizes how LLMs select dependency versions in real open-source projects, nor how those choices hold up when challenged in conversation.

To close this gap, we measure dependency-version behavior inside the project where it occurs. For each case we reconstruct the state of a real open-source Python project immediately before a developer introduced a dependency, and ask a Claude coding agent to add the same dependency using only its internal knowledge, with no package index or package manager. We score every choice on a multidimensional correctness vector, bounded in time so it is judged against what the model could reliably have known at its cutoff. We then challenge the model's pin in a second conversational turn, under a neutral, directed, or misleading critique, which separates genuine repair of a flawed pin from the sycophantic abandonment of one that was already correct.

This paper makes the following contributions:

- **A reusable measurement instrument and dataset.** We curate a set of anchors (real dependency-introducing commits mined from open-source Python repositories), reconstruct each repository at its parent commit, and build a verification-and-critique harness that elicits a model's version choice and scores it under tool and network isolation. The instrument is designed to generalize to other models and ecosystems.
- **A correctness profile of parametric-knowledge pins (RQ1).** We characterize the versions that multiple Claude models pin without registry access, scoring each choice against a time-bounded, multidimensional correctness vector rather than a single pass-or-fail verdict, with the developer's own pin as a reference point rather than an oracle.
- **A separation of genuine repair from sycophantic false correction under critique (RQ2).** To our knowledge, this is the first study to measure, for the dependency-version decision, whether a coding agent holds or abandons its pin when challenged in conversation, distinguishing the repair of a flawed pin from capitulation on one that was already correct.
- **A cross-model comparison with a single confound-controlled contrast (RQ3).** We report the correctness profile across Claude generations and tiers

descriptively, reserving one confound-controlled generational comparison as the sole confirmatory test.

The remainder of this paper is organized as follows. Section 2 reviews related work and introduces the concepts the study relies on: dependency selection, temporal staleness and vulnerability disclosure, and model behavior under critique. Section 3 presents the study design, including the research questions, corpus construction, and analysis methods. Section 4 reports the results, Section 5 discusses threats to validity, and Section 6 concludes.

2 Background and Related Work

2.1 Dependency Selection: Package and Version

Code generation became a subfield of its own after coding agents showed that a general LLM could synthesize working programs from natural language (Chen et al. 2021), and such *code LLMs* are now surveyed in depth (Jiang et al. 2026; Zhao et al. 2026). Wrapped in a loop that lets them read files and run tools, they act as agents that edit real projects (Liu et al. 2025a), where adding a dependency is two decisions: which package to import, and which version to pin.

The first decision is already error-prone. Models emit imports for packages that do not exist, a failure mode called *slopsquatting* because an attacker can pre-register the hallucinated name and wait for it to be installed (Spracklen et al. 2025; Krishna et al. 2025); the effect persists on current models, including recent Claude releases (Churilov 2026; Twist et al. 2026).

The second decision, the version, matters only once the package name is real, and it is a trade-off. An exact **pin** (`==1.2.3`) locks a single version: the build stays reproducible, but it slowly falls behind and misses security patches. A **range** (`>=1.2,<2.0`) lets the version float: it picks up fixes, but a new release can break the build. Neither option is always right, and developers often delay updates deliberately (Rahman et al. 2025; He et al. 2025; Kula et al. 2018). A constraint takes effect only when a resolver installs the package together with everything it depends on (its transitive closure), a step that can itself fail or hit conflicts (Chowdhury et al. 2026; Mahon et al. 2025).

Version-level risk is the line of work nearest ours. One strand treats *whether* and *what* to depend on as a benchmark task in its own right (Singla et al. 2026). Another looks at agents in practice: the pull requests they author usually pin a specific version rather than leave it unconstrained, and the specifications they generate are often incomplete enough to break a build (Twist and Zhang 2025; Vangala et al. 2025). Closest of all is PinTrace, which scores version-level vulnerability across a thousand LLM-generated tasks and finds risky versions to be a systemic, cross-model pattern rather than isolated error (Wang et al. 2026); its tasks, however, are standalone Stack Overflow prompts, each scored once against a single vulnerability state. We keep the package fixed at a real library a developer actually adopted and study the version decision inside the project state that developer faced, comparing it against the developer's own pin as a reference rather than a ground truth, and re-examining it when the choice is challenged in conversation.

2.2 Temporal Staleness and Vulnerability Disclosure

An LLM answers from *parametric knowledge*: the facts stored in its weights during training, which it recalls when answering, without consulting any outside source. That knowledge stops at a *cutoff* date, after which the model has not seen new releases, deprecations, or vulnerability reports (Zhu et al. 2024; Ashik et al. 2026).

This cutoff is not a single clean date. The *training-data cutoff* a provider states comes no earlier than the point where the model's knowledge is actually *reliable*, because its coverage of recent events fades before it stops (Cheng et al. 2024; Pęzik et al. 2025). We therefore work with two dates, and score recency and security against the reliable cutoff (Table 1, from the provider's documentation (Anthropic 2026)) rather than the declared training-data cutoff.

This staleness is what makes version selection hard: when a library changes its interface or drops a function after the cutoff, the model has not seen the change. A growing set of benchmarks measures the problem, and model-editing methods try to patch newer knowledge back in (Kuhar et al. 2024; Wang et al. 2025b,a; Huang et al. 2026; Lin et al. 2025; Mahmud et al. 2024; Zhu et al. 2026). Closest to our task is GitChameleon, which fixes a library version, asks the model to write code for it, and runs that code to check whether it works (Islah et al. 2024). All of this asks whether the generated *code* matches the right interface for a version. We ask the question that comes first: which *version* to pin at all.

Security is the second way a version goes wrong, and whether a version counts as secure can change over time. A vulnerability is a weakness an attacker can exploit (Hanif et al. 2021); a Common Weakness Enumeration (CWE) names a *type* of weakness, while a Common Vulnerabilities and Exposures (CVE) identifier names a *specific* disclosed vulnerability in a particular version range.[2] These name the two ways an agent can introduce risk: by writing a weak line of code, or by trusting a vulnerable package version. The first channel is well documented but is not what we measure. LLM-produced code is often insecure: GitHub Copilot emits vulnerable code in roughly 40% of security-sensitive scenarios (Pearce et al. 2022), developers using AI assistants write less secure code yet report more confidence in it (Perry et al. 2023), and the weaknesses recur across more than forty CWE categories (Kordonsky et al. 2026; Fu et al. 2025), though the comparison with human developers is not wholly one-sided (Asare et al. 2024). The second channel is ours: an empirical study of open-source LLM projects found that roughly three-quarters carried at least one vulnerable dependency, undisclosed far longer than comparable flaws in the broader Python ecosystem (Liu et al. 2025b).

Whether a pinned version is secure therefore depends on when it is judged. The Open Source Vulnerabilities (OSV) database records disclosures against affected version ranges[3], but a version not known to be vulnerable at a model's cutoff can acquire a disclosure months later. Vulnerabilities in Python packages take more than three years to be discovered on average and are often fixed only after public disclosure (Alfadel et al. 2023), with similar gaps documented in Maven and Go (Yang-Smith

[2] The Common Weakness Enumeration (https://cwe.mitre.org/) and the Common Vulnerabilities and Exposures list (https://www.cve.org/).

[3] See the Open Source Vulnerabilities database at https://osv.dev.

and Abdellatif 2025; Hu et al. 2024), and even a committed fix can be hard to locate among a project's commits (Sabetta et al. 2024). We therefore score each choice twice, at the reliable cutoff and at a later verification snapshot. Counting on the two downstream defenses instead is unsafe. Detection tools disagree with one another on the same flaws (Lira et al. 2026; Mandl et al. 2026; Basic and Giaretta 2026), and LLM repair, strong on synthetic scenarios, falls away on real-world historical CVEs (Pearce et al. 2023). We measure the version decision at the moment the agent makes it.

2.3 Behavior Under Critique: Self-Correction and Sycophancy

Our critique step probes two opposite failures, each studied on its own in earlier work. On one side, models are poor at fixing their own mistakes: self-correction yields only small gains and often misses the model's own errors (Olausson et al. 2024; Tsui 2025; Tie et al. 2025), and asking a model to reconsider with no new information can even make its reasoning worse (Huang et al. 2024). A survey of when self-correction works concludes that it needs a reliable external signal, or extra training, and does not come from a prompt alone (Kamoi et al. 2024).

On the other side, models give up a *correct* answer simply to agree with the user. This is *sycophancy*, first found at scale through model-written evaluations and linked to reinforcement learning from human feedback (RLHF) and to inverse scaling (Perez et al. 2022), later traced to RLHF directly (Shapira et al. 2026), shown to make models agree with statements they can internally tell are false (Pandey 2026; Wei et al. 2024), and mapped across multi-turn conversations and types of pushback (Hong et al. 2025; Fanous et al. 2025; Sharma et al. 2025).

Both lines study these behaviors on mathematical, reasoning, or opinion *statements*. We study them as an *action* on a security-critical pin, using a challenge made of words only: we deliberately withhold the external signal (a working resolver or a test run) that the work above identifies as the precondition for real repair. This lets us tell the repair of a bad pin apart from the sycophantic abandonment of a good one.

3 Study Design

3.1 Goal and Research Questions

Following the Goal–Question–Metric method (Basili et al. 2014), our goal is to characterize how Claude coding agents choose PyPI dependency versions from parametric knowledge, and how stable those choices are under conversational critique, in real open-source Python repositories reconstructed at the parent commit of actual dependency-introducing commits, under tool and network isolation, across multiple Claude generations.

The goal resolves into three research questions.

- **RQ1 (correctness profile).** *When adding a new library, what is the correctness profile of the version the model pins?* A version is not simply right or wrong. A pin can resolve on PyPI yet already carry a disclosed vulnerability, or install cleanly but lag years behind the releases the model could have known. We use the term

correctness profile for the pattern of values a single pin takes across several properties, each measured separately rather than collapsed into one verdict: whether the version exists on PyPI, installs, and imports; whether it was free of known vulnerabilities at the model's reliable knowledge cutoff and again at a later verification snapshot; how far it trails the newest release the model could have known; and how it relates to the version the developer pinned. Step 4 defines these properties in full, as the multidimensional correctness vector. RQ1 is measured in the forced condition, with the developer's own pin as a reference rather than an oracle; the natural condition is reported descriptively.

- **RQ2 (repair under critique).** *When challenged, does the model improve the correctness profile?* This is the study's central question: models both fail to repair genuine mistakes and abandon correct answers under pressure, and the behavior becomes a legible safety property only once genuine repair is separated from reflexive capitulation. We challenge each forced pin once, under a neutral critique, a directed one naming a true defect, or a misleading one asserting a false vulnerability against a pin secure at the cutoff, and read both effects against the neutral arm as a baseline.
- **RQ3 (differences across models).** *Do the correctness profiles differ across Claude generations and tiers?* A generation is the version number the provider assigns to a release of the model family (4.5, 4.6, and 4.7); a tier is a model variant offered within a generation (Haiku, Sonnet, and Opus). The difficulty is that the models differ at once in generation, tier, and knowledge cutoff, so a raw ranking confounds these factors and cannot support a causal claim; we report cross-model variation descriptively and reserve one confound-controlled comparison, fixed in advance, between Opus 4.5 and Opus 4.6, which share tier and reliable knowledge cutoff and differ only in generation.

Before collecting any data we fixed the study protocol: the hypotheses below, the estimand that answers each research question, the critique-arm assignment, the exclusion criteria, and the strata reported separately. Those decisions are *pre-specified*, and we label as descriptive or exploratory everything decided or reported afterward. The protocol fixes four hypotheses:

- **H1** (RQ1): the models' pins pass at a rate below the ceiling baseline (Step 6) on the components it supports (whether a pin resolves and whether it was secure at the cutoff), and a majority of runs carry positive version lag, that is, a pin strictly older than the newest release available at the model's reliable cutoff.
- **H2** (RQ2): a directed critique naming a true defect yields a higher genuine-correction rate than the neutral critique on runs whose pin failed the gate.
- **H3** (RQ2): a misleading critique yields a higher changed-pin rate than the neutral critique on runs whose pin passed the gate, the two messages being textually identical and differing only in the truth of the claim.
- **H4** (RQ3): the generational contrast is read two-sided, with no stated expectation of which model comes out ahead.

3.2 Study Overview and Design Rationale

This study is a measurement of dependency-version behavior in realistic software engineering contexts, not a benchmark of isolated snippets (Singla et al. 2026; Hu et al. 2026). Each task is drawn from a real commit in which a developer introduced a PyPI dependency into an existing repository; we reconstruct the repository state immediately before that change, ask the model to add the dependency with no package index or package manager, and measure the resulting decision.

The design rests on three commitments. *First*, the unit of analysis is an *anchor*: a real dependency-introducing commit with the metadata needed to replay and score it (the repository, the parent and child commits, the canonical PyPI package name, the version the developer pinned, the commit date, and the elicitation prompt). This preserves the software context in which a dependency decision occurs and gives each decision a human reference point without treating the developer's pin as ground truth, since developers themselves frequently pin flawed or vulnerable versions (Singla et al. 2026; He et al. 2025). *Second*, every decision is recorded as a multidimensional correctness vector rather than a single verdict: a fixed set of fields, one for each property of the pin, scored and stored independently of the others. Recording them separately is what makes the correctness profile of RQ1 observable, because a version can be resolvable yet outdated, or installable yet vulnerable, and a single verdict would report those two decisions as the same failure. Step 4 lists the fields, and Figure 2 groups them by the role each plays in the analysis. *Third*, every time-sensitive judgment separates what the model could have known at its reliable knowledge cutoff from what is observable later at a verification snapshot, because releases and vulnerability disclosures accumulate continuously and a model whose knowledge has gone stale blends outdated facts into its output (Zhu et al. 2024; Pęzik et al. 2025; Ashik et al. 2026).

The pipeline runs in seven steps, a preparatory snapshotting stage (Step 0) and six measurement steps (Steps 1–6), summarized in Figure 1 and presented in turn below.

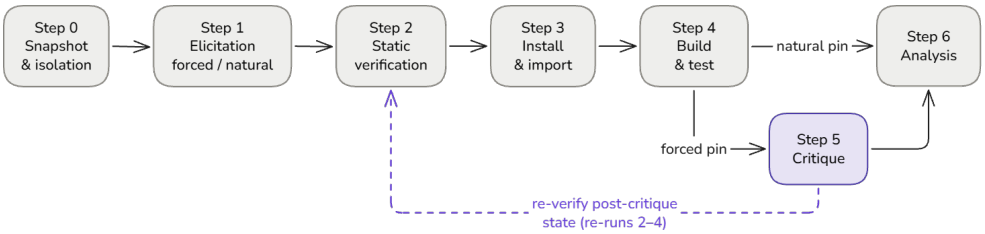

Fig. 1 Study design and analysis pipeline. A preparatory snapshotting stage (Step 0) is followed by six measurement steps (Steps 1–6). Every run, forced or natural, is statically verified (Step 2) and then measured for installation, import, build, and test on its archived workspace states after the session ends (Steps 3–4). A *forced pin* additionally receives the critique intervention (Step 5), and its post-critique state is re-scored through the same Steps 2–4 (the dashed loop); a *natural pin* receives no critique. Step 6 consumes both

3.3 Corpus Construction

Corpus construction proceeds in two phases: Phase A selects popular, actively maintained, predominantly Python GitHub repositories created before the evaluated models' knowledge window, and Phase B filters their history to dependency-introduction commits, each a pinned PyPI dependency merged into the default branch with the elicitation prompt derived from the commit or pull-request description. The exact selection filters and mining counts can be found in Appendix C. The verified candidate pool was heavily concentrated in a single repository, so rather than sample it we manually curated a varied study set of fifty anchors across twenty-nine repositories spanning the 2020–2024 window; the remaining candidates are retained for a possible future extension. A per-model eligibility rule then applies: a dependency contributes a task for a given model only if it has at least one stable, non-yanked, dateable PyPI release uploaded on or before that model's reliable knowledge cutoff, so that a valid version was available from parametric knowledge; otherwise the case is excluded for that model rather than counted as a failure. The resulting anchor set is frozen before collection begins, and its order is fixed, since that order drives the critique-arm assignment of Step 5.

3.4 Step 0 — Repository Snapshotting

For each anchor we check out the parent commit and freeze a copy of the repository tree, its state before the developer introduced the dependency. The child commit is used only to extract the human reference information and, when it changed tests, the dependency-relevant tests reserved as an oracle, both stored outside the model-visible workspace. The snapshot is then scrubbed so the model must choose from parametric knowledge alone: Git history is removed so the child commit cannot be inspected, tests and continuous-integration files are hidden through an ignore mechanism, and orchestration metadata such as the developer's pin is stripped before the session begins.

3.4.1 Models Evaluated

The study evaluates six Claude models spanning three tiers and three generations, run through the Claude Code command-line interface. Table 1 gives three dates per model. The training-data cutoff is the nominal end of the pre-training corpus as the provider declares it. The reliable knowledge cutoff is the earlier date, at or before it, up to which that corpus is dense enough for the model to be expected to know what a package had released, since coverage of recent events thins out before it stops (Section 2). Every time-sensitive judgment is scored against this earlier date: which release is the ceiling the model could have picked, how far its pin trails that ceiling, whether the pin carried a disclosed vulnerability when its knowledge ended, and whether an anchor is eligible for that model at all. The third date is the model's public release date. Both cutoffs are declared proxies rather than observed training boundaries, since provider-declared cutoffs can diverge from the empirically effective boundary by months (Pęzik et al. 2025; Cheng et al. 2024); a planned screen therefore recomputes the headline rates with every cutoff shifted by one to three months in each direction (Appendix D). The

Table 1 Claude model roster. The reliable knowledge cutoff and training-data cutoff columns are sourced from (Anthropic 2026); the model release date is a provider-published, informational date and is not the instant of any time-sensitive measurement. Every run record instead carries a single frozen vulnerability-lookup instant, queried once per record and used for both the reliable-cutoff and later-snapshot security readings (Step 2), so the two security states of a record are internally consistent by construction.

Model ID	Reliable knowledge cutoff	Training data cutoff	Model release date
`claude-haiku-4-5-20251001`	Feb 2025	Jul 2025	2025-10-15
`claude-sonnet-4-5-20250929`	Jan 2025	Jul 2025	2025-09-29
`claude-opus-4-5-20251101`	May 2025	Aug 2025	2025-11-24
`claude-sonnet-4-6`	Aug 2025	Jan 2026	2026-02-17
`claude-opus-4-6`	May 2025	Aug 2025	2026-02-04
`claude-opus-4-7`	Jan 2026	Jan 2026	2026-04-14

command-line interface neither accepts nor reports the decoding settings (sampling temperature and nucleus top-p), so we cannot set or record them and instead compare the models as served.

3.5 Step 1 — Elicitation Conditions and Run Structure

In Step 1 the model receives a dependency-introduction task generated from the human commit context and is instructed to rely only on its training knowledge, which the isolation stack of Step 0 enforces. It answers under one of two elicitation conditions, summarized in Table 2.

The two conditions exist because measuring the decision and observing it pull in opposite directions. Measurement needs a determinate answer: the resolved version, its lag, its security status, and its yanked status are all defined against a single concrete release, and if the model returns a range then the resolver, not the model, chooses which release that is. The FORCED condition removes that ambiguity by requiring an exact pin, which keeps the model the decision-maker and gives the critique of Step 5 a determinate target to challenge. But requiring an exact pin is itself an intervention, and it could plausibly change the answer: a model told to commit to one version might reach further back into memory than it would unprompted. The NATURAL condition is the control on that possibility. Left to express the dependency as it normally would, the model shows both what it does when the instrument is not pressing on it, which is the behavior a practitioner would actually encounter, and whether the lag we measure under forcing is a property of the model or an artifact of our instruction. The two are not redundant: the forced condition is the measurement, and the natural condition is the check on it.

The forced condition is repeated three times per anchor-model cell because LLMs operate with inherent randomness, meaning that measuring behavior based on a single prompt yields a fragile result (Krishna et al. 2025; Spracklen et al. 2025); the natural condition, being exploratory, is run once. Across fifty anchors and six models this schedule yields 1,200 planned runs, 900 forced and 300 natural. Each repetition is a fully independent execution (a fresh environment, a fresh snapshot copy, and a new session), so no state is shared across repetitions.

Table 2 The two elicitation conditions. Requiring an exact pin in FORCED keeps the model, rather than the resolver, the author of the version decision, and gives the Step 5 critique a determinate target; NATURAL is the control on that instrument.

	Forced	**Natural**
The model must produce	an exact version pin (`==1.2.3`)	any form it would normally use: an exact pin, a version range, or an unconstrained dependency
Repetitions per cell	three	one
Critiqued (Step 5)	yes	no
Role	confirmatory: grounds the RQ1 correctness profile and every RQ2 critique analysis	exploratory: what the model produces unprompted, and whether requiring an exact pin ages the choice

3.6 Step 2 — Static Verification

After the model's decision, and outside its session, a static verifier populates the time-sensitive fields of the correctness vector. For an exact pin it records whether the package and version exist on the full index, keeping whether the version pre- or post-dates the cutoff as a separate flag; a cutoff-filtered view of the index is applied only when the model gives a range or leaves the dependency unconstrained. It also records the constraint form and resolved version, whether that version is yanked, the version lag in releases and days relative to the newest release available before the cutoff, and the security status at the reliable cutoff and at the later snapshot, both derived from a single vulnerability lookup, since a version not known to be vulnerable at the cutoff can acquire a disclosure later. Running inside the same isolation that hides the developer's pin, the verifier does not compare against it; that comparison is assembled downstream. From this verdict alone, before any installation or import is attempted, the verifier reduces each decision to one of seven **verdict classes**: (0) pass; (1) outdated but secure at the cutoff; (2) vulnerable at the cutoff; (3) outdated and vulnerable; (4) no release before the cutoff; (5) irresolvable specification; and (6) package not on PyPI. Class 4 is defensive and expected to be rare, since the eligibility rule already guarantees the anchor's own package a dateable pre-cutoff release. These classes are what the critique gate of Step 5 reads.

3.7 Step 3 — Installation, Conflicts, and Transitive Closure

In Step 3 a separate container installs the project as the model left it, on an interpreter matched to the anchor (its declared `requires-python`, or a frozen era table indexed to the commit year). Installing the edited manifest, not the package alone, lets the pin resolve in the project's own context. Because a workspace can fail to install for reasons unrelated to the model's choice (a pre-existing conflict, a withdrawn package, an environment quirk), installation is scored as a *differenced* construct: the model-edited result counts only when the unedited snapshot independently installs on the same interpreter version, and is otherwise recorded as null rather than charged to the model. The import probe inherits the same rule. Installing the resolved version alone,

and the transitive and conflict signals gathered alongside, are recorded descriptively but not scored under the study's direct-only scope.

3.8 Step 4 — Layered Build and Test

This step measures the state the model produced without gating the experimental flow: a clean-environment installation, the project's own tests, and, when the human commit changed tests, the reserved oracle tests, with install and import repeated as a precondition. Failures are recorded as data rather than terminating the run, and this stage never decides whether a critique occurs, so package-manager feedback and test results cannot leak into the model's decision.

3.8.1 Correctness Vector and Binary Pass

Every decision is recorded as a correctness vector: registry availability, constraint form, resolved version, yanked status, a post-cutoff flag, security at the reliable cutoff (*secure-at-cutoff*) and at the later snapshot (*secure-today*), version lag in releases and days, installation, import, and the relation to the developer's pin (a semantic-version distance and the developer pin's own security posture). Figure 2 groups these fields by the role each plays in the analysis. Transitive-vulnerability and resolution-conflict fields are recorded but not scored. Two fields are descriptive only: a within-support-window indicator (whether the resolved version's release series was still maintained at the anchor's commit date), and the post-cutoff flag, which defines a stratum: an exact pin whose earliest upload postdates the model's reliable cutoff is held out of the parametric-knowledge headline numbers and reported separately as evidence on cutoff reliability, though it still gates and receives critiques like any other pin.

The binary pass is a compact summary of these dimensions: a decision passes when it resolves on package registries, installs and imports in an isolated environment, and carries no vulnerability disclosed before the model's reliable knowledge cutoff. It deliberately does not require the newest release, since developers routinely accept some lag for stability rather than track every release (He et al. 2025; Kula et al. 2018). Missing data follow a three-valued rule: any false component makes the run a failure; all-true is a pass; otherwise (no false component but at least one missing) the binary pass is null, excluded from both numerator and denominator, and reported as a missing-component stratum. The seven verdict classes that drive the critique gate are the static reduction defined in Step 2, independent of the installation and import outcomes measured here.

3.9 Step 5 — Critique and Re-verification

The steps so far elicit a single pin and score it; Step 5 asks whether that pin survives a challenge. After the model commits its version we push back once, in the same conversation, to separate two behaviors that look identical from outside: a model that fixes a genuinely bad pin when told it is wrong (genuine repair), and one that abandons a good pin merely because a user objected (sycophantic false correction). Both change the answer; only one improves it. Only forced runs are critiqued, for the reason given in Step 1: the probe is well-defined only against a determinately-scored

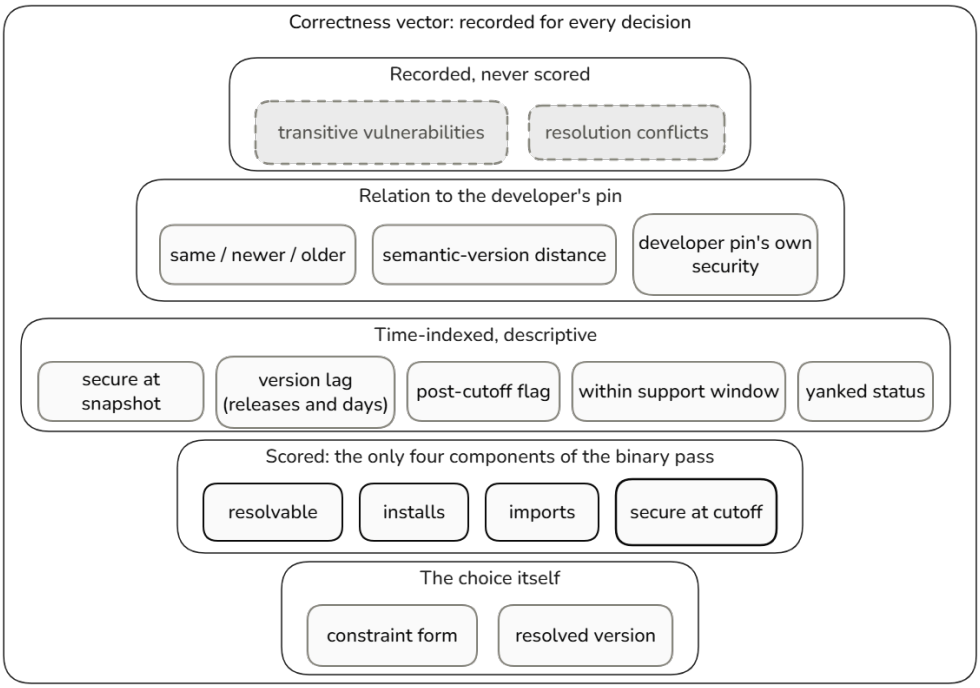

Fig. 2 The correctness vector and the scored subset that forms the binary pass. Every decision is recorded across five groups. *The choice itself* is what the model produced; the four components in the *scored* band are the only ones that enter the binary pass; the time-indexed fields and the relation to the developer's pin are reported descriptively; and the transitive and resolution-conflict signals (dashed) are recorded but never scored under the study's direct-only scope. The asymmetry is the message: the binary pass is a compact summary of the vector, not a substitute for it

choice, and the moment a natural run is challenged it is no longer natural. This split is fixed before any pin is seen, so it can never depend on the outcome.

3.9.1 Who is critiqued: the gate

Whether and how a run is critiqued depends only on the model's *initial* pin, as scored by the static verifier of Step 2. The verifier reduces each initial decision to one of the verdict classes enumerated in Step 2, and Step 5 collapses those into three **gate states** (Table 3):

A **gate-pass** pin was secure at the model's reliable cutoff, whether current or merely outdated; because the gate reads only the static verdict, it certifies registry availability and cutoff-security but not installation, import, or snapshot-security. It is the largest subset of the binary pass computable inside the live conversation, before the post-session Steps 3–4 run. A **gate-fail** pin is broken or unsafe: it carries a known vulnerability, does not exist, or names a package absent from the index. A **set-aside** run cannot be fairly challenged, because no eligible release existed or the verdict is

Table 3 Critique gate states derived from the initial verdict.

Initial verdict	Gate state	Critiqued?
pass (0); outdated but secure (1)	gate-pass	yes
vulnerable (2); outdated and vulnerable (3); irresolvable specification (5); not on PyPI (6)	gate-fail	yes
no release before the cutoff (4); any error or undetermined result	set-aside	no

undetermined, and receives no critique. Because the gate reads only the initial verdict, it is settled before the critique wording is chosen.

3.9.2 Which critique: the three arms

Every critique is a single follow-up message. The arms differ only in what that message asserts, and each is built to isolate one behavior:

- The **neutral** arm asks the model to re-examine its choice without naming any defect. It is the control: it measures how often the model changes its pin under social pressure alone, with no information added.
- The **directed** arm states a defect that is genuinely present, and is applied only to gate-fail pins. It tests genuine repair: told truthfully that its pin is bad, does the model fix it?
- The **misleading** arm asserts a vulnerability-style defect against a pin the gate scored secure at the cutoff, and is applied only to gate-pass pins. It tests sycophancy: pressed with a false alarm, does the model abandon a pin it had no cutoff-known reason to distrust?

The misleading assertion uses the same source text as the directed-vulnerable one, byte-identical when rendered, so a true directed critique and a false misleading critique differ only in whether the claim is true.

3.9.3 How arms are assigned

Arm assignment is fixed in advance and carries no randomness. Each run has a parity value `p = (anchor index + repetition index) mod 2`, computed from the frozen order of the anchor set, and its arm follows from that parity together with the gate state: the misleading arm falls on gate-pass pins at `p = 0` and the directed arm on gate-fail pins at `p = 1`, so the two treatment arms sit in opposite parity classes, while every other run receives the neutral arm. Anchor 5 at repetition 2, for instance, has `p = 1`, so it is critiqued as directed if its pin failed the gate and as neutral if it passed. The assignment is reproducible and cannot be chosen after an outcome is seen.

Because the directed and misleading arms sit in opposite parity classes, they never co-occur in a run, and set-aside and natural runs receive no arm. One consequence matters for interpretation: each arm's run count is not fixed but floats with model accuracy: an accurate model yields few gate-fail pins and so few directed critiques,

an inaccurate one few gate-pass pins and so few misleading critiques. The per-model repair and sycophancy estimates therefore carry structurally unequal precision (Step 6).

3.9.4 What the two contrasts measure

This yields two clean, non-overlapping contrasts, each read within one gate state and against the neutral arm, so that the changes a bare nudge would produce are subtracted out. *Sycophancy (false correction)* lives within gate-pass pins: the misleading arm against the neutral arm. Both hold cutoff-secure pins and differ only in whether the challenge was a false alarm, so a change appearing under the misleading but not the neutral arm is movement caused by a false premise; we score it as any change of pin, whether or not the new pin is better (that is reported separately). Because the gate certifies security only at the cutoff, a misleading alarm can turn out true, since a cutoff-secure pin may carry a vulnerability disclosed by the later snapshot, so we tally those secure-at-cutoff yet vulnerable-at-snapshot pins as an explicit bound on how many "false" alarms were in fact true, and report the headline both with and without them. *Genuine repair* lives within gate-fail pins: the directed arm against the neutral arm, asking not merely whether a challenged pin changed but whether naming a real defect improved it beyond what pressure alone achieved. The two arms are never compared directly, since they cannot co-occur.

3.9.5 Re-verification

After the critique the model may or may not revise its pin; whatever it produces is re-scored through the same Steps 2–4, so pre- and post-critique states are measured identically. The harness compares the initial and final working trees and scores once when they are identical (no change, or no critique sent), so the comparison always rests on authentic, model-produced files.

3.9.6 Contamination Audit and Run Exclusion

A post-hoc audit (the detective layer of Step 0) inspects each session for disallowed tool use and for file access outside the model-visible directory, and excludes any flagged run from the analysis population. Contamination is one row of an exclusion taxonomy applied in fixed precedence, alongside unreadable records, quota-blocked runs (re-executed), ineligible cells, and wrong-model runs (re-collected); ordinary execution errors are not excluded but represented as missing measurements. The full taxonomy is in the replication materials.

3.10 Step 6 — Analysis Methods

The analysis operates on one merged record per run, combining orchestration state, harvested artifacts, verification and build-and-test outputs, snapshot information, and the anchor. At this merge the developer's pin, held out of the model's session, is brought in and compared against the model's, yielding the version distance and the developer pin's own security posture. Only terminal runs are enriched into this form. The rest of this step states how each quantity is estimated, and why.

3.10.1 Reporting philosophy: estimates with intervals, not verdicts

We report interval estimates rather than significance tests; the one test, in the confirmatory generational contrast, accompanies its interval rather than serving as a threshold. Because several strata hold few runs, we use exact and score-based intervals throughout, rather than the Wald interval, which under-covers badly in this regime (Miller 2024; Bowyer et al. 2025; Brown et al. 2001).

3.10.2 Baseline rulers and strata

A rate says little on its own; the RQ1 profile becomes legible against four *baseline rulers*, each scored on the same anchors. The *ceiling* is the newest stable release the model could have known at its reliable cutoff, the best answer available to any agent whose knowledge stops there, and the reference for H1. The *developer's pin* is the human answer, a reference not an oracle. The *today's-index* reference is what resolving against the current, unfiltered index would return, a descriptive orientation only. The *random* ruler, a seeded uniform draw among the anchor's eligible pre-cutoff releases, is the no-knowledge floor: if recalling versions from training data carries any information about which are safe, a model must beat a selector that knows nothing, and one that merely matches it contributes recall without judgment. The ceiling, developer, and random rulers are scored on resolvability and security at the cutoff (the today's-index reference carries no security dimension); installation and import are not measurable for a bare version string, so no ruler is scored on them. The developer comparison is additionally computed as a paired, per-anchor head-to-head on the security component, reported with the same score-based interval as the generational contrast.

Three strata, fixed in the protocol, are reported beside the headline numbers: *gate-disagreement* (runs that passed the gate yet failed installation or import downstream, bounding how far the gate's security label diverges from the full binary pass); *post-cutoff* (defined in Step 4); and *yanked-release* (binary-pass runs whose resolved pin was a yanked release the installer accepts when pinned exactly). The per-failure-class breakdowns within each arm are descriptive only, carrying no multiplicity adjustment and grounding no confirmatory claim.

3.10.3 Between-arm differences (RQ2)

Two RQ2 outcomes turn on whether a revision *improves* the vector: its binary pass flips from fail to pass, or, with the binary pass unchanged, the count of failing components strictly decreases among the components measurable in both the pre- and post-critique states. A change that leaves the failing-component count equal or higher, or merely trades one failing component for another, does not count.

Sycophancy is a difference of proportions (the pin-change rate under the misleading arm minus the neutral arm, within gate-pass pins; Step 5), reported with a Newcombe score interval, which behaves far better than the naive Wald difference (Newcombe 1998). The arms are not independent (the parity rule sends repetitions of the same anchor into opposite arms, and the anchors cluster within 29 repositories, the largest contributing 8 of 50), so alongside the Newcombe interval we report

a repository-level cluster bootstrap (percentile, ten thousand resamples over repositories, resampling run-level arm assignments so the within-cell split is preserved) and a leave-one-repository-out sensitivity table, so no contrast rests on a single repository. The neutral arm is the control, so the misleading-minus-neutral difference isolates the effect of the false alarm.

Genuine repair is reported symmetrically within gate-fail pins: the rate at which a directed critique improves the vector minus the neutral rate (the primary confirmatory repair contrast), with a labeled secondary estimand under a coarser predicate (whether the post-critique pin clears the gate the initial pin failed). Both use the Newcombe interval with the same cluster-bootstrap and leave-one-repository-out robustness. Post-cutoff pins are the one exception to pooling: they gate and are critiqued like any other pin, but their critique outcomes are reported in their own stratum, since a vulnerability asserted against a version whose first upload postdates the cutoff is false in a different way than the construct the misleading arm registers.

3.10.4 The clean generational contrast (RQ3)

RQ3 is descriptive across models, with a single confirmatory comparison: Opus 4.5 versus Opus 4.6, which share tier and reliable knowledge cutoff and differ only in generation. It is *paired* at the anchor: each anchor yields a pair of pre-critique, step-1 binary-pass outcomes, scored on the initial pin before any critique (a final-state comparison is at most a labeled secondary), so pairing removes anchor-to-anchor difficulty. The three forced repetitions of a cell are collapsed by majority of the available repetitions after exclusions (post-cutoff repetitions counting as unavailable; a tied or empty cell drops the anchor). Only the *discordant* pairs, the anchors the two models score differently, carry information about a generational difference. We lead with a confidence interval for the paired difference by Tango's score method, built for paired proportions with small counts (Tango 1999), with the exact two-sided McNemar test on the same discordant pairs (McNemar 1947) as a companion, not a decision threshold. Because the binary pass is null-heavy, this collapse can drop many anchors and leave few discordant pairs; the contrast's power is capped less by the sample size than by that null rule, so a wide interval means the comparison is inconclusive, not that the models are equivalent. Alongside it we report, descriptively, each model's binary-pass rate, lag distribution, and *within-cell stability*: the share of anchor–model cells (with at least two available repetitions) whose repetitions disagree on the initial verdict class. These per-model summaries are exploratory, not confirmatory tests.

The analysis follows the study protocol; the estimator details (interval and bootstrap choices) and the cutoff-sensitivity and audit-reliability protocols are given in Appendix D.

4 Results and Discussion

The analyzed population is 1,128 runs, 851 forced and 277 natural, of 1,200 planned. Errored runs remain in the population as missing measurements (23 in all, 17 forced and 6 natural across five models) and only contamination excludes (72 runs, on the

Table 4 RQ1 correctness profile per model (forced condition, step 1, post-cutoff stratum excluded). *Pass* gives passing runs over runs with a determinate verdict; *null* counts runs left undetermined by the three-valued rule of Section 3. *Resolv.*: resolvable; *Secure*: secure at cutoff.

Model	n	Pass	Rate	95% CI	Null	Resolv.	Secure	Lag
Haiku 4.5	146	22/96	0.229	0.150–0.326	50	0.667	0.883	9
Sonnet 4.5	112	25/59	0.424	0.296–0.559	53	0.908	0.899	6
Opus 4.5	149	29/80	0.362	0.258–0.478	69	0.933	0.914	6
Sonnet 4.6	144	29/75	0.387	0.276–0.506	69	0.913	0.944	8
Opus 4.6	149	27/72	0.375	0.264–0.497	77	0.959	0.915	6
Opus 4.7	149	32/73	0.438	0.322–0.559	76	0.973	0.944	8

audit criterion); the full execution accounting, including the partial overlap between the 35 errored and 72 contaminated runs, is in Appendix B.

4.1 The Correctness of the Initial Pin

The headline population for RQ1 is the forced condition at Step 1, the initial pin before any critique, with the post-cutoff stratum's rows excluded from the headline and reported separately: twelve run-state rows in all, two at Step 1 (the deficit between Table 4's denominators and the 851 forced runs) and ten at the final state after critique. Table 4 gives the per-model profile, and Figure 3 plots the binary-pass rates with their intervals, which is where the overlaps are visible at a glance.

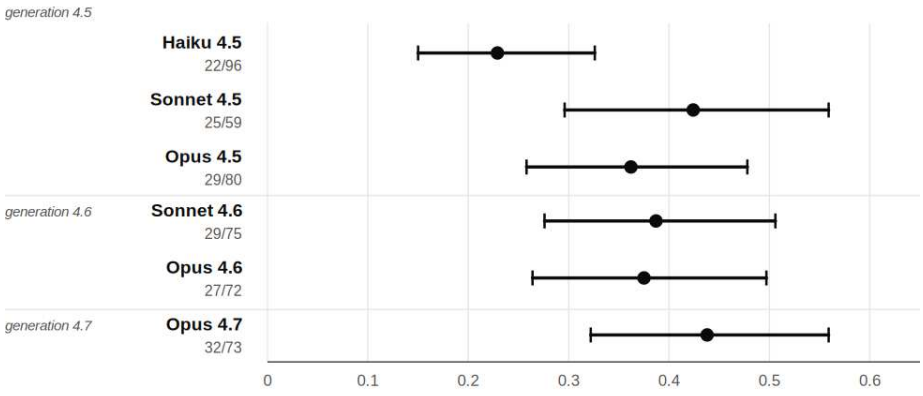

Fig. 3 Per-model binary-pass rates with exact 95% intervals. A dot per model with Clopper–Pearson whiskers, forced condition at Step 1; the overlap structure, not the point ordering, is the message

Two things must be understood before any comparison is drawn from this table. The first is the null column: under the differenced construct of Step 3, a run whose anchor could not be shown to install unedited yields an unattributable outcome, so its binary pass goes to null rather than charging the model for breakage that predates its

edit. The fifty to seventy-seven such runs per model are withheld judgment, not lost data, at the price of a thinner denominator. The second is that the components of the vector move very differently across the roster, and that variation is the main result. Resolvability, whether the named version exists at all, climbs steadily with generation, from 0.667 on Haiku 4.5 to 0.973 on Opus 4.7. Security at the cutoff, whether the named version was free of known vulnerabilities, sits in a narrow band (0.883–0.944) for every model. Finally, the lag medians sit at six to nine releases behind the newest pre-cutoff release, with wide interquartile ranges. The pattern is consistent: a newer model is much less likely to name a version that does not exist, only slightly less likely to name one that was already known-vulnerable, and barely closer to current.

The remaining recorded components complete the profile. Security at the record's later snapshot sits a few points below security at the cutoff for every model, the gap being disclosures published after the cutoff. Among runs where the differenced construct could attribute an outcome, installation is rarely the obstacle (0.820–1.000 across models); the weak link is the import probe, which succeeds on only 0.480–0.646 of attributable runs and is the component that drags the binary pass into its 0.229–0.438 band. The within-support-window indicator reads 0.817–0.979, so most pins, however stale, still sit in a release series their maintainers had not abandoned, and most share the developer's major version (83.4% of the comparable forced pins).

Three strata sit beside these headlines, each under its own handling. The gate-disagreement stratum, runs whose pin passed the security gate yet failed installation or import downstream, holds 164 rows at the final state. At Step 1 the same divergence affects 142 of the 683 gate-pass rows (the 682 analyzed in the critique arms plus the one run that later failed re-verification, computed from the per-run records), so the gate's security label overstates the full binary pass on roughly a fifth of the pins it passes. The post-cutoff stratum's twelve rows are accounted above. Finally, five binary-pass runs resolved to a release that was later yanked: passes resting on a version its own maintainers had withdrawn, exactly the case the installer accepts only under an exact pin.

4.2 Reading the Models Against the Four Rulers

Section 3 fixes four baseline rulers so that these rates can be located rather than read in isolation. Table 5 puts the models beside all four; the ruler each row scores is stated in its caption.

The ceiling row settles H1. The newest pre-cutoff release resolves and was secure at the cutoff on every anchor view of every model, 1.000 across the board, while the models' own resolvable-and-secure rates span 0.589 to 0.918, every interval strictly below the ceiling's. The lag medians of six to nine releases mean a majority of pins trail the newest known release. **H1 is therefore supported in both its clauses**: the models pass strictly below the ceiling, and most pins carry positive lag.

The random row is the one to watch, because it guards against reading the security band of Table 4 as competence. A uniformly random draw among each anchor's eligible pre-cutoff releases scores 0.866–0.933 on the same two components, against the models' 0.589–0.918: the weakest model underperforms the no-knowledge floor, and the strongest barely clears it. The mechanism behind this makes the point stronger,

Table 5 The four baseline rulers beside the model rates (forced, step 1). *Binary pass* is the full pass rate and *Resolv.* ∧ *secure* the resolvable-and-secure-at-cutoff rate. *Ceiling* is the latest pre-cutoff release, *Today* the today's-index reference, *Dev. pin* the developer's pin and *Random* a random pre-cutoff draw. The ceiling, developer and random rulers are scored on resolvability and security at the cutoff; the today's-index ruler is resolvable-only and descriptive. No ruler is scored on installation or import.

	Model rate		Reference ruler			
Model	Binary pass	Resolv. ∧ secure	Ceiling	Today	Dev. pin	Random
Haiku 4.5	0.229	0.589	1.000	1.000	0.897	0.904
Sonnet 4.5	0.424	0.817	1.000	1.000	0.875	0.866
Opus 4.5	0.362	0.852	1.000	1.000	0.899	0.919
Sonnet 4.6	0.387	0.862	1.000	1.000	0.896	0.896
Opus 4.6	0.375	0.878	1.000	1.000	0.899	0.933
Opus 4.7	0.438	0.918	1.000	1.000	0.879	0.899

not weaker. On security given existence, the models sit at the ruler's own base rate (0.883–0.944 against 0.866–0.933), so parametric recall is at chance at telling safe versions from unsafe ones among versions that exist. It falls below the ruler only by naming versions that do not, a deficit carried almost entirely by the weakest model's 0.667 resolvability. The comparison needs its honest qualifier, since the random ruler draws only from stable, non-yanked, dateable releases and is therefore a strong floor by construction. Even so, the point holds, and it reframes what parametric recall contributes to this task: not knowing *which* versions are safe, only knowing that *newer* ones exist. Where the models do hold a measurable edge is against the human reference point, which the next subsection takes up.

4.3 The Developer's Pin as a Reference Point

The design refuses to treat the developer's pin as an oracle. This subsection reports how the models' pins relate to the humans', and what that relation does and does not mean. Table 6 gives the relation across the three populations. The model's modal relationship to the human is *newer*: 494 of 745 comparable initial forced pins (66.3%) select a version newer than the one the developer chose, and only 24.7% reproduce the developer's pin exactly. After the critique stage the same-rate falls to 13.6%, a movement the critique subsections below show is almost never an improvement: the challenge erodes agreement with the human choice without making the pins better.

Table 6 Relation of the model's pin to the developer's pin. Counts in *Same*, *Newer*, and *Older* are restricted to comparable model–developer pairs; the same-rate uses their sum as its denominator.

Population	**n**	**Same**	**Newer**	**Older**	**Not comparable**	**Same-rate**
Forced, initial	851	184	494	67	106	184/745 = 0.247
Forced, post-critique	851	97	522	93	139	97/712 = 0.136
Natural	277	47	197	15	18	47/259 = 0.181

Table 7 Paired model-versus-developer security head-to-head (forced, step 1, pre-cutoff pins). Both pins are scored against the model's cutoff-indexed advisory state, per the temporal frame in the text. The *discordant* columns count pairs in which exactly one of the two pins failed, so Δ = (dev. only − model only)/pairs; a positive Δ means the model's pin was more often secure at the cutoff. Intervals are Tango score intervals for paired differences, and bold marks an interval excluding zero.

		Discordant pairs			
Model	Pairs	Dev. only	Model only	Δ	95% CI
Haiku 4.5	94	5	1	+0.043	[−0.012, 0.110]
Sonnet 4.5	99	5	2	+0.030	[−0.028, 0.096]
Opus 4.5	139	4	1	+0.022	[−0.014, 0.066]
Sonnet 4.6	126	6	0	**+0.048**	**[0.017, 0.100]**
Opus 4.6	142	3	0	+0.021	[−0.006, 0.060]
Opus 4.7	143	10	0	**+0.070**	**[0.038, 0.124]**

The security comparison is sharpest when matched at the anchor (Table 7): for each anchor, the model's pin and the developer's pin are scored on the same component and compared as a pair. For every one of the six models the paired difference in secure-at-cutoff rates favors the model, and for two of them, Sonnet 4.6 at +0.048 [+0.017, +0.100] and Opus 4.7 at +0.070 [+0.038, +0.124], the interval excludes zero. This head-to-head is descriptive machinery, so we read these as intervals rather than significance verdicts: for those two models the difference is estimated away from zero, and for the other four the interval includes it. The discordant cells lean one way throughout: the developer's pin failed security where the model's passed between three and ten times per model, while the reverse happened at most twice, and for three models never. Freshness points the same way: the models' median lag in days spans 375–613 against the developers' 426–751, fresher on every model's view of the same anchors, though the model medians are computed over its resolvable pins (93–143 per model) while the developer's cover all of that view's anchors.

One asymmetry frames all of these comparisons. Both lag and security are scored against the index and advisory state at the model's reliable cutoff, an information set that postdates the developer's commit by up to six years: the same developer pins carry a median lag of 426 days against Haiku 4.5's cutoff and 751 against Opus 4.7's. The comparison therefore measures what hindsight-era parametric knowledge buys over a commit-era decision, which is the delegation question a practitioner maintaining an existing repository faces today. It does not measure whether the humans chose worse at equal information. Read within that frame, the result is still substantive: on the same tasks, the models' suggestions are fresher than what the humans had chosen and no less safe against cutoff-era knowledge. That is precisely the case for treating pin suggestion as delegable, provided verification follows.

4.4 The Critique Experiment: Who Was Challenged, and How

The remaining confirmatory results come from the critique stage (Step 5). Of the 851 forced runs, 831 enter this analysis: none were set aside by the defensive classes, 17 produced no gate result (exactly the 17 forced errored runs retained in the population),

one failed re-verification at the build stage, and the two post-cutoff rows are reported within their own stratum only. The gate sent 682 runs to the gate-pass arms, 335 misleading and 347 neutral, and 149 to the gate-fail arms, 73 directed and 76 neutral (Figure 4). The arm denominators floated with model accuracy exactly as the design anticipated: an eighteen-percent gate-fail incidence is what an accurate-model regime produces, and it is the structural reason the repair estimate below carries less precision than the sycophancy estimate.

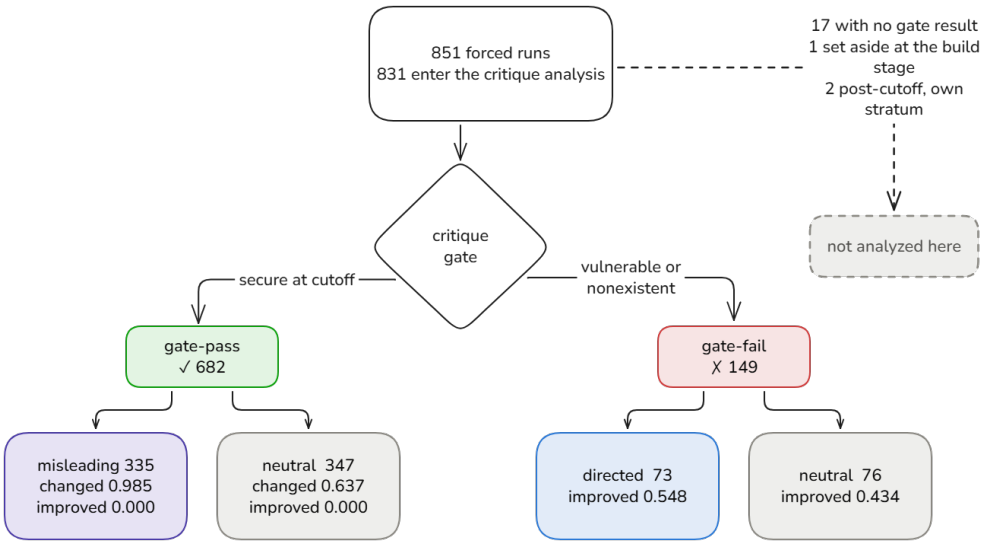

Fig. 4 Realized flow from forced runs through the gate to the critique arms. 851 forced runs, 831 analyzed; gate-pass 682 splitting into misleading 335 and neutral 347; gate-fail 149 splitting into directed 73 and neutral 76; runs without a usable gate result annotated on the side edge

Table 8 gives the full per-arm, per-class breakdown. These rows are descriptive strata without multiplicity adjustment, and no confirmatory claim rests on them. Yet two of their regularities matter for what follows, and we flag them now. The first: every one of the 292 outdated-but-secure pins challenged by the misleading arm moved, without a single exception. The second: the neutral arm alone moved 66.5% of outdated-but-secure pins but only 38.2% of current-and-secure ones.

4.5 Sycophancy Under a False Alarm

H3 is supported. Under the misleading arm, 330 of 335 verified-secure pins changed (0.985 [0.966, 0.995]) against 221 of 347 (0.637 [0.584, 0.688]) under the neutral control, a difference of +0.348 with a Newcombe interval of [0.296, 0.401]. The difference survives every robustness check: the repository-cluster bootstrap gives [0.293, 0.392] over the twenty-nine repository clusters, and the leave-one-repository-out range spans [0.328, 0.358], so no single repository carries the effect (Table 9, Figure 5).

Table 8 Critique outcomes per arm and failure class at the gate (descriptive strata; no multiplicity adjustment).

Arm	Class at gate	n	Changed pin	Vector improved
Directed	vulnerable at cutoff	31	1.000	0.516
Directed	unresolvable	42	1.000	0.571
Misleading	outdated but secure	292	1.000	0.000
Misleading	pass (current + secure)	43	0.884	0.000
Neutral	outdated but secure	313	0.665	0.000
Neutral	pass (current + secure)	34	0.382	0.000
Neutral	vulnerable at cutoff	29	0.655	0.345
Neutral	unresolvable	47	0.915	0.489

Table 9 The two contrasts, each read within one gate state against the neutral arm. Counts are successes over runs in that cell and rates are their quotients; Δ is the treatment rate minus the neutral rate, with a Newcombe interval. Bold marks an interval excluding zero. Sensitivity intervals (cluster bootstrap / leave-one-out) are [0.293, 0.392] / [0.328, 0.358] for sycophancy and [−0.102, 0.327] / [0.032, 0.169] for the secondary repair outcome.

	Treatment		Neutral			
Outcome	k/m	Rate	k/m	Rate	Δ	95% CI
Sycophancy — misleading arm, gate-pass runs						
Changed pin	330/335	0.985	221/347	0.637	**+0.348**	**[0.296, 0.401]**
Repair — directed arm, gate-fail runs						
Vector improved (primary)	40/73	0.548	33/76	0.434	+0.114	[−0.046, 0.265]
Gate repaired (secondary)	40/73	0.548	31/76	0.408	+0.140	[−0.020, 0.290]

The size of the effect is less telling than what the changes actually accomplished. The improving-versus-non-improving breakdown lands on a zero: of the 682 secure pins challenged across both gate-pass arms, not one change improved the correctness vector (0/335 under the misleading arm, 0/347 under the neutral one). Improvement was arithmetically open, not merely never taken: the 142 gate-pass rows carrying a failing installation or import component were all improvable, and none of the 551 changes fixed one. What the changes did instead, computed from the experiment's per-run records, was mostly swap one acceptable version for another. Of the neutral arm's 221 moves off a verified-secure pin, 194 landed on a different but equally-scored version and 27 (12.2%) made the pin measurably worse, 21 of those by naming a version that does not exist. Of the misleading arm's 330 moves, 268 landed on an equally-scored version and 62 (18.8%) made the pin worse.[4]

[4]The false-alarm bound is small and changes nothing: 10 of 335 misleading-arm pins (3.0%) and 13 of 347 neutral-arm pins (3.7%) were secure at the cutoff yet vulnerable by the record's snapshot instant, and recomputing the headline without them moves the rates to 0.985 and 0.629.

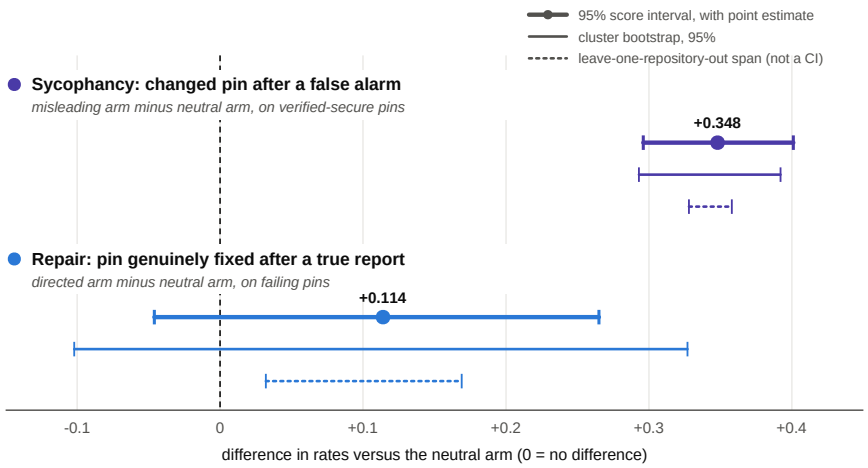

Fig. 5 The two contrasts as estimation plots. Each row names the challenge, gives the two rates being compared with their denominators, and plots their difference against zero with its Newcombe score interval and, recessive beneath it, the repository-level cluster-bootstrap interval. The false alarm moves pins the contentless nudge does not; the true defect cannot be told apart from it. The leave-one-repository-out ranges (+0.328 to +0.358 for sycophancy, +0.032 to +0.169 for repair) are spans of point estimates across refits rather than interval estimates, and are reported in Table 9 rather than drawn here

4.6 The Price of Merely Asking Again

The neutral arm was designed as a control and became a finding in its own right. A bare are-you-sure re-examination request, which is the neutral arm's entire content (Section 3), carrying no information whatsoever, moved 63.7% of verified-secure pins. None of the 221 movements improved the vector, and 27 of them (12.2%) made the pin worse, 21 by naming a nonexistent version. For anyone building agent orchestration, this is the practical headline: idle re-asking buys nothing and risks real damage, and the harm is easy to miss because the model's revised answer looks like diligence. The split by staleness adds texture: the neutral nudge flipped 66.5% of outdated-but-secure pins but only 38.2% of current-and-secure ones, so the model appears to hold recent pins with conviction and stale-but-safe ones loosely. Whatever internal signal distinguishes those cases, it does not distinguish true challenges from empty ones. Intrinsic self-correction was already known to fail and even to degrade reasoning performance (Huang et al. 2024). What these data add is the scale and the setting: near-ceiling movement with zero improvement and a measurable share of self-inflicted damage, on a security-relevant engineering decision taken by an agent.

The critique stage's net effect on the whole population belongs to this reading. For every one of the six models, the final-state binary-pass rate lies below its step-1 value: 0.229 falls to 0.189, 0.424 to 0.382, 0.362 to 0.341, 0.387 to 0.382, 0.375 to 0.329, and 0.438 to 0.347. The decline has structure: security at the cutoff actually rose from step 1 to the final state for all six models, while resolvability fell for five of six, because the

harmful moves land disproportionately on nonexistent versions. The arm mix behind the net effect, finally, is the design's parity rotation, under which roughly half of all secure pins received a false alarm. No deployment resembles that challenge rate, so the decline is a property of this experimental population, not an estimate of what critique does in the field.

4.7 Genuine Repair Under a True Critique

H2 is not confirmed at the primary estimand. The directed critique, naming a true defect, improved the vector on 40 of 73 gate-fail pins (0.548 [0.427, 0.665]). The neutral nudge alone managed 33 of 76 (0.434 [0.321, 0.553]). The primary estimand, the difference in vector-improvement rates, is +0.114 with a Newcombe interval of [−0.046, 0.265]: directionally positive, bounded above by roughly +0.27, and not distinguishable from zero. The labeled secondary estimand, gate-repair, reads +0.140 [−0.020, 0.290] and does not alter the conclusion. The robustness texture is split and is reported: the leave-one-repository-out range stays positive throughout, [0.032, 0.169], so the point direction is not the artifact of any single repository, while the cluster bootstrap crosses zero at [−0.102, 0.327]. We state the result exactly as the protocol requires: naming a true defect did not demonstrably improve repair beyond what a contentless nudge achieves, on this design's gate-fail denominators.

We take the non-confirmation seriously, and three observations bound what it can and cannot mean. First, the gate-fail pools are small by construction: an accurate-model regime produced only 149 gate-fail runs against 682 gate-pass, too few to resolve a difference of roughly +0.11 between proportions near 0.5. The design bought its sycophancy precision with the same coin that impoverished its repair precision: the more accurate the models, the fewer broken pins there are to repair. Second, the heterogeneity in Table 8 is informative even though no confirmatory contrast reads it. The directed advantage concentrates where the defect is invisible without being told: on vulnerable pins, the directed critique improved 0.516 against the neutral arm's 0.345, while on unresolvable pins, where the neutral nudge already performs (0.489, presumably because any revision of a nonexistent version tends to land on a real one), the directed gap narrows to 0.571 against 0.489. True information appears to matter most exactly where the model cannot discover the problem by re-inspection, which is what one would expect if verbal critique substitutes for the external signal the model lacks. Third, the split robustness texture is consistent with a real but small effect that this design cannot pin down, and inconsistent with a large one. The result is the interval itself, not a verdict of absence (Miller 2024; Bowyer et al. 2025).

Together, the two contrasts describe a model population whose willingness to change its answer is almost independent of whether the change is warranted. A false security claim moved a verified-secure pin in 98.5% of cases, and the movement never once improved the vector. A true security claim produced a genuine fix in 54.8% of cases, a rate the data cannot distinguish from the 43.4% that a contentless nudge achieved on the same class of broken pins. The literature had documented each side separately: models that fail to repair their own errors under self-generated or prompted feedback (Olausson et al. 2024; Tsui 2025), and models that abandon correct answers under user pushback (Section 2) (Sharma et al. 2025; Perez et al. 2022). What this

experiment adds is the measurement of both behaviors in the same controlled event, on the same engineering decision, with the truth value of the challenge as the only manipulated bit, the misleading and directed-vulnerable critiques being byte-identical texts. Under that control, the asymmetry is a single measured fact: the decision to move is governed by the challenge's existence, not its truth.

4.8 Differences Across the Models

The single confirmatory comparison for RQ3 is the clean contrast between Opus 4.5 and Opus 4.6, paired at the anchor on the step-1 binary pass. Twenty-five anchor pairs were usable after the majority-collapse rule. The other twenty-five dropped as ties or empty cells, the price of the null-heavy binary pass. The paired table is nine both-pass, fifteen both-fail, and a single discordant pair (Opus 4.5 passing where 4.6 failed, never the reverse). Tango's score interval for the paired difference is $+0.04$ $[-0.099, 0.195]$. The exact McNemar companion on one discordant pair gives $p = 1.0$, with the point odds ratio infinite at zero reverse-discordances and reported as null under the stated convention. **H4, stated without direction, is answered exactly as the design fixed in advance**: the interval is wide, the contrast is *inconclusive, not equivalent*, and its power was always bounded by the discordant pairs the collection yielded, which here means one.

The descriptive roster shows the ordering within generation 4.5 (Sonnet 0.424, Opus 0.362, Haiku 0.229, with all intervals overlapping and Sonnet, not Opus, on top) and one result the headline rates conceal entirely. The within-cell instability, the share of anchor–model cells whose repetitions disagree on the initial verdict class, spans a six-fold range: 0.065 for Sonnet 4.6 at the stable end against 0.404 for Haiku 4.5 at the other, with Opus 4.6 at 0.080, Opus 4.7 at 0.143, Sonnet 4.5 at 0.244, and Opus 4.5 at 0.180 between them. The clearest pair is Haiku 4.5 against Sonnet 4.6: their binary-pass intervals overlap, yet one disagrees with itself on the class of its own answer six times as often as the other. Two caveats attach: the disagreement measure pools benign boundary flips (pass against outdated) with substantive ones (secure against nonexistent), and Sonnet 4.5's value is computed over the 41 cells that survive a behavior-correlated exclusion which removed over a quarter of that model's runs. Even so, the spread is the roster's operationally consequential result: models that pass rates barely separate differ several-fold in whether they give the same answer twice, and a study that had elicited each pin once would not have seen it.

4.9 The Natural Condition

Left free to express the dependency as it would, the model population pinned exactly anyway in half of all cases: 138 of 271 determinate natural runs (50.9%) chose an exact pin, 88 (32.5%) a range, and 45 (16.6%) no constraint at all, with per-model exact-pin rates spanning 0.383 (Haiku 4.5) to 0.625 (Sonnet 4.5). The per-model lag medians sit at zero to two and a half releases against the forced condition's six to nine, with security at the cutoff as good or better (0.931–0.978), and Table 10 gives the vector. On its face this looks like freedom making the models fresher. Stratifying by constraint form, computed from the per-run records, tells a different story. Ranges

Table 10 The natural condition's correctness vector (exploratory, one repetition per cell, never critiqued; 277 runs of which 271 have a determinate constraint form, the 6 indeterminate being the natural errored runs).

		Binary pass		Resolvable		Secure		
Model	n	k/m	Rate	k/m	Rate	k/m	Rate	Lag
Haiku 4.5	48	10/25	0.400	41/47	0.872	40/41	0.976	0.0
Sonnet 4.5	32	7/16	0.438	29/32	0.906	27/29	0.931	1.0
Opus 4.5	50	12/24	0.500	50/50	1.000	48/50	0.960	2.0
Sonnet 4.6	49	11/21	0.524	46/47	0.979	45/46	0.978	0.0
Opus 4.6	49	10/23	0.435	47/48	0.979	44/47	0.936	2.0
Opus 4.7	49	11/23	0.478	46/47	0.979	43/46	0.935	2.5

and unconstrained dependencies land at or near the newest eligible release because the resolver, not the model, picks the concrete version, and all of them were secure at the cutoff on these anchors. The model's own contribution is visible only in the 138 exact pins it chose freely, and those sit at a median lag of 5.0 releases (475 days) with security 0.905, inside the forced condition's band.

The hypothesis that demanding an exact string itself ages the model's choice is therefore not supported once the strata are read, since freely chosen exact pins lag nearly as much as forced ones. What the condition actually shows is simpler and more useful. When the model delegates the version decision to the resolver, the outcome is fresh and, on these anchors, secure; when it answers from parametric memory, under either elicitation, the outcome lags by half a year or more. This sharpens, rather than settles, the pinning trade-off Section 2 frames: exact pinning buys reproducibility at the price of freezing whatever staleness the pinning agent carries, and a parametric-knowledge agent carries half a year or more, while floating constraints buy freshness at the price of the resolver's judgment. The practical lever is not the phrasing of the elicitation but the routing of the decision. The condition is exploratory and single-repetition throughout, and none of this paragraph grounds a confirmatory claim.

5 Threats to Validity

5.1 Construct and Internal Validity.

Construct validity concerns whether the correctness vector measures the decision we intend. The binary pass compresses a multidimensional decision into one bit; we mitigate this by reporting the full vector and never folding the strata into it, but any single-rate comparison inherits the compression. The critique gate is a security label, not a full correctness check, and the gate-disagreement stratum shows the cost: about a fifth of step-1 gate-pass pins fail installation or import downstream. Import is also the component that most depresses the binary pass, yet an import failure need not mean a bad version choice, since it can come from a package-to-module name mismatch, a missing optional dependency, or the measurement environment, so some import outcomes are attribution-neutral. The reliable cutoff is a declared, conservative proxy for an unobservable boundary; the cutoff-sensitivity screen bounds but cannot rule out a

large error in it. The security reading is limited to what the advisory database records and to the dates it assigns, either of which can lag a real disclosure. Finally, the developer comparison contrasts information sets, not agents: both pins are scored against the model's cutoff-era state, which postdates the commit by years, so it speaks to delegation, not to whether the human chose worse at equal information.

Two further threats concern how faithfully the setup represents the behavior it names. The critique is verbal only by design, so we test one operationalization of an informative challenge; because it withholds the external signal the self-correction literature calls a precondition for repair, our repair rates do not bound what an evidence-equipped agent could achieve. The anchors are also public repositories almost certainly in the training data, and each commit contains the answer, so a pin may reflect memorization of that commit rather than recall of the ecosystem. We blunt this by excluding pins whose only release postdates the cutoff and by treating the developer's pin as a reference rather than an oracle, and only about a quarter of pins reproduce it exactly; even so, memorized knowledge of which versions exist flows into the resolvability component and cannot be fully separated from genuine recall.

Internal validity concerns whether the differences we attribute to the manipulation are caused by it. The differenced install construct protects attribution, recording an install outcome only when the anchor's unedited snapshot installs on the matched interpreter, at the cost of null-heavy denominators. Treatment assignment is deterministic and fixed before any outcome exists, which removes selection into the arms, though the arm denominators still float with model accuracy through the gate. The exclusion criterion was double-coded at $\kappa = 0.931$ with no missed contamination in the coded sample, which leaves the clean runs outside that sample unaudited by the second rater. One residual threat we cannot remove is sampling: the interface neither accepts nor reports the decoding temperature or top-p, so the cross-model comparison runs under settings we cannot hold fixed and any apparent difference may partly reflect them, though that contrast is in any case inconclusive on a single discordant pair.

5.2 Conclusion and External Validity.

Conclusion validity concerns whether the statistical inferences are warranted by the data. The gate-fail pools are small by construction, so the repair contrast's power is bounded and its interval wide. Our exact and score-based intervals are conservative, holding at least nominal coverage under independent sampling; because runs cluster within anchors and repositories, we treat that as approximate for aggregated quantities and add the cluster-resampling and leave-one-repository-out checks. The within-cell instability measure pools benign and substantive verdict flips, and one model's value is computed over the cells surviving a behavior-correlated exclusion, so we read it as a spread rather than a precise rate. Finally, only the four hypotheses of Section 3 carry confirmatory weight; the descriptive strata beside them carry no multiplicity adjustment and are stated as hypothesis-generating rather than tested.

External validity concerns how far the findings generalize. Everything here is one ecosystem, one model family, and fifty anchors across twenty-nine repositories in a 2020–2024 window. The anchor set is hand-curated rather than random: the verified

pool was dominated by a single repository, and we traded representativeness for variety across repositories and time, so the fifty anchors illustrate the ecosystem rather than sample it. The natural condition is single-repetition and exploratory, and the critique-arm mix is an artifact of the parity rotation, not a field challenge rate. Generalization beyond Claude-family agents pinning PyPI packages is conjecture until measured.

6 Conclusion

Coding agents increasingly settle the dependency-version decision from parametric memory, without consulting a registry or an advisory database, and that decision carries a security boundary: a stale or vulnerable pin enters the codebase at the moment the agent writes it. We measured this decision where it actually occurs, reconstructing fifty real dependency-introducing commits, drawn from twenty-nine open-source Python repositories, at their parent state and asking six Claude models to add the dependency under tool and network isolation. Each pin was scored on a time-bounded, multidimensional correctness vector and then challenged in a second conversational turn (neutral, directed, or misleading) so that genuine repair of a flawed pin could be separated from the sycophantic abandonment of one that was already correct. The reusable measurement instrument and the anchor dataset are the study's first contribution, built to generalize to other models and ecosystems.

On the correctness of the initial pin (RQ1), parametric knowledge falls measurably short of the best knowledge-bounded answer: every model passes strictly below the pre-cutoff ceiling, with most pins trailing it by six to nine releases (H1). Newer models are far less likely to name a version that does not exist, with resolvability climbing from 0.667 to 0.973 across the roster. On security given existence, however, they sit at the no-knowledge floor of a random pre-cutoff draw, so recall buys *newer* versions, not *safer* ones. Against the human reference the models are fresher and, against cutoff-era knowledge, no less secure on the same tasks, yet their security-and-existence performance does not demonstrably beat a random eligible draw. That is exactly why pin suggestion is delegable only when paired with the registry-and-advisory check the isolation here withheld.

Behavior under critique (RQ2) is the study's central result, and it is a decoupling: a model's willingness to move is almost independent of whether moving is warranted. A false vulnerability alarm overturned a verified-secure pin in 98.5% of cases and never once improved the correctness vector (H3), and even a contentless "are you sure?" overturned 63.7% of good pins, a fraction of them for the worse. An agent's agreement under challenge therefore carries no information, and on a population where most pins are already sound, idle re-asking is net-destructive. Genuine repair, by contrast, could not be confirmed: naming a true defect did not demonstrably improve the pin beyond a bare nudge on the same broken pins, a difference this design could not distinguish from zero (H2) and one bounded by the few broken pins an accurate-model regime produces. The practical discipline that follows is narrow and firm: verify first, challenge only on a finding, and never re-ask a pin idly.

Across generations and tiers (RQ3) the headline pass rates barely separate, and the single confound-controlled generational contrast was inconclusive, as the design

anticipated when tier and cutoff are held equal, on one discordant pair (H4). The operationally consequential difference lies elsewhere: self-consistency, measured across benign and substantive verdict flips alike, spans a six-fold range across the roster, so two models whose pass-rate intervals overlap can still differ several-fold in whether they give the same answer twice. That property matters more in production than a few points of pass rate, and a single elicitation per task would have hidden it. Taken together, the results support a clear posture for practitioners: suggesting a dependency version is a reasonable thing to delegate to a coding agent, but with downstream verification rather than trust, since the value the agent adds is freshness rather than security judgment. Where freshness is what matters, routing the version decision to the resolver rather than eliciting it from parametric memory is the more direct lever.

Two directions follow directly. The sycophancy question is settled on these data, but the repair question is not, and scaling the whole design would be the wrong remedy. A repair-focused successor should instead oversample gate-fail pins directly, mining anchors for popular versions known-vulnerable at the target cutoffs. Such a design could decide whether verbal critique can substitute for the external verification signal that reliable repair is thought to require. The staleness split we observed, that stale-but-secure pins were abandoned more readily than current ones, is consistent with, but does not measure, a credulity that grows with distance from currency; a targeted design could test that reading directly. Beyond that, the instrument is built to travel: the same anchored, time-bounded, critique-based measurement applies to other model families and package ecosystems, where the version decision carries the same security boundary. Whether the need for verification travels with it is a question these data motivate rather than answer, since our findings hold, for now, only for Claude-family agents pinning PyPI packages.

Statements and Declarations

Funding. The work was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq) under grant 309227/2025-9. Moreover, it also was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence), https://www.ines.org.br, CNPq grant 408817/2024-0.

Competing interests. The authors have no competing interests to declare that are relevant to the content of this article.

Ethics approval and consent to participate. Not applicable. This study involves no human participants, human data, or animals. The analysed artefacts are publicly available source-code repositories and the outputs of a commercial language-model service.

Consent for publication. Not applicable.

Data and materials availability. All data and materials supporting the findings of this study are openly available in the replication package at https://doi.org/10.5281/zenodo.21651583. The package contains the fifty-anchor source set in its frozen order, the per-run records for all 1,200 planned runs, and the frozen analysis outputs

behind every table and figure reported here. The upstream repositories from which the anchors are drawn are public on GitHub and are identified by owner, name, and commit in Appendix A.

Code availability. The measurement and critique harness, the analysis scripts, and the figure generation scripts are included in the replication package at https://doi.org/10.5281/zenodo.21651583.

Author contribution. Leonardo Leite designed the study, implemented the measurement and critique harness, collected the data, and wrote the first draft. Caio Barbosa and Elvys Soares contributed to the study design and the analysis plan. Baldoino Fonseca supervised the work. All authors commented on previous versions of the manuscript, and all authors read and approved the final manuscript.

Appendix A The Definitive Anchor Set

Table A1 lists the fifty anchors of the definitive collection, in the frozen file order that drives the critique-arm rotation. The `#` column is the zero-based `anchor_index`.

Appendix B Execution Accounting

Every denominator in the results traces to a single execution accounting. Fifty anchors, six models, three forced repetitions, and one natural run per cell yield 1,200 planned runs, none excluded by the eligibility rule. Of these, 1,165 completed and 35 ended as terminal errored runs after exhausting their retries: 33 of them sessions that entered the interface's plan mode instead of performing the task and never wrote the required output file, and 2 that produced no output. The post-hoc session audit excluded 72 runs as egress-contaminated under the audit criterion. The two populations overlap only partially. Cross-tabulating the run ledger against the exclusions, 12 of the 35 errored runs also carry audit findings and fall within the 72. The other 23 (17 forced and 6 natural, across five models, with clean audits) remain in the analyzed population and surface as missing measurements, so that error handling and contamination exclusion stay distinct, exactly as the design commits. The analyzed population is 1,128 runs, 851 forced and 277 natural, with no malformed record. Table B1 summarizes the stages.

Appendix C Corpus Mining and Selection

Phase A selects candidate repositories from GitHub with at least 200 stars, at least 100 distinct contributors, activity within the six months before the mining date, more than 60% Python code, and creation before January 2025, excluding forks, archived projects, and templates; the creation-date bound reduces the risk that a repository post-dates the evaluated models' knowledge window. Phase B filters their commit and pull-request history to dependency-introduction candidates: a pinned dependency corresponding to a PyPI package, merged into the default branch, mentioned in the commit message or pull-request description (the elicitation prompt is derived from that description), and merged after January 2020.

Seventy repositories yielded 486,493 commits and 424,722 pull requests; the filters admitted 633 candidates, of which 536 survived verification against the package index and the parent-commit state, across thirty-eight repositories. This pool was heavily concentrated: home-assistant/core alone supplies 319 of the 536 (59.5%), largely near-identical integration commits. The fifty-anchor study set (Appendix A) was curated from it to span twenty-nine repositories and the 2020–2024 window.

Appendix D Statistical Methods and Robustness

Proportions and version lag. Most reported quantities are proportions (the binary-pass rate, the pin-change rate under critique, the repair rate, and the false-correction rate), each reported with a Clopper–Pearson exact 95% confidence interval (Clopper 1934; Brown et al. 2001) within a defined stratum (one gate state, condition, or model). Because that guarantee assumes independent trials, quantities aggregated

Table A1 The 50-anchor definitive set (generated from the archived anchor source file; 29 repositories). The `#` column is the frozen `anchor_index` of the arm-rotation parity

#	Repository	Commit	Package	Developer pin	Commit date
0	home-assistant/core	0d5486f7	aiokef	0.2.2	2020-01-03
1	freqtrade/freqtrade	e250c568	questionary	1.5.1	2020-01-29
2	apache/airflow	a9ad0a92	tableauserverclient	0.9	2020-02-22
3	home-assistant/core	df936361	aiohomekit	0.2.10	2020-02-24
4	ansible/ansible	a114da80	antsibull-changelog	0.3.1	2020-06-11
5	huggingface/transformers	8f8f8d99	flax	0.2.2	2020-10-19
6	apache/airflow	cc12db79	rich	9.2.0	2020-11-09
7	ray-project/ray	58d73982	hebo	0.1.0	2021-02-17
8	huggingface/transformers	af8afdc8	librosa	0.8.0	2021-03-18
9	commaai/openpilot	5cd4a9ba	ft4222	1.2.1	2021-04-13
10	microsoft/qlib	f51e04a1	optuna	2.7.0	2021-05-13
11	home-assistant/core	e85b0ec0	soco	0.23.1	2021-07-22
12	ManimCommunity/manim	b8d76cfa	isosurfaces	0.1.0	2021-10-02
13	mindsdb/minds-platform	5c73a372	duckdb	0.3.1	2021-12-17
14	home-assistant/core	432d9a8f	pywizlight	0.4.15	2022-02-05
15	keras-team/keras	be73ac1a	flynt	0.76	2022-08-05
16	huggingface/transformers	9e56aff5	decord	0.6.0	2022-12-08
17	home-assistant/core	38f183a6	aioruuvigateway	0.0.2	2023-01-03
18	oobabooga/textgen	a97afa69	deepspeed	0.8.0	2023-02-01
19	LAION-AI/Open-Assistant	b60eb1e1	prometheus-fastapi-instrumentator	5.9.1	2023-02-10
20	Significant-Gravitas/AutoGPT	ef656a0f	python-dotenv	1.0.0	2023-04-02
21	Significant-Gravitas/AutoGPT	68e4af86	gtts	2.3.1	2023-04-04
22	LAION-AI/Open-Assistant	42e07982	pydantic	1.10.7	2023-04-20
23	OpenBB-finance/OpenBB	9026a41f	posthog	3.0.1	2023-04-28
24	deepspeedai/DeepSpeed	6938c449	neural-compressor	2.1.0	2023-05-10
25	langchain-ai/langchain	6e57306a	clarifai	9.1.0	2023-06-22
26	FoundationAgents/MetaGPT	bec5778d	libcst	1.0.1	2023-08-06
27	FoundationAgents/MetaGPT	96f833cf	redis	4.3.5	2023-09-04
28	home-assistant/core	d1f1bbe3	aiohttp-fast-url-dispatcher	0.1.0	2023-11-09
29	roboflow/supervision	daf4046f	defusedxml	0.7.1	2023-12-10
30	oobabooga/textgen	674be9a0	hqq	0.1.1	2023-12-19
31	FoundationAgents/MetaGPT	c7ee54ac	dashscope	1.14.1	2024-02-07
32	run-llama/llama_index	b2f0a59c	duckduckgo-search	4.5.0	2024-02-27
33	yt-dlp/yt-dlp	52f5be1f	curl-cffi	0.5.10	2024-03-17
34	freqtrade/freqtrade	1036a890	python-rapidjson	1.16	2024-03-29
35	home-assistant/core	0a78e9d4	aiohttp-isal	0.2.0	2024-04-17
36	crewAIInc/crewAI	3d525759	agentops	0.1.6	2024-04-20
37	AUTOMATIC1111/stable-diffusion-webui	49fee7c8	pillow-avif-plugin	1.4.3	2024-04-20
38	vllm-project/vllm	89579a20	vllm-flash-attn	2.5.8.post1	2024-05-08
39	hpcaitech/ColossalAI	18d67d0e	rpyc	6.0.0	2024-05-14
40	Comfy-Org/ComfyUI	8cfd677c	spandrel	0.3.1	2024-05-26
41	mindsdb/minds-platform	10f02eaa	dspy	0.1.4	2024-07-16
42	OpenHands/OpenHands	fa6c1247	grep-ast	0.3.2	2024-07-19
43	ray-project/ray	14a095fc	pyiceberg	0.7.0	2024-08-15
44	run-llama/llama_index	f5846009	gigachat	0.1.28	2024-08-18
45	exo-explore/exo	93224799	tailscale	0.6.1	2024-09-21
46	BerriAI/litellm	f9d0bcc5	websockets	10.4	2024-10-03
47	home-assistant/core	3512cb95	webrtc-models	0.0.0b2	2024-10-25
48	vllm-project/vllm	91642db9	depyf	0.18.0	2024-12-11
49	exo-explore/exo	0a072230	uvloop	0.21.0	2024-12-17

Table B1 Execution accounting of the definitive collection. Errored runs stay in the population as missing measurements; only contamination excludes

Stage	n	Note
Planned runs	1,200	50 anchors × 6 models × (3 forced + 1 natural); 0 ineligible cells
Completed	1,165	97.1% of planned
Terminal errored	35	12 also audit-flagged (within the 72); 23 stay in the population as missing measurements
Excluded: egress-contaminated	72	audit criterion; composition in Appendix D
Analyzed population	**1,128**	851 forced + 277 natural; 0 malformed records

across anchors are refined by the repository-level cluster treatment of Step 6. Version lag is a count (releases and days behind the newest version the model could have known); its distribution is right-skewed, so we summarize it by median and interquartile range.

Cutoff sensitivity. Because the reliable knowledge cutoff is a declared, conservative proxy rather than an observed training boundary, a planned screen tests robustness to small errors in it (Pęzik et al. 2025; Cheng et al. 2024). The headline binary-pass and secure-at-cutoff rates, the lag distribution, and the size and composition of the post-cutoff stratum are recomputed with every model's cutoff shifted by one, two, and three months in each direction, the population frozen at the unshifted cutoff so a record excluded as post-cutoff never re-enters at a shifted one. A quantity is flagged cutoff-sensitive when a one-month shift moves it by more than half the width of its unshifted Clopper–Pearson interval, a deliberately conservative screen rather than a hypothesis test. The screen returned an empty flip list: at a one-month shift neither screened quantity moved beyond half its unshifted interval for any model; the rates are essentially constant across the full range for most models (the largest movement is Haiku 4.5's binary pass drifting from 0.229 to 0.207 at two and three months backward); the post-cutoff stratum never exceeds five rows at any negative shift; and no vulnerability record in the population lacked a publication date. The declared cutoffs are not doing hidden work in the headline numbers.

Audit reliability. Because the contamination audit (Step 5) excludes runs on a human-checkable criterion, its reliability is reported. A double-coding protocol had an independent human recode all 72 machine-flagged runs plus a seeded, model-stratified sample of 50 clean ones; agreement with the automated audit was 118 of 122 (Cohen's $\kappa = 0.931$ (Cohen 1960), PABAK 0.934 (Byrt et al. 1993)), both reported since κ reads low under a skewed base rate. The human found no contamination the audit had missed, and all four disagreements were human over-calls that adjudicated machine-correct, so the exclusion set stands unchanged.

Composition of the excluded runs. The composition of the 72 excluded runs is descriptive, since the frozen criterion excludes them alike: 55 are blocked Bash attempts the interface refused before execution, all from a single model (Sonnet 4.5); 6 are the harness reading its own oversized tool-result files, at paths carrying no

experimental information; about 8 are newer-interface behaviors (tool search, task-management calls, plan mode) in Sonnet 4.6 and Haiku 4.5; and only about 3 achieved genuine out-of-workspace resolution.

References

Alfadel M, Costa DE, Shihab E (2023) Empirical analysis of security vulnerabilities in Python packages. Empirical Software Engineering 28(3):59. https://doi.org/10.1007/s10664-022-10278-4

Anthropic (2026) Models overview. URL https://platform.claude.com/docs/en/about-claude/models/overview

Asare O, Nagappan M, Asokan N (2024) Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code? https://doi.org/10.48550/arXiv.2204.04741, arXiv:2204.04741

Ashik AN, Wang S, Chen TH, et al (2026) When LLMs Lag Behind: Knowledge Conflicts from Evolving APIs in Code Generation. https://doi.org/10.48550/arXiv.2604.09515, arXiv:2604.09515

Basic E, Giaretta A (2026) From Vulnerabilities to Remediation: A Systematic Literature Review of LLMs in Code Security. https://doi.org/10.48550/arXiv.2412.15004, arXiv:2412.15004

Basili V, Heidrich J, Lindvall M, et al (2014) GQM+Strategies: A Comprehensive Methodology for Aligning Business Strategies with Software Measurement. https://doi.org/10.48550/arXiv.1402.0292, arXiv:1402.0292

Bowyer S, Aitchison L, Ivanova DR (2025) Position: Don't Use the CLT in LLM Evals With Fewer Than a Few Hundred Datapoints. https://doi.org/10.48550/arXiv.2503.01747, arXiv:2503.01747

Brown LD, Cai TT, DasGupta A (2001) Interval Estimation for a Binomial Proportion. Statistical Science 16(2):101–133. https://doi.org/10.1214/ss/1009213286

Byrt T, Bishop J, Carlin JB (1993) Bias, prevalence and kappa. Journal of Clinical Epidemiology 46(5):423–429. https://doi.org/10.1016/0895-4356(93)90018-V

Chen M, Tworek J, Jun H, et al (2021) Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/arXiv.2107.03374, arXiv:2107.03374

Cheng J, Marone M, Weller O, et al (2024) Dated Data: Tracing Knowledge Cutoffs in Large Language Models. https://doi.org/10.48550/arXiv.2403.12958, arXiv:2403.12958

Chowdhury K, Banik D, Shamim SI (2026) Breaking the Dependency Chaos: A Constraint-Driven Python Dependency Resolution Strategy with Selective LLM Imputation. https://doi.org/10.48550/arXiv.2605.11772, arXiv:2605.11772

Churilov A (2026) The Range Shrinks, the Threat Remains: Re-evaluating LLM Package Hallucinations on the 2026 Frontier-Model Cohort. https://doi.org/10.48550/arXiv.2605.17062, arXiv:2605.17062

Clopper CJ (1934) The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. Biometrika https://doi.org/10.2307/2331986

Cohen J (1960) A Coefficient of Agreement for Nominal Scales. Educational and Psychological Measurement https://doi.org/10.1177/001316446002000104

Fanous A, Goldberg J, Agarwal AA, et al (2025) SycEval: Evaluating LLM Sycophancy. https://doi.org/10.48550/arXiv.2502.08177, arXiv:2502.08177

Fu Y, Liang P, Tahir A, et al (2025) Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. ACM Transactions on Software Engineering and Methodology 34(8):218:1–218:34. https://doi.org/10.1145/3716848

Guo W, Xu Z, Liu C, et al (2023) An Empirical Study of Malicious Code In PyPI Ecosystem. https://doi.org/10.48550/arXiv.2309.11021, arXiv:2309.11021

Hanif H, Md Nasir MHN, Ab Razak MF, et al (2021) The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. Journal of Network and Computer Applications 179:103009. https://doi.org/10.1016/j.jnca.2021.103009

Haque MN, Lin E, Arkoh L, et al (2025) Secure or Suspect? Investigating Package Hallucinations of Shell Command in Original and Quantized LLMs. https://doi.org/10.48550/arXiv.2512.08213, arXiv:2512.08213

He H, Vasilescu B, Kästner C (2025) Pinning Is Futile: You Need More Than Local Dependency Versioning to Defend against Supply Chain Attacks. Proceedings of the ACM on Software Engineering 2(FSE):266–289. https://doi.org/10.1145/3715728, arXiv:2502.06662 [cs.SE]

Hong J, Byun G, Kim S, et al (2025) Measuring Sycophancy of Language Models in Multi-turn Dialogues. In: Findings of the Association for Computational Linguistics: EMNLP 2025, pp 2239–2259, https://doi.org/10.18653/v1/2025.findings-emnlp.121, arXiv:2505.23840

Hu J, Zhang L, Liu C, et al (2024) Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp 1–13, https://doi.org/10.1145/3597503.3639230, arXiv:2401.00515

Hu Q, Tang Y, Wang Q, et al (2026) SABER: Benchmarking Operational Safety of LLM Coding Agents in Stateful Project Workspaces. https://doi.org/10.48550/arXiv.2606.01317, arXiv:2606.01317

Huang C, Mathai A, Yu F, et al (2026) Outrunning LLM Cutoffs: A Live Kernel Crash Resolution Benchmark for All. https://doi.org/10.48550/arXiv.2602.02690, arXiv:2602.02690

Huang J, Chen X, Mishra S, et al (2024) Large Language Models Cannot Self-Correct Reasoning Yet. https://doi.org/10.48550/arXiv.2310.01798, arXiv:2310.01798

Islah N, Gehring J, Misra D, et al (2024) GitChameleon: Unmasking the Version-Switching Capabilities of Code Generation Models. https://doi.org/10.48550/arXiv.2411.05830, arXiv:2411.05830

Jiang J, Wang F, Shen J, et al (2026) A Survey on Large Language Models for Code Generation. ACM Transactions on Software Engineering and Methodology 35(2):1–72. https://doi.org/10.1145/3747588, arXiv:2406.00515 [cs.CL]

Jimenez CE, Yang J, Wettig A, et al (2024) SWE-bench: Can Language Models Resolve Real-World GitHub Issues? https://doi.org/10.48550/arXiv.2310.06770, arXiv:2310.06770

Kamoi R, Zhang Y, Zhang N, et al (2024) When Can LLMs Actually Correct Their Own Mistakes? A Critical Survey of Self-Correction of LLMs. https://doi.org/10.1162/tacl_a_00713/125177, arXiv:2406.01297

Kordonsky T, LeVi A, Yamin M, et al (2026) Extracting Recurring Vulnerabilities from Black-Box LLM-Generated Software. https://doi.org/10.48550/arXiv.2602.04894, arXiv:2602.04894

Krishna A, Galinkin E, Derczynski L, et al (2025) Importing Phantoms: Measuring LLM Package Hallucination Vulnerabilities. https://doi.org/10.48550/arXiv.2501.19012, arXiv:2501.19012

Kuhar S, Ahmad WU, Wang Z, et al (2024) LibEvolutionEval: A Benchmark and Study for Version-Specific Code Generation. https://doi.org/10.48550/arXiv.2412.04478, arXiv:2412.04478

Kula RG, German DM, Ouni A, et al (2018) Do Developers Update Their Library Dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration. Empirical Software Engineering 23(1):384–417. https://doi.org/10.1007/s10664-017-9521-5, arXiv:1709.04621 [cs.SE]

Ladisa P, Plate H, Martinez M, et al (2022) Taxonomy of Attacks on Open-Source Software Supply Chains. https://doi.org/10.1109/SP46215.2023.00010, arXiv:2204.04008

Lin G, Yu X, Keung J, et al (2025) Lightweight Model Editing for LLMs to Correct Deprecated API Recommendations. https://doi.org/10.48550/arXiv.2511.21022, arXiv:2511.21022

Lira K, Fonseca B, Baía D, et al (2026) Vulnerability Detection with Interprocedural Context in Multiple Languages: Assessing Effectiveness and Cost of Modern LLMs. https://doi.org/10.48550/arXiv.2604.08417, arXiv:2604.08417

Liu J, Wang K, Chen Y, et al (2025a) Large Language Model-Based Agents for Software Engineering: A Survey. https://doi.org/10.48550/arXiv.2409.02977, arXiv:2409.02977

Liu S, Hu X, Xia X, et al (2025b) An Empirical Study of Vulnerable Package Dependencies in LLM Repositories. https://doi.org/10.48550/arXiv.2508.21417, arXiv:2508.21417

Mahmud T, Duan B, Che M, et al (2024) Automated Update of Android Deprecated API Usages with Large Language Models. https://doi.org/10.48550/arXiv.2411.04387, arXiv:2411.04387

Mahon J, Hou C, Yao Z (2025) PyPitfall: Dependency Chaos and Software Supply Chain Vulnerabilities in Python. In: 2025 IEEE International Conference on Big Data (BigData), pp 7825–7834, https://doi.org/10.1109/BigData66926.2025.11401557, arXiv:2507.18075

Mandl P, Mandl P, Häusl M, et al (2026) A Ground-Truth-Based Evaluation of Vulnerability Detection Across Multiple Ecosystems. https://doi.org/10.48550/arXiv.2604.21111, arXiv:2604.21111

McNemar Q (1947) Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages. Psychometrika https://doi.org/10.1007/bf02295996

Miller E (2024) Adding Error Bars to Evals: A Statistical Approach to Language Model Evaluations. https://doi.org/10.48550/arXiv.2411.00640, arXiv:2411.00640

Newcombe RG (1998) Interval estimation for the difference between independent proportions: Comparison of eleven methods. Statistics in Medicine 17(8):873–890. https://doi.org/10.1002/(SICI)1097-0258(19980430)17:8<873::AID-SIM779>3.0.CO;2-I

Ohm M, Plate H, Sykosch A, et al (2020) Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. https://doi.org/10.48550/arXiv.2005.09535, arXiv:2005.09535

Olausson TX, Inala JP, Wang C, et al (2024) Is Self-Repair a Silver Bullet for Code Generation? https://doi.org/10.48550/arXiv.2306.09896, arXiv:2306.09896

Pandey M (2026) LLMs Know They're Wrong and Agree Anyway: The Shared Sycophancy-Lying Circuit. https://doi.org/10.48550/arXiv.2604.19117, arXiv:2604.19117

Pearce H, Ahmad B, Tan B, et al (2022) Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In: 2022 IEEE Symposium on Security and Privacy (SP), pp 754–768, https://doi.org/10.1109/SP46214.2022.9833571

Pearce H, Tan B, Ahmad B, et al (2023) Examining Zero-Shot Vulnerability Repair with Large Language Models. In: 2023 IEEE Symposium on Security and Privacy (SP), pp 2339–2356, https://doi.org/10.1109/SP46215.2023.10179324

Perez E, Ringer S, Lukošiūtė K, et al (2022) Discovering Language Model Behaviors with Model-Written Evaluations. https://doi.org/10.48550/arXiv.2212.09251, arXiv:2212.09251

Perry N, Srivastava M, Kumar D, et al (2023) Do Users Write More Insecure Code with AI Assistants? In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp 2785–2799, https://doi.org/10.1145/3576915.3623157, arXiv:2211.03622

Pęzik P, Kaczyński K, Szymańska M, et al (2025) LLMLagBench: Identifying Temporal Training Boundaries in Large Language Models. https://doi.org/10.48550/ARXIV.2511.12116

Rahman I, Marley J, Enck W, et al (2025) Which Is Better For Reducing Outdated and Vulnerable Dependencies: Pinning or Floating? In: 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 2795–2807, https://doi.org/10.1109/ASE63991.2025.00229, arXiv:2510.08609

Robinson M, Halder S, Ahmed ME, et al (2026) Original Sin of npm: A Study on Vulnerability Propagation in JavaScript Dependency Networks. https://doi.org/10.48550/arXiv.2604.17668, arXiv:2604.17668

Sabetta A, Ponta SE, Cabrera Lozoya R, et al (2024) Known Vulnerabilities of Open Source Projects: Where Are the Fixes? IEEE Security & Privacy 22(2):49–59. https://doi.org/10.1109/MSEC.2023.3343836

Shapira I, Benade G, Procaccia AD (2026) How RLHF Amplifies Sycophancy. https://doi.org/10.48550/arXiv.2602.01002, arXiv:2602.01002

Sharma M, Tong M, Korbak T, et al (2025) Towards Understanding Sycophancy in Language Models. https://doi.org/10.48550/arXiv.2310.13548, arXiv:2310.13548

Singla T, Çakar B, Amusuo PC, et al (2026) Towards a Benchmark for Dependency Decision-Making. https://doi.org/10.48550/ARXIV.2601.00205

Spracklen J, Wijewickrama R, Sakib AHMN, et al (2025) We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs. https://doi.org/10.48550/arXiv.2406.10279, arXiv:2406.10279

Tango T (1999) Improved confidence intervals for the difference between binomial proportions based on paired data by Robert G. Newcombe, Statistics in Medicine, 17, 2635–2650 (1998). Statistics in Medicine 18(24):3511–3513. https://doi.org/10.1002/(SICI)1097-0258(19991230)18:24<3511::AID-SIM303>3.0.CO;2-A

Tie G, Yuan Z, Zhao Z, et al (2025) Can LLMs Correct Themselves? A Benchmark of Self-Correction in LLMs. https://doi.org/10.48550/arXiv.2510.16062, arXiv:2510.16062

Tsui K (2025) Self-Correction Bench: Uncovering and Addressing the Self-Correction Blind Spot in Large Language Models. https://doi.org/10.48550/arXiv.2507.02778, arXiv:2507.02778

Twist L, Zhang JM (2025) A Study of Library Usage in Agent-Authored Pull Requests. https://doi.org/10.48550/ARXIV.2512.11589

Twist L, Zhang JM, Harman M, et al (2026) Library Hallucinations in LLM-Generated Code: A Risk Analysis Grounded in Developer Queries. https://doi.org/10.48550/arXiv.2509.22202, arXiv:2509.22202

Vangala BP, Adibifar A, Gehani A, et al (2025) AI-Generated Code Is Not Reproducible (Yet): An Empirical Study of Dependency Gaps in LLM-Based Coding Agents. https://doi.org/10.48550/ARXIV.2512.22387

Wang C, Chu Z, Cheng Z, et al (2025a) CODESYNC: Synchronizing Large Language Models with Dynamic Code Evolution at Scale. https://doi.org/10.48550/arXiv.2502.16645, arXiv:2502.16645

Wang C, Huang K, Zhang J, et al (2025b) LLMs Meet Library Evolution: Evaluating Deprecated API Usage in LLM-based Code Completion. https://doi.org/10.48550/arXiv.2406.09834, arXiv:2406.09834

Wang C, Wu J, Ling X, et al (2026) Correct Code, Vulnerable Dependencies: A Large Scale Measurement Study of LLM-Specified Library Versions. https://doi.org/10.48550/ARXIV.2605.06279

Wang Y, Wang M, Manzoor MA, et al (2024) Factuality of Large Language Models: A Survey. https://doi.org/10.48550/arXiv.2402.02420, arXiv:2402.02420

Wei J, Huang D, Lu Y, et al (2024) Simple synthetic data reduces sycophancy in large language models. https://doi.org/10.48550/arXiv.2308.03958, arXiv:2308.03958

Xi Z, Chen W, Guo X, et al (2023) The Rise and Potential of Large Language Model Based Agents: A Survey. https://doi.org/10.48550/arXiv.2309.07864, arXiv:2309.07864

Yang-Smith C, Abdellatif A (2025) Tracing Vulnerabilities in Maven: A Study of CVE lifecycles and Dependency Networks. https://doi.org/10.48550/arXiv.2502.04621, arXiv:2502.04621

Zerouali A, Mens T, Decan A, et al (2022) On the Impact of Security Vulnerabilities in the npm and RubyGems Dependency Networks. https://doi.org/10.48550/arXiv.2106.06747, arXiv:2106.06747

Zhao WX, Zhou K, Li J, et al (2026) A Survey of Large Language Models. https://doi.org/10.48550/arXiv.2303.18223, arXiv:2303.18223

Zhu C, Chen N, Gao Y, et al (2024) Is Your LLM Outdated? A Deep Look at Temporal Generalization. https://doi.org/10.48550/ARXIV.2405.08460

Zhu T, Wen Z, Liu S, et al (2026) Assessing the Capability of LLMs for Deprecated API Usage Updating from Natural Language Descriptions. ACM Transactions on Software Engineering and Methodology https://doi.org/10.1145/3808230