THE REWARD HORIZON PROBLEM: LEARNING FROM DELAYED FEEDBACK

Anonymous authors
Paper under double-blind review

ABSTRACT

People increasingly use language models to pursue objectives that are realized over long time horizons. However, post-training rewards are typically computed over short horizons, which creates a mismatch between the reward signal and the ultimate objective. For example, while a short-horizon training signal may reward code that runs, does it reward code that stays stable? We study this *reward horizon problem* in a software engineering setting, where the consequences of a code change may not be observed for several days or months. Using code corrections as a source of delayed feedback, we find substantial differences between the signals available at short and long horizons. Short-horizon feedback omits information about long-horizon objectives and encodes a different distribution of label sources. We then analyze how critic models trained on long-horizon outcomes may distill some of this delayed feedback into short-horizon rewards.

1 INTRODUCTION

Post-training has been a key source of progress in language model development (Ouyang et al., 2022; DeepSeek-AI, 2025). Because rewards can be optimized at scale, models can climb a reward even when the desired behavior is difficult to specify (Christiano et al., 2017). This setting may create alignment problems (Amodei et al., 2016; Gao et al., 2023), where behaviors that maximize reward may not maximize objectives.

One important dimension of alignment is time horizon. While post-training rewards are computed over short time horizons, many economic objectives are realized after a delay. For example, short-horizon rewards may optimize for code that runs, product recommendations that receive clicks, and emails that produce responses. However, such behaviors may not align with long-horizon objectives, like code that is stable, products that people are satisfied with, and emails that eventually produce collaborations.

In this paper, we study the **reward horizon problem**: a mismatch between the horizon over which a reward is computed and the horizon over which the underlying objective is realized. We develop a simple model in which feedback about an action changes with the evaluation horizon. Longer horizons can reveal outcomes that are unobservable in the short run, and can change the distribution of the sources of feedback.

We study this question empirically in software engineering. Software repositories provide a rich history of changes and their documentation, which allows us to observe feedback across several time horizons. Our empirical strategy uses code corrections as delayed feedback about earlier code changes. We measure both whether an initial code change was corrected within a year, and how long it took to make the correction. While such labels do not span all issues with prior code (observability is a key bottleneck to using long-horizon signals), they provide a clearly-documented and standardized data source for us to measure feedback across time horizons.

To enable our analysis, we construct a dataset of 257,788 code changes covering 11 repositories. Our main contributions are as follows:

- **A characterization of the reward horizon problem.** We model how short-run rewards can be misaligned with long-run objectives.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

- **A dataset of long-horizon RL signals in software engineering.** Our dataset links initial code changes to later corrections, and includes code diffs and change timestamps.
- **An analysis of differences in reward signals across horizons.** We find that short-horizon feedback omits information and changes the distribution of feedback sources.
- **Critic models trained across horizons and implications for post-training.** We train models to predict corrections at varying horizons and analyze their properties.

Our work connects to the literature on delayed feedback and surrogate outcomes in online learning, bandits, advertising, and recommender systems (Joulani et al., 2013; Chapelle, 2014; Vernade et al., 2017; Pike-Burke et al., 2018; Mann et al., 2019; Vernade et al., 2020; Lancewicki et al., 2022; Wang et al., 2022; Yang et al., 2024; Maystre et al., 2023; Kleinberg et al., 2024; Zhang et al., 2026; Athey et al., 2026). The reward horizon problem relates to the trajectory horizon and temporal credit assignment problems in reinforcement learning (Sutton, 1984; Arumugam et al., 2021; Park et al., 2025); however, feedback can arrive with a delay even after a very short RL trajectory. Empirically, we build on software engineering research that links code changes to later fixes (Śliwerski et al., 2005; Kim et al., 2006; Kamei et al., 2013; da Costa et al., 2017; Cabral et al., 2019; Rosa et al., 2021); we use outcomes directly labeled by maintainers as a clear source to study how feedback changes over horizons. Our discussion of critic models to distill delayed outcomes into training rewards also relates to literatures on reward modeling and scalable oversight (Christiano et al., 2017; Leike et al., 2018; Christiano et al., 2018).

2 Reward Horizons

Let x denote the context for action a, and let R_h denote a reward based on feedback available at horizon h. For example, in our empirical setting a code change a is made under task context x (such as the PR description); the reward $R_h = -1\{\text{corrected by } h\}$ negates the indicator of whether the change received an observed correction within h days. We take $H > h$ to be a longer horizon that better reflects the underlying objective. The relationship between short- and long-horizon rewards is

$$\underbrace{\mathbb{E}[R_H \mid x, a]}_{\text{long-horizon reward}} = \underbrace{\mathbb{E}[R_h \mid x, a]}_{\text{short-horizon reward}} + \underbrace{\mathbb{E}[R_H - R_h \mid x, a]}_{D_h(x,a)\text{: delayed feedback}} \tag{1}$$

The term $D_h(x, a)$ denotes how the expected reward changes as feedback matures from h to H. This formalism relates to that of Mullainathan & Obermeyer (2017), where models may learn features of both the signal of interest and the measurement process.

Delayed feedback. Misalignment depends on how delayed feedback $D_h(x, a)$ varies across actions. If significant feedback arrives between h and H, the short-horizon reward may omit information about the underlying objective. In a given domain, it may therefore help to measure how much of the longer-horizon reward signal is available at shorter horizons.

Different feedback. Delayed feedback is not necessarily representative of early feedback. In software engineering, for example, it may take longer to identify subtle failures that arise after executing code in multiple settings. It may therefore help to measure how horizons affect the composition of feedback signal.

Critic models. One challenge of using a long-horizon reward at scale is it requires waiting for delayed feedback. However, it may help to use critic models that predict $\mathbb{E}[R_H \mid x, a]$ by using historical observations that have already matured to horizon H.

3 Data

We analyze short and long horizon feedback in the context of software engineering—specifically open source GitHub repositories. GitHub repositories provide detailed records of code changes, reviews, issues, and subsequent fixes, making them well suited for studying feedback over short and long time horizons. We use 11 popular GitHub repositories to construct our dataset: Django, Transformers, NumPy, pandas, pytest, scikit-learn, SciPy, Rust, Kubernetes, Rails, and TypeScript.

Each repository has a consistent workflow for introducing changes to the project code. Code changes are typically introduced through pull requests (PRs), which are reviewed and then merged into the main branch. Subsequent PRs or commits can explicitly reference earlier changes when identifying, fixing, or reverting problems introduced by them. We build a set of candidate edges across this graph by identifying PRs and commits that reference earlier PRs or commits. We define an edge on this graph as being a "correction" when PR content or commit messages mention reverting or fixing something caused by the previous PR. We define the time horizon as the time from when the referenced PR is merged to main to when the referencing PR or commit is merged to main. We include more information about dataset construction in Appendix A.

The correction time horizon measures when the outcome becomes observable in version control. It does not necessarily reflect the time to discovery, which can vary depending on the difficulty of the fix. However, among the 12,391 corrections where the fix belongs to a PR, the median time to merge for the correcting PR (0.7 days) is substantially shorter than the median correction horizon over those same corrections (8.9 days). More importantly, our object of interest is when feedback becomes observable, which under this measurement strategy is at the time of merge.

Our dataset consists of 13,176 corrected PRs and 244,612 uncorrected PRs. The data span 2011–2025. Of the corrected PRs, 10,050 were classified as corrective followups (76.3%) and 3,126 as reverts (23.7%).

4 RESULTS

We investigate the importance, characteristics, and design implications of long horizon feedback in the context of software corrections. First, we show that a substantial amount of economically important corrections surface at longer horizons. Second, we demonstrate the importance of these long-horizon signals by comparing critics trained on short- versus long-horizon correction windows. Third, we analyze how the characteristics associated with corrections differ across short versus long horizons. Finally, we highlight the importance of conditioning the critic on context to help isolate code quality from other variables such as task difficulty.

4.1 DELAYED FEEDBACK: HALF OF CORRECTIONS OCCUR AFTER ONE WEEK

We study how feedback accumulates over time. For each merged PR, we observe whether it was corrected, and if so, the time until its earliest observed correction. We use 365 days as our long-horizon benchmark. We measure the share of this one-year feedback that is observable at shorter horizons. Because we define corrections as code changes that explicitly attribute a causing PR, we do not label fixes that occur without such attribution. Our measurement of correction incidence therefore likely understates the true incidence. This observation strengthens our argument that current training paradigms may not sufficiently measure feedback, including long-horizon feedback, that may provide useful training signal. If the likelihood of explicit attribution decreases over time, for example if memory deteriorates over time (e.g. Johnson et al., 1993; Bornstein & LeCompte, 1995), our analysis may understate the degree of long-horizon feedback. Existing reward pipelines, which may favor feedback that is quick to observe and easy to attribute, may miss substantial downstream feedback.

Figure 1 plots the cumulative rate of observed corrections across horizons. Of 257,788 PRs, 13,176 (5.1%) receive an observed correction within 365 days. Of those corrections, 6,270 occur within the first week and 6,906 occur after seven days. Therefore, after one week, 52.4% of the PRs that we observe were eventually corrected within one year had not yet been corrected. Under this specific short-horizon feedback window, approximately half of the PRs we identify as corrected within a year would appear to have no bugs or issues under this definition.

Among the individual repositories, scikit-learn has the shortest median time to correction at 4.9 days and Django has the longest median time to correction at 62.4 days. Between 42.3% and 73.7% of each repository's one-year corrections occur after the first week. Appendix A reports each repository's PR count. Therefore, across all repositories, short-horizon windows miss a substantial amount of eventually corrected PRs.

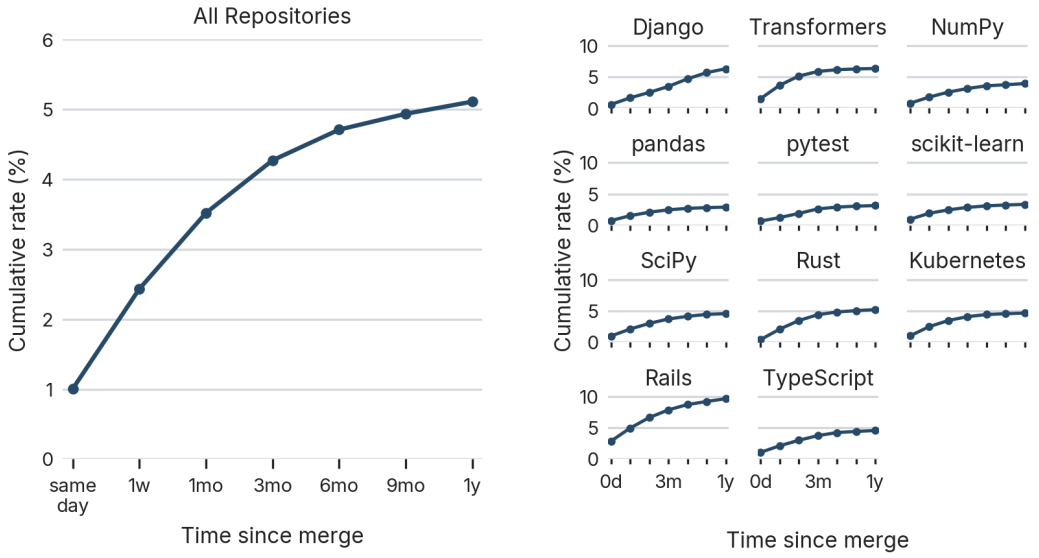


Figure 1: **Feedback across horizons.** This figure reports the cumulative incidence of the earliest observed correction or revert after a merged PR, plotted at quantized horizons (same day, 1 week, 1 month, 3 months, 6 months, 9 months, and 1 year). Left panel: pooled over all repositories. Right: within each repository. The scaling of the vertical axis differs between the pooled panel and the repository-level panels. The denominator is the set of merged PRs that are one year or older.

The observation horizon may change the labels available for training. We next ask whether longer horizon labels allow critic models to be more informative of correction outcomes.

4.2 More Information: Long-Horizon Critics Outperform Short-Horizon Critics

We investigate how extending the feedback window affects critic models' ability to predict correction outcomes. Longer feedback horizons may affect predictability in two ways. First, they may provide more positive examples for training, which may improve the predictive power of the critic. Second, they may contain positive examples associated with a different kind of feedback, which may improve the critic's ability to predict delayed feedback sources. We first evaluate the effects of training using longer feedback windows. We then evaluate whether different horizon feedback windows contain different kinds of training signal relevant to correction risk at different horizons.

For every merged PR, we define nested cumulative correction windows: corrected within 7 days ($\leq 7d$), within 30 days ($\leq 30d$), and within 365 days ($\leq 365d$). We also define disjoint correction windows: corrected within 7 days ($\leq 7d$), later than 7 days but within 30 days ($(7, 30]d$), and later than 30 days but within 365 days ($(30, 365]d$). We train three critics, C7, C30, and C365, to predict corrections within the corresponding cumulative windows. We also train three disjoint critics, D7, D7-30, and D30-365, which are trained with an equal number of positive labels. The disjoint training sets contain a down-sampled set of PRs that are corrected within their respective disjoint window and a fixed negative set of PRs that are not corrected within a year.

Train and evaluation sets use a balanced chronological split with a 365-day gap in between split dates to avoid the potential for lookahead bias (there are 115,784 PRs in the train set, 115,783 in the evaluation set, and 26,221 in the removed gap).[1] For each merged PR, we encode (1) the PR title and body and (2) the code change using Qwen3-Embedding-8B embeddings. For our main specification, our predictive variable is the concatenation of these two embedding vectors. We train each critic using a logistic regression model with an L2 penalty calibrated using five-fold cross validation on

[1]By PR merge date (UTC), the training set spans April 25, 2011–October 14, 2019, and the evaluation set spans October 13, 2020–August 20, 2025.

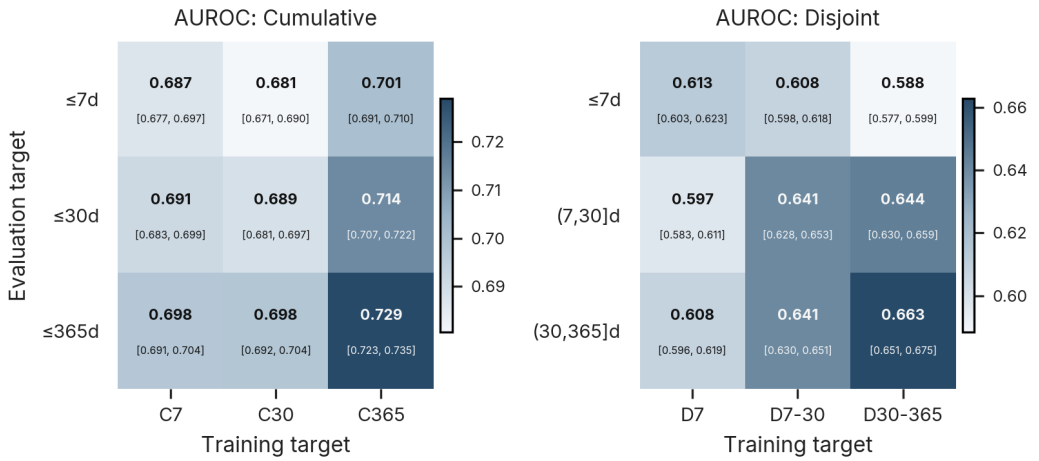


Figure 2: **Critic model performance.** This figure reports the performance of critic models trained on cumulative (left) and disjoint (right) correction outcomes using embeddings of the code change and PR context. Columns indicate the critic's training target and rows indicate the evaluation target; the left panel reports AUROC on the cumulative evaluation targets and the right reports AUROC on disjoint evaluation windows, where each row includes only PRs whose first correction falls in the indicated window and the same 109,129 held-out PRs with no observed correction within a year. Figure C.1 reports AUPRC. C7, C30, and C365 predict correction within 7, 30, or 365 days; each is trained with the PRs not corrected within its own window as negatives. D7, D7-30, and D30-365 each train on 1,025 positives sampled uniformly without replacement from their own window and the same 110,426 training PRs with no observed correction within a year. Each cell shows the point estimate with a bootstrapped 95% confidence interval across the fitted values.

the training set. We evaluate these models on cumulative and disjoint horizon targets from the held-out evaluation set using AUROC. For disjoint evaluation targets, negatives are PRs not corrected within a year. For cumulative targets, negatives are PRs not corrected within the cumulative horizon window.

The left panel of Figure 2 shows the AUROC performance of the C7, C30, and C365 critics on cumulative horizon windows. C365 has the highest AUROC point estimates across all evaluation targets. On the one week target, the performance gap is smallest ($+0.014$ AUROC, 95% CI $[+0.008, +0.019]$) while the largest performance gap is on the one year target ($+0.031$ AUROC, $[+0.028, +0.035]$). No 95% confidence interval crosses zero (Table C.1). These results demonstrate that longer-horizon supervision improves prediction quality over longer-horizon correction windows, without reducing prediction quality at shorter horizons in this setting. These results suggest that long-horizon critics could provide useful reward signals for post-training. In held-out data, code changes ranked differently by the critic have systematically different future outcomes. Because running an on-policy experiment and waiting up to a year for feedback would be impractical, these observational results provide a tractable test of the critics' potential reward signal.

To isolate the second mechanism—different feedback—we evaluate the performance of the D7, D7-30, and D30-365 disjoint critics on disjoint correction horizons (PRs that are corrected within 0–7 days, 7–30 days, and 30–365 days) with a matched positive class sample count across each critic in the right panel of Figure 2. The results show that on the 30–365 day evaluation set, the D30-365 critic has higher AUROC than the D7 critic. Conversely, on the 0–7 day evaluation set, the D7 critic has higher AUROC than the D30-365 critic. These results are consistent with different horizons having distinct correction-relevant signals.

4.3 Different Feedback: Characteristics Vary Across Correction Horizons

What are some ways that feedback can differ across horizons? In a descriptive, exploratory analysis of a subset of corrected PRs, we find that mean time to correction differs systematically across

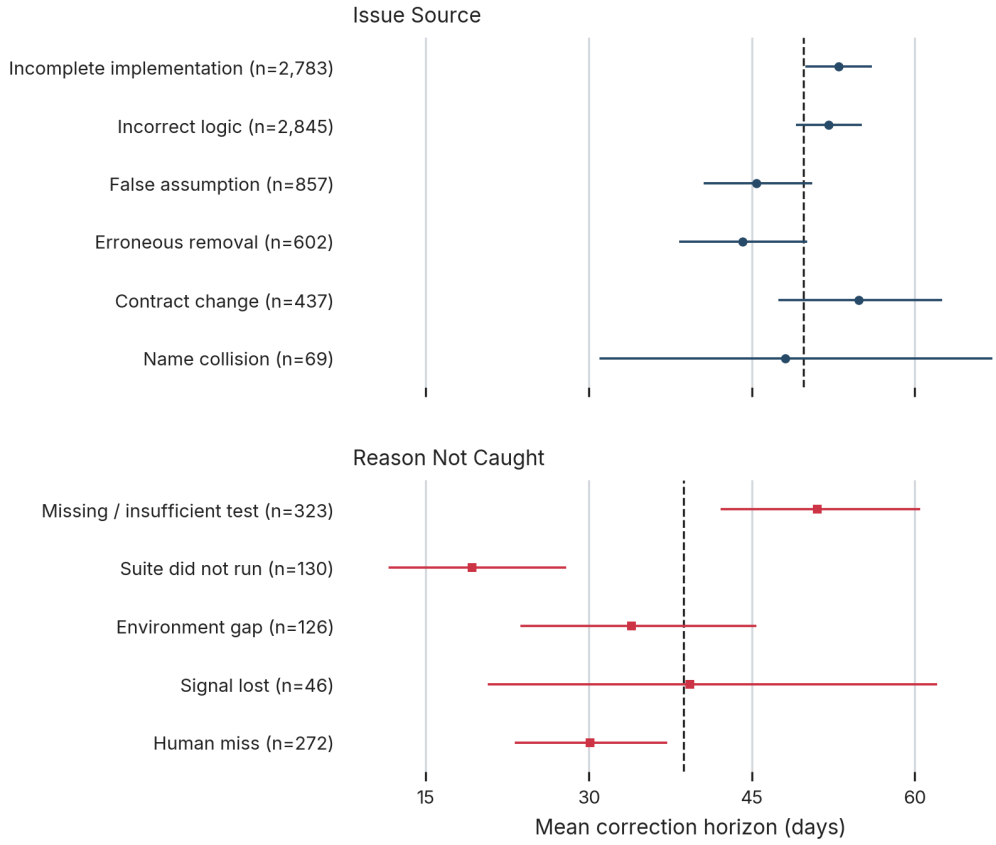


Figure 3: **Feedback characteristics across horizons.** This figure reports the mean correction horizon among PRs with an observed correction within 365 days, labeled by Issue Source (top) and Reason Not Caught (bottom). The top panel covers 7,639 corrections with a panel mean of 49.7 days, the bottom panel 1,385 corrections with a panel mean of 38.7 days. The dashed line in each panel marks that panel's mean. Statistics for Other and Unknown labels are reported in the Appendix. Points are label-level means of the days from causing PR merge to correcting merge, with bootstrapped 95% confidence intervals. Labels can co-occur, and Table D.1 reports full statistics for all labels.

PR characteristics. If PRs with certain characteristics are corrected later on average, short horizon feedback may disproportionately undercount correction sources for PRs with those characteristics.

For each correction, we generate two fields from the correcting change's title, body, commit message and the code diff: **Issue Source** (whether the breaking change happened because of incomplete implementation, incorrect logic, false assumption, erroneous removal, contract change, or name collision) and **Reason Not Caught** (whether the change was not initially caught because of missing or insufficient tests, suite did not run, environment gap, signal lost, or human miss). We generate multi-label classifications, with additional details presented in Appendix E and examples of each label in Appendix F.

This empirical strategy aims to provide one possible explanation for the benefits of long-horizon supervision: feedback characteristics differ across horizons. However, as we only observe these characteristics when documented, we do not claim that these labels are representative of all corrections. The results instead provide descriptive evidence of some potential sources of variation in feedback composition across horizons.

Our outcome variable is the number of days from causing PR merge to correcting merge among 7,639 and 1,385 corrections whose correcting change explicitly mentioned the issue source or reason not caught, respectively. For each taxonomy group, we report the mean correction horizon and

the present-versus-absent difference in means, each with a bootstrapped 95% confidence interval. Figure 3 plots mean days to correction for the Issue Source labels and Reason Not Caught labels.

Figure 3 shows that, among issue sources, incomplete implementation, incorrect logic, and contract change tend to have later horizons ($+5.2$ days, 95% CI $[+1.3, +8.9]$; $+3.7$ days, 95% CI $[-0.0, +7.6]$; $+5.4$ days, 95% CI $[-2.2, +13.3]$) while false assumption and erroneous removal tend to have shorter horizons (-4.9 days, 95% CI $[-10.1, +0.5]$; -6.1 days, 95% CI $[-12.2, +0.1]$). Among reasons not caught, missing or insufficient tests tend to have later horizons ($+16.0$ days, 95% CI $[+6.4, +26.3]$) and human misses and unrun test suites have shorter horizons on average (-10.7 days, 95% CI $[-19.0, -2.4]$; -21.5 days, 95% CI $[-30.2, -11.8]$). Figure D.2 reports these statistics after adjusting for repository fixed effects, and finds the same ordering. Figure D.3 reports label co-occurrence between the two taxonomies.

Shorter horizon labels appear to be more associated with issues that could easily be avoided if the PR went through proper review and testing: such as a human PR review or running a test suite. One implication is short horizon feedback might disproportionately capture these cheaper short horizon failure modes over the more involved long horizon failure modes such as a missing test or having incorrect logic.

It is possible that because labels exist only where the correcting change documented the mechanism, the likelihood of documentation could be confounded with the complexity of the issue. For example, if the issue is something simple, it may not be worth documenting it. Another possibility is that if a correction comes later it could reflect the difficulty of the fix rather than failure mode, since horizons are measured based on merges. However, as previously mentioned, the median time to merge is multiple times shorter than the median time to correction.

4.4 Training Strategies: Conditioning on Context Improves Predictability

How can researchers develop training strategies to address the reward horizon problem? Observational data provides a natural source of long-horizon feedback. The prior critic results study how feedback horizons affect critic performance. A complementary design choice is what information to provide the critic. One challenge with using observational, offline data is that actions are taken under varying contexts. In our framework, the target object is $\mathbb{E}[R_H \mid x, a]$, but a critic that observes a but insufficiently observes x may conflate action quality with baseline risk.

In software engineering, for example, we might observe higher correction for some types of code changes because the code was faultier than expected, or because some contexts (like complex new features) typically bring about more failures. A critic model that does not observe enough task context may conflate these two sources of risk, and penalize code written for difficult tasks instead of penalizing poor implementations. Optimizing on such a critic could lead to unintended behaviors, for example by producing code that does not satisfy a user's more complex objective. One way to address this issue is to condition on context as best we can.

We examine this possibility by comparing the predictions of two critic models trained on varying degrees of context. The code-change-only critic uses only an embedding of the code change. The PR-context critic additionally uses the embedding of the PR title and body, which could provide additional information about the underlying task.[2] The critic models predict correction risk, so the negative of the critic score can provide a reward signal. For each PR, we measure three observable proxies for task difficulty: the natural logarithm of lines changed, a complexity label from an LLM, and the residual from regressing natural logarithm of lines on the complexity label. The exact LLM prompt is shown in E.2. On the held-out evaluation set, we score PRs using the critics and analyze the association between critic scores and the proxy metrics. We report Pearson correlations of critic scores and proxy metrics.[3]

[2] As the PR-context regression is trained on both embeddings, the coefficients on the code change embedding may differ from the coefficients from the code-change-only critic regression. Therefore, the PR-context critic may rank implementations of the same context differently from the code-change-only critic.

[3] The goal of these regressions is to diagnose how much each critic's predictions rely on observable proxies for task complexity. They are not intended to demonstrate a causal effect of lines changed or complexity on correction risk.

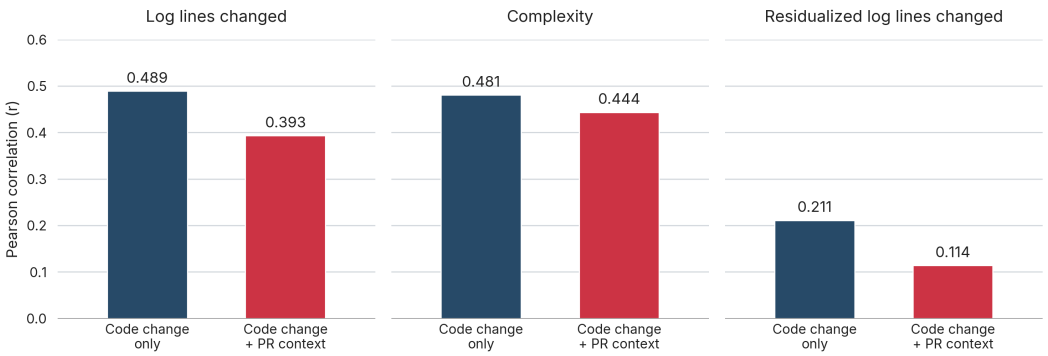


Figure 4: **Behavior of critics that condition on context.** This figure reports Pearson correlations between critic risk scores and complexity proxies on the held-out evaluation set: log lines changed, LLM-labeled change complexity, and log lines changed residualized on complexity. Each panel compares the code-change-only critic with the PR-context critic, which additionally uses the PR title and body. Residualized log lines changed is log lines changed minus its complexity-conditional mean, fit on the held-out PRs with categorical complexity.

Figure 4 shows that both critics assign higher scores to more complex PRs, but that this relationship is weaker for the PR-context critic. The Pearson correlation between critic scores and our complexity proxies is reduced by conditioning on the task context. This reduced correlation is consistent with, though does not causally establish, the PR-context critic evaluating the code change relative to the task, and consistent with less confounding by task complexity (e.g., treating a complex task as an indicator of a poor implementation). An alternative explanation is that the relationship is weaker because the PR-context critic is a weaker predictor. This is not the case, as on the evaluation set the PR-context model outperforms the code-change critic by $+0.046$ AUROC (0.729 versus 0.683, 95% CI $[+0.042, +0.050]$).

This result is largely an illustration of the potential confound between length and complexity in the context of code quality critics. We do not claim to identify a mechanism of reduced confounding. The result is not meant to argue that an ideal critic should have no correlation with lines changed or complexity. Additionally, most critic training configurations would certainly include the PR context. The point we are making is not to state the obvious, but to emphasize the importance of conditioning on as much of the context as possible. For example, not only the PR context, but perhaps also the state of the repository at the time.

5 Discussion and Limitations

We have characterized the reward horizon problem: an alignment problem created by the mismatch between short-horizon rewards and long-horizon objectives. To study this problem empirically, we developed a dataset of correction feedback in open source software repositories. We found that short-horizon feedback can omit information relative to long-horizon feedback, and change the composition of feedback sources.

Software engineering is a favorable setting to study feedback horizons, given the density of information we can find in open-source repositories. Even in this setting, however, our empirical task is difficult. Our correction labels require explicit attribution, so they likely understate the true incidence of issues. Moreover, our measurement strategy considers the time to both discover and fix the issue, not just to discover it. Our goal is to present evidence on the reward horizon problem, not to completely solve it, and these measurement challenges suggest that the problem may be even stronger in the field. These measurement challenges are also likely to be stronger in domains with weaker systems of record and attribution. Addressing these issues in the field will require further research and additional record-keeping.

One way to incorporate long-horizon feedback at the scale required for post-training is to approximate it through critic models. Such models, when trained on observational data, may need to

condition on rich context in order to reduce confounding. In our setting, critics trained at longer cumulative horizons can outperform critics trained at shorter horizons, which suggests that some delayed feedback can be distilled into shorter-term feedback. Our data and findings create a natural recipe for future research that applies these strategies to post-training.

AI USE STATEMENT

In this work, we used generative AI tools for generating code for analysis and data reformatting, drafting and reviewing portions of the manuscript, conducting literature searches, and eliciting feedback on research ideas including hypotheses and experiments. We have not used generative AI tools for synthetic data generation (the underlying repository and code data are from public repositories) or proving mathematical theorems. Additionally, we used generative AI tools for labeling examples in the dataset. We have reviewed all AI-assisted work. For example, we reviewed and edited generated writing, conducted manual literature surveys, and tested generated code. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

Our paper studies how short-horizon rewards may miss delayed feedback about an action; this reward horizon problem may misalign models with respect to long-run objectives. Our findings motivate incorporating delayed feedback, which may help develop training signals that better align with these objectives.

REPRODUCIBILITY STATEMENT

Appendix A describes dataset construction, and Section 4 describes the experimental design. Our dataset is available during the review period at this anonymized link.

REFERENCES

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

Dilip Arumugam, Peter Henderson, and Pierre-Luc Bacon. An information-theoretic perspective on credit assignment in reinforcement learning. *arXiv preprint arXiv:2103.06224*, 2021.

Susan Athey, Raj Chetty, Guido W. Imbens, and Hyunseung Kang. The surrogate index: Combining short-term proxies to estimate long-term treatment effects more rapidly and precisely. *The Review of Economic Studies*, 93(4):2284–2312, 2026. doi: 10.1093/restud/rdaf087.

Brian H. Bornstein and Denny C. LeCompte. A comparison of item and source forgetting. *Psychonomic Bulletin & Review*, 2(2):254–259, 1995. doi: 10.3758/BF03210966.

George G. Cabral, Leandro L. Minku, Emad Shihab, and Suhaib Mujahid. Class imbalance evolution and verification latency in just-in-time software defect prediction. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 666–676. IEEE Computer Society, 2019. doi: 10.1109/ICSE.2019.00076. URL https://doi.org/10.1109/ICSE.2019.00076.

Olivier Chapelle. Modeling delayed feedback in display advertising. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1097–1105. ACM, 2014. doi: 10.1145/2623330.2623634.

Paul Christiano, Buck Shlegeris, and Dario Amodei. Supervising strong learners by amplifying weak experts. *arXiv preprint arXiv:1810.08575*, 2018.

Paul F. Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017. doi: 10.1109/TSE.2016.2616306.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10835–10866. PMLR, 2023.

Marcia K. Johnson, Shahin Hashtroudi, and D. Stephen Lindsay. Source monitoring. *Psychological Bulletin*, 114(1):3–28, 1993. doi: 10.1037/0033-2909.114.1.3.

Pooria Joulani, Andras Gyorgy, and Csaba Szepesvari. Online learning under delayed feedback. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28(3) of *Proceedings of Machine Learning Research*, pp. 1453–1461. PMLR, 2013.

Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi: 10.1109/TSE.2012.70.

Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pp. 81–90, 2006. doi: 10.1109/ASE.2006.23.

Jon Kleinberg, Sendhil Mullainathan, and Manish Raghavan. The challenge of understanding what users want: Inconsistent preferences and engagement optimization. *Management Science*, 70(9): 6336–6355, 2024. doi: 10.1287/mnsc.2022.03683.

Tal Lancewicki, Aviv Rosenberg, and Yishay Mansour. Learning adversarial markov decision processes with delayed feedback. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36 (7):7281–7289, 2022. doi: 10.1609/aaai.v36i7.20690.

Jan Leike, David Krueger, Tom Everitt, Miljan Martic, Vishal Maini, and Shane Legg. Scalable agent alignment via reward modeling: A research direction. *arXiv preprint arXiv:1811.07871*, 2018.

Timothy Arthur Mann, Sven Gowal, Andras Gyorgy, Huiyi Hu, Ray Jiang, Balaji Lakshminarayanan, and Prav Srinivasan. Learning from delayed outcomes via proxies with applications to recommender systems. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 4324–4332. PMLR, 2019.

Lucas Maystre, Daniel Russo, and Yu Zhao. Optimizing audio recommendations for the long-term: A reinforcement learning perspective. *arXiv preprint arXiv:2302.03561*, 2023.

Sendhil Mullainathan and Ziad Obermeyer. Does machine learning automate moral hazard and error? *American Economic Review*, 107(5):476–480, 2017. doi: 10.1257/aer.p20171084.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Seohong Park, Kevin Frans, Deepinder Mann, Benjamin Eysenbach, Aviral Kumar, and Sergey Levine. Horizon reduction makes RL scalable. In *Advances in Neural Information Processing Systems*, volume 38, pp. 9258–9297, 2025. doi: 10.52202/085713-0284.

Ciara Pike-Burke, Shipra Agrawal, Csaba Szepesvari, and Steffen Grunewalder. Bandits with delayed, aggregated anonymous feedback. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4105–4113. PMLR, 2018.

Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. Evaluating SZZ implementations through a developer-informed oracle. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*, pp. 436–447. IEEE, 2021. doi: 10.1109/ICSE43902.2021.00049.

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the Second International Workshop on Mining Software Repositories*, pp. 24–28, 2005. doi: 10.1145/1083142.1083147.

Richard S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, 1984.

Claire Vernade, Olivier Cappé, and Vianney Perchet. Stochastic bandit models for delayed conversions. In *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence*, 2017.

Claire Vernade, Andras Gyorgy, and Timothy Mann. Non-stationary delayed bandits with intermediate observations. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 9722–9732. PMLR, 2020.

Yuyan Wang, Mohit Sharma, Can Xu, Sriraj Badam, Qian Sun, Lee Richardson, Lisa Chung, Ed H. Chi, and Minmin Chen. Surrogate for long-term user experience in recommender systems. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4100–4109. ACM, 2022. doi: 10.1145/3534678.3539073.

Jeremy Yang, Dean Eckles, Paramveer Dhillon, and Sinan Aral. Targeting for long-term outcomes. *Management Science*, 70(6):3841–3855, 2024. doi: 10.1287/mnsc.2023.4881.

Kelly W. Zhang, Thomas Baldwin-McDonald, Kamil Ciosek, Lucas Maystre, and Daniel Russo. Impatient bandits: Optimizing for the long-term without delay. *Journal of Machine Learning Research*, 27(156):1–58, 2026. URL `https://jmlr.org/papers/v27/25-0030.html`.

A Data

We first construct all possible text reference connection edges between GitHub PRs and commits by ID or SHA. All nodes are rolled up to their parent PR if present for any PR related context such as title or body text, but not for the raw code diff. For example, if a reference is made from a commit message and the commit is a member of a PR, then its node will be considered its parent PR. All PR information comes from the parent PR, but the code diff comes from the specific commit (not the PR's merge commit). In the case where the reference is made from a PR text body, then the PR information for that node obviously comes from the PR, but the code diff comes from that PR's merge commit. Each PR has a single unique merge commit.

To filter our set of connections to corrections, we first remove any negative horizon edges. We remove negative horizons both at the commit level and at the PR merge level. Then we use an LLM to classify the type of connection and filter to corrective followups or reversions using the metadata from the PRs and commits. Exact prompts are listed in E.1, and we compare these labels with two human labelers in Appendix B. We then restrict to connections where the causing nodes are PRs or belong to parent PRs. However, for correcting nodes we also include commits that do not have parent PRs. If a causing PR has been corrected by multiple later nodes, its earliest direct correcting node is kept. A correcting node can appear in multiple correction edges in cases where it corrects multiple PRs. In our dataset, we only include corrections made within 365 days.

For the main-text comparison of correcting-PR merge time and correction horizon, we select each causing PR's earliest correction within the observation window that is a PR or belongs to a parent PR. This includes cases where an earlier correction was a commit without a parent PR. Both quantities are measured using the selected PR-backed correction, consistent with the PR-backed taxonomy analysis in the appendix. Other analyses retain the earliest correction regardless of whether it has a parent PR.

Figure 1 uses 257,788 merged PRs that are at least one year old: Django (8,951), Transformers (15,954), NumPy (12,629), pandas (26,647), pytest (5,415), scikit-learn (13,203), SciPy (10,126),

Rust (64,011), Kubernetes (61,364), Rails (24,873), and TypeScript (14,615). Django, NumPy, pandas, scikit-learn, and SciPy are primarily licensed under BSD-3-Clause; pytest and Rails under MIT; Transformers, Kubernetes, and TypeScript under Apache-2.0; and Rust under Apache-2.0 or MIT.

B Human Validation of LLM Labels

We validate the three binary LLM labels and the Issue Source and Reason Not Caught taxonomies (Appendix E.2). The binary labels are whether a candidate edge is a correction (Appendix E.1), whether the fixing PR or commit mentions the issue source (Appendix E.2), and whether it mentions why the issue was not caught (Appendix E.2). For the correction label, the connection types returned by the prompt are collapsed to correction (corrective followup or revert) versus not correction. Two labelers with computer science degrees independently labeled the same examples using the same context as the LLM. The correction sample contains 200 candidate edges, half of which were drawn from GPT-5.6 Terra's correction labeled set and the other half from GPT-5.6 Terra's non-correction labeled set.

Figure B.1 and Table B.1 report percent agreement and Cohen's κ for each pair of labelers. Terra agrees with each human labeler about as often as the two labelers agree with each other.

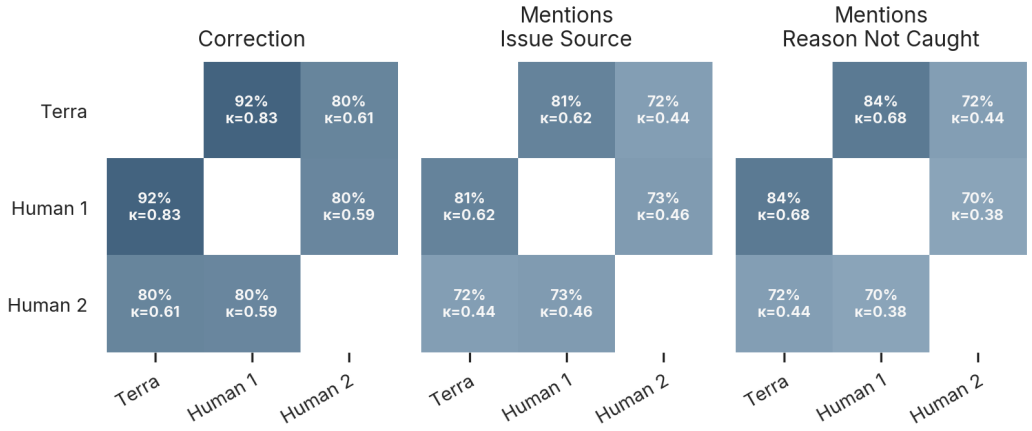


Figure B.1: This figure reports pairwise agreement on the three binary validation labels. Each cell gives the percentage of examples on which the row and column labelers agree, and Cohen's κ. Terra is GPT-5.6 Terra with low reasoning effort. The correction sample has 200 examples and each mention sample 100, all balanced on Terra's label.

Table B.1: This table reports percent agreement and Cohen's κ for each pair of labelers on the three binary validation labels, over 200 correction examples and 100 examples for each mention label.

	Correction		Mentions Issue Source		Mentions Reason Not Caught	
Pair	Agree (%)	κ	Agree (%)	κ	Agree (%)	κ
Human 1 / Human 2	80.0	0.59	73.0	0.46	70.0	0.38
Terra / Human 1	91.5	0.83	81.0	0.62	84.0	0.68
Terra / Human 2	80.5	0.61	72.0	0.44	72.0	0.44

The two taxonomies are validated on corrections that Terra labeled as mentioning an issue source or a reason not caught. Two sets of 100 corrections were drawn, each in two steps. First, we drew corrections for each category until every category appeared at least 5 times. This resulted in 36 corrections for Issue Source and 34 for Reason Not Caught, since a correction drawn for one category can also carry others. Second, we drew the remaining corrections uniformly at random

from the rest of the pool. Both taxonomies are multi-label, so we count a pair as agreeing only when their label sets match exactly, and compute Cohen's κ treating each distinct label set as one category. Figure B.2 reports both for every pair. Terra's label sets match Human 1's on 60.0% (Issue Source) and 58.0% (Reason Not Caught) of examples and Human 2's on 37.0% and 33.0%, against 39.0% and 38.0% between the two labelers. The exact set match counts any difference as a disagreement, including on the Other and Unknown labels that are omitted in Figure 3.

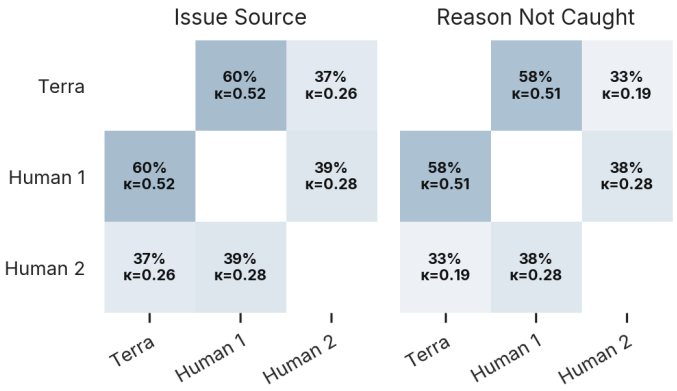


Figure B.2: This figure reports pairwise exact set match on the two multi-label taxonomies. Each cell gives the percentage of examples on which the row and column labelers assign the same label set, and Cohen's κ with each distinct label set treated as one category. Each sample has 100 corrections, drawn with a minimum quota for each of Terra's categories.

Table B.2: This table reports Cohen's κ for the presence of each taxonomy label displayed in Figure 3, for Terra against each human labeler and between the two labelers. Each label is scored as present or absent on each of the 100 examples in its taxonomy's sample.

Label	Terra / Human 1	Terra / Human 2	Human 1 / Human 2
Issue Source			
Incomplete implementation	0.68	0.50	0.53
Incorrect logic	0.57	0.19	0.19
False assumption	0.48	0.55	0.43
Erroneous removal	0.68	0.35	0.39
Contract change	0.59	0.23	0.21
Name collision	0.58	0.56	0.26
Reason Not Caught			
Missing or insufficient test	0.61	0.31	0.29
Suite did not run	0.81	0.40	0.41
Environment gap	0.49	0.37	0.50
Signal lost	0.71	0.10	0.12
Human miss	0.68	0.58	0.68

C CUMULATIVE AND DISJOINT HORIZON CRITICS

We report additional results for the critics in Figure 2, as well as extensions that modify their training with additional positive class construction strategies. In the disjoint-window training ablation with matched positive counts, each of the three critics trains on the same number of positives, 1,025, which is the size of the smallest training window, $(7, 30]$d, and on the same 110,426 PRs with no observed correction within a year. All 95% confidence intervals in the section are formed by 10,000 bootstrap draws of the evaluation dataset rows. The fitted critic models are held fixed.

Table C.1 reports the C365 versus C7 and D30-365 versus D7 AUROC comparisons behind Figure 2, on the cumulative evaluation targets and the disjoint evaluation windows. Table C.2 reports the same comparisons for the AUPRC results in Figure C.1. Table C.3 reports the C365 versus C7 comparison on the disjoint evaluation windows in Figure C.2. Table C.4 and Table C.5 report the AUROC of C7, C30, and C365 broken out by repository, on the cumulative evaluation targets (left panel of Figure 2) and the disjoint evaluation windows (Figure C.2), respectively.

Table C.1: The C365 versus C7 and D30-365 versus D7 AUROC comparisons underlying Figure 2, on the cumulative evaluation targets and the disjoint evaluation windows. Δ is C365 or D30-365 performance minus C7 or D7 performance.

Evaluation	Target	C7	C365	Δ	95% CI
Cumulative	$\leq$ 7d	0.687	0.701	$+0.014$	$[+0.008, +0.019]$
Cumulative	$\leq$ 30d	0.691	0.714	$+0.023$	$[+0.019, +0.027]$
Cumulative	$\leq$ 365d	0.698	0.729	$+0.031$	$[+0.028, +0.035]$
Evaluation	Target	D7	D30-365	Δ	95% CI
Disjoint	$\leq$ 7d	0.613	0.588	-0.025	$[-0.039, -0.010]$
Disjoint	$(7, 30]$d	0.597	0.644	$+0.047$	$[+0.027, +0.067]$
Disjoint	$(30, 365]$d	0.608	0.663	$+0.055$	$[+0.039, +0.072]$

Table C.2: The C365 versus C7 and D30-365 versus D7 AUPRC comparisons underlying Figure C.1, on the cumulative evaluation targets and the disjoint evaluation windows. Δ is C365 or D30-365 performance minus C7 or D7 performance.

Evaluation	Target	C7	C365	Δ	95% CI
Cumulative	$\leq$ 7d	0.059	0.063	$+0.003$	$[+0.001, +0.006]$
Cumulative	$\leq$ 30d	0.090	0.099	$+0.009$	$[+0.007, +0.012]$
Cumulative	$\leq$ 365d	0.135	0.156	$+0.021$	$[+0.017, +0.024]$
Evaluation	Target	D7	D30-365	Δ	95% CI
Disjoint	$\leq$ 7d	0.046	0.044	-0.002	$[-0.006, +0.001]$
Disjoint	$(7, 30]$d	0.024	0.031	$+0.007$	$[+0.005, +0.011]$
Disjoint	$(30, 365]$d	0.034	0.044	$+0.010$	$[+0.007, +0.014]$

Tables C.4 and C.5 report the full cross-target AUROC results broken out by repository, on the cumulative evaluation targets and the disjoint evaluation windows respectively, using the same critics as Figure 2. Each critic is fit once on the full training split; evaluation rows are then filtered per repository to compute point estimates.

Table C.3: The C365 versus C7 comparison underlying Figure C.2. The critics retain their cumulative training targets and are evaluated on disjoint correction windows. Δ is C365 performance minus C7 performance.

Metric	Eval target	C7	C365	Δ	95% CI
AUROC	$\leq$ 7d	0.694	0.709	+0.015	[+0.010, +0.020]
AUROC	(7, 30]d	0.698	0.739	+0.041	[+0.033, +0.048]
AUROC	(30, 365]d	0.703	0.749	+0.046	[+0.040, +0.052]
AUPRC	$\leq$ 7d	0.066	0.071	+0.005	[+0.002, +0.008]
AUPRC	(7, 30]d	0.036	0.044	+0.008	[+0.005, +0.010]
AUPRC	(30, 365]d	0.052	0.066	+0.014	[+0.011, +0.017]

Table C.4: Per-repository AUROC for every (training critic, cumulative evaluation target) combination. Columns are grouped by evaluation target ($\leq$ 7d, $\leq$ 30d, $\leq$ 365d); within each group, C7/C30/C365 denote the training horizon. All critics are fit on the full training split; evaluation rows are filtered per repository.

	$\leq$ 7d			$\leq$ 30d			$\leq$ 365d		
Repository	C7	C30	C365	C7	C30	C365	C7	C30	C365
Django	0.658	0.660	0.686	0.663	0.676	0.706	0.715	0.725	0.746
Transformers	0.685	0.692	0.710	0.699	0.708	0.728	0.701	0.711	0.731
NumPy	0.679	0.681	0.695	0.692	0.691	0.722	0.682	0.684	0.724
pandas	0.660	0.650	0.649	0.652	0.648	0.645	0.656	0.657	0.656
pytest	0.782	0.799	0.747	0.645	0.650	0.696	0.649	0.657	0.732
scikit-learn	0.690	0.685	0.707	0.690	0.687	0.709	0.703	0.703	0.730
SciPy	0.639	0.639	0.663	0.656	0.657	0.675	0.666	0.666	0.688
Rust	0.671	0.671	0.694	0.672	0.674	0.704	0.678	0.681	0.714
Kubernetes	0.709	0.709	0.724	0.702	0.700	0.719	0.697	0.698	0.725
Rails	0.688	0.691	0.698	0.701	0.706	0.717	0.706	0.710	0.727
TypeScript	0.677	0.668	0.693	0.680	0.671	0.698	0.685	0.679	0.727

Table C.5: Per-repository AUROC on the disjoint evaluation windows, using the same critics and protocol as Table C.4. Each window keeps only PRs whose first correction falls in it, against the repository's held-out PRs with no observed correction within a year.

	$\leq$ 7d			(7, 30]d			(30, 365]d		
Repository	C7	C30	C365	C7	C30	C365	C7	C30	C365
Django	0.672	0.675	0.701	0.680	0.707	0.744	0.737	0.746	0.762
Transformers	0.692	0.699	0.718	0.726	0.740	0.764	0.698	0.709	0.732
NumPy	0.682	0.684	0.700	0.721	0.712	0.778	0.653	0.661	0.722
pandas	0.662	0.652	0.651	0.633	0.643	0.636	0.663	0.674	0.679
pytest	0.784	0.801	0.751	0.430	0.417	0.620	0.652	0.662	0.767
scikit-learn	0.694	0.688	0.711	0.688	0.691	0.711	0.734	0.745	0.783
SciPy	0.645	0.644	0.669	0.696	0.701	0.703	0.677	0.676	0.704
Rust	0.678	0.678	0.702	0.673	0.677	0.718	0.680	0.686	0.724
Kubernetes	0.714	0.714	0.730	0.688	0.682	0.708	0.682	0.688	0.729
Rails	0.704	0.707	0.715	0.722	0.731	0.752	0.698	0.700	0.727
TypeScript	0.684	0.675	0.702	0.686	0.677	0.710	0.685	0.684	0.758

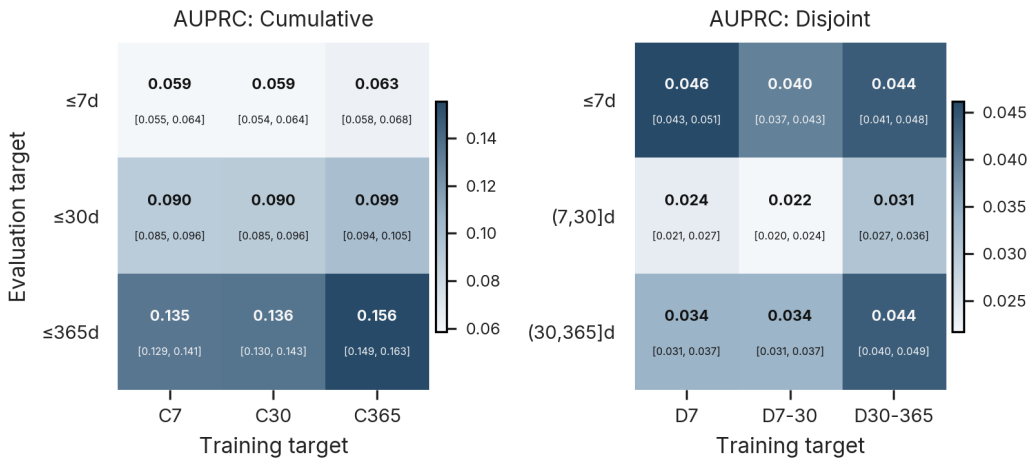


Figure C.1: AUPRC of the critics from Figure 2. Columns are critic training targets and rows are evaluation targets. The left panel reports AUPRC of C7, C30, and C365 on the cumulative evaluation targets ($\leq$ 7d, $\leq$ 30d, $\leq$ 365d) and the right panel reports AUPRC of D7, D7-30, and D30-365 on disjoint evaluation windows ($\leq$ 7d, $(7, 30]$d, $(30, 365]$d). Each disjoint row includes only PRs whose first correction falls in the indicated window and the same 109,129 held-out PRs with no observed correction within a year. Positives from other windows are excluded. Each cell shows the point estimate with a 95% confidence interval from 10,000 bootstrap resamples.

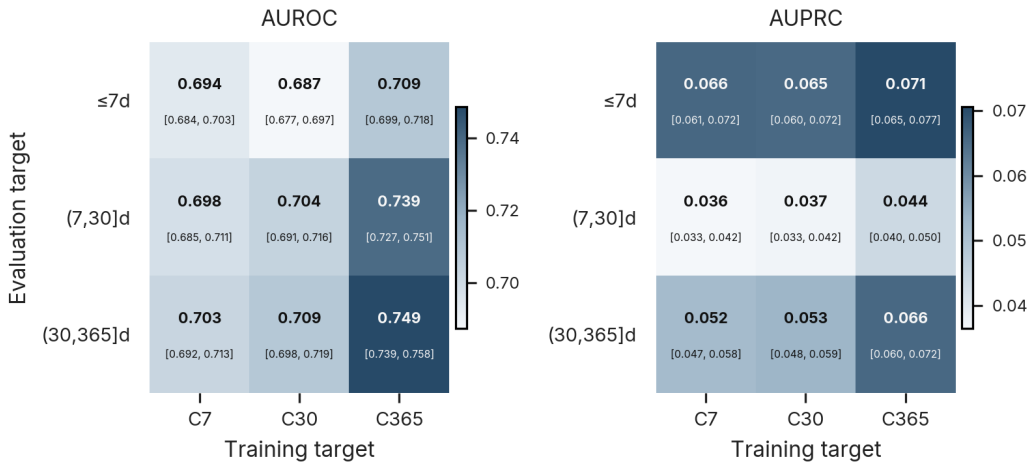


Figure C.2: The cumulative critics from Figure 2, evaluated on disjoint correction windows. Each row includes only PRs whose first correction falls in the indicated window and the same 109,129 held-out PRs with no observed correction within a year. Positives from other windows are excluded. Columns are critic training targets (C7, C30, C365) and rows are evaluation targets ($\leq$ 7d, $(7, 30]$d, $(30, 365]$d). The left panel reports AUROC and the right panel reports AUPRC. Each cell shows the point estimate with a 95% confidence interval from 10,000 bootstrap resamples.

D CORRECTION HORIZONS BY TAXONOMY LABEL

Table D.1: This table reports statistics underlying Figure 3. We report both mean correction horizon and the difference between horizons where a label is present or absent. n counts corrections with the label. Means, differences, and both intervals come from 10,000 bootstrapped draws of the corrections.

Label	n	Mean	95% CI	Δ	95% CI
Issue Source					
Incomplete implementation	2,783	53.0	$[49.9, 56.1]$	$+5.2$	$[+1.3, +8.9]$
Incorrect logic	2,845	52.1	$[49.1, 55.1]$	$+3.7$	$[-0.0, +7.6]$
False assumption	857	45.4	$[40.6, 50.5]$	-4.9	$[-10.1, +0.5]$
Erroneous removal	602	44.1	$[38.3, 50.1]$	-6.1	$[-12.2, +0.1]$
Contract change	437	54.8	$[47.4, 62.5]$	$+5.4$	$[-2.2, +13.3]$
Name collision	69	48.1	$[31.0, 67.2]$	-1.7	$[-18.8, +17.4]$
Other	249	37.3	$[29.4, 45.8]$	-12.8	$[-21.0, -4.1]$
Unknown	275	32.5	$[25.6, 40.0]$	-17.8	$[-25.0, -10.1]$
Reason Not Caught					
Missing or insufficient test	323	51.0	$[42.1, 60.5]$	$+16.0$	$[+6.4, +26.3]$
Suite did not run	130	19.3	$[11.6, 27.9]$	-21.5	$[-30.2, -11.8]$
Environment gap	126	33.9	$[23.7, 45.4]$	-5.3	$[-16.3, +6.8]$
Signal lost	46	39.3	$[20.7, 62.0]$	$+0.6$	$[-18.5, +23.7]$
Human miss	272	30.1	$[23.2, 37.2]$	-10.7	$[-19.0, -2.4]$
Other	75	35.9	$[20.2, 54.0]$	-3.0	$[-19.1, +15.4]$
Unknown	431	42.2	$[35.4, 49.5]$	$+5.1$	$[-3.1, +13.4]$

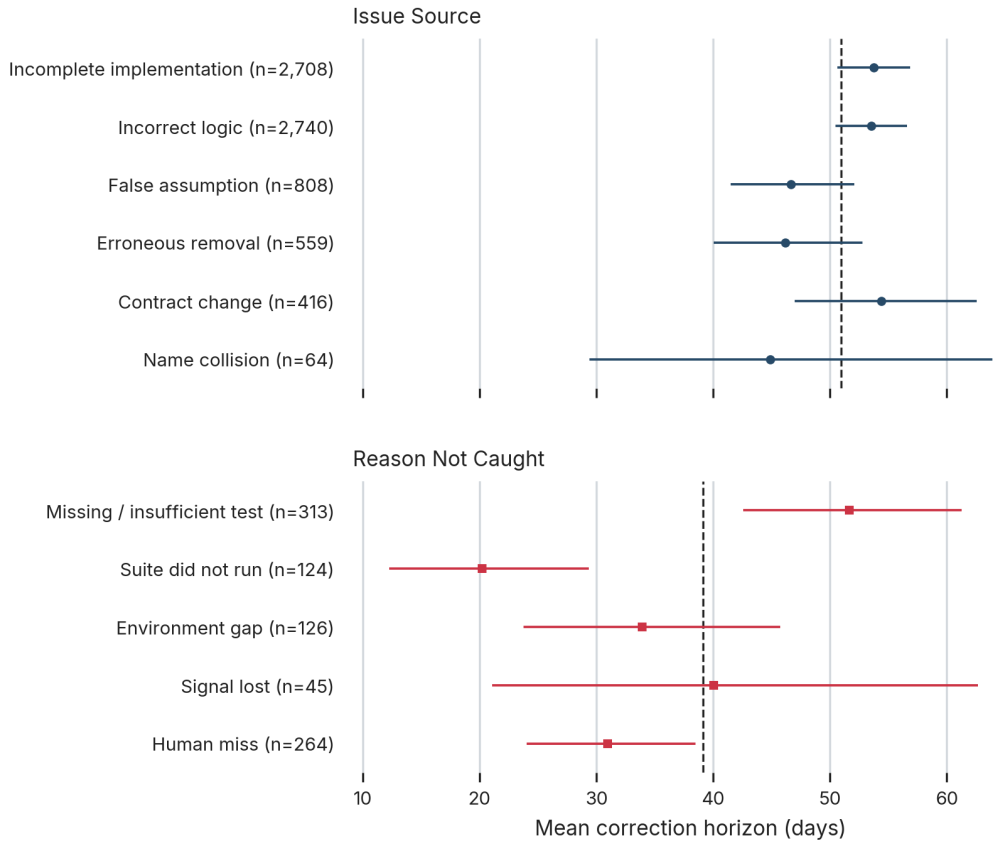


Figure D.1: This figure follows the procedure of Figure 3, but restricts to corrections whose fixing artifact is a pull request or belongs to a parent pull request (fixes that are a product of bare commits are excluded). Of the 13,176 corrections in the main cohort, 901 were first fixed by a bare commit. 785 of those never received a pull-request-backed fix and are excluded from the cohort. The remainder are labeled using the horizon of their earliest pull-request-backed fix instead. The panels therefore cover 7,315 Issue Source and 1,342 Reason Not Caught corrections, against 7,639 and 1,385 in the main text. Results are largely unchanged.

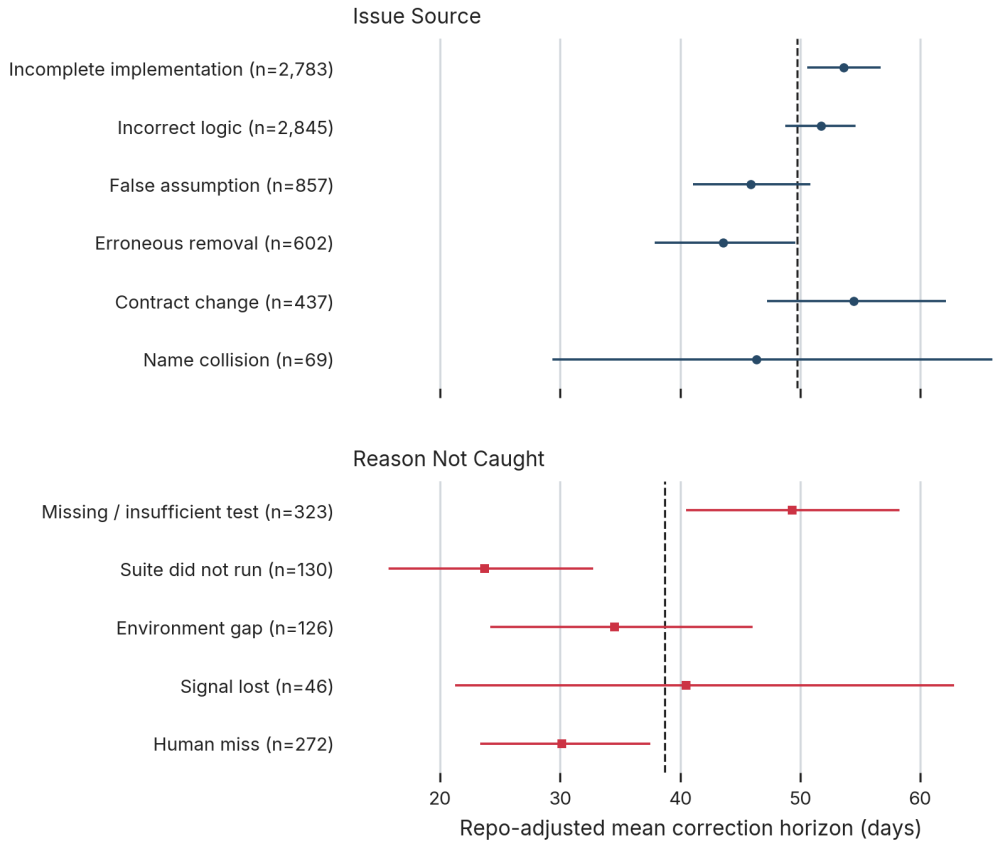


Figure D.2: This figure follows the procedure of Figure 3, but nets repository fixed effects out of every label mean. This way, each label is compared against the repositories that supplied its corrections, instead of against the pooled cohort. Each correction is first measured against the mean horizon of its own repository and the panel mean is then added back. This leads to common units in days and the dashed line still at the panel mean. Confidence intervals come from 10,000 bootstrap draws that resample corrections within each repository, which holds the cohort's repository composition fixed, and redo the adjustment in every draw.

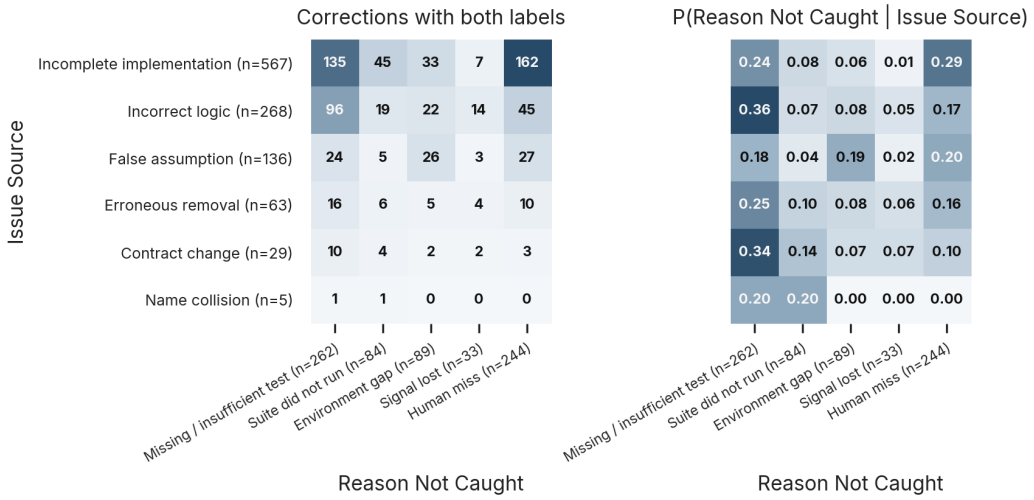


Figure D.3: This figure reports the co-occurrence of the two taxonomies among the 1,061 corrections whose correcting PR stated both an Issue Source and a Reason Not Caught. Rows are Issue Source labels and columns are Reason Not Caught labels; Other and Unknown labels are excluded. Because both fields are multi-label, this is a co-occurrence matrix and not a contingency table: each cell counts the corrections with both labels, so neither rows nor columns sum to the cohort size. Left panel: counts. Right panel: each row normalized to $P(\text{Reason Not Caught} \mid \text{Issue Source})$. Axis labels report each label's individual count within the cohort.

E PROMPTS

GPT-5.6 Terra with low reasoning effort was used for all LLM labeling except PR complexity, which was labeled with GPT-5.4 Nano.

E.1 LLM CORRECTION LABELING PROMPTS

Change References Change

SYSTEM PROMPT

```
You judge how a later change (the source) references an earlier
change (the target) in the same repository, where the source
references the target DIRECTLY in its own text (by commit SHA or by
 #PR number).

Goal: identify cases where the source reverts the target, or fixes
a bug that the target change introduced.

Choose exactly one label:
- revert: the source explicitly reverts the target change or its
commit/merge commit. Dependency or upstream release-note reverts
are dependency_noise.
- corrective_followup: the source fixes, mitigates, or supersedes a
 problem that the target change itself caused. Bare links,
permalinks, "based on", "see also", or "follow-up to" without a
described defect are NOT enough.
- related_discussion: substantively related, but not clearly a
revert or a fix of a bug the target caused.
- dependency_noise: the reference appears in dependency bumps,
generated changelogs, vendored/upstream release notes, or cross-
repo references copied from another project.
- mechanical_merge: backport, cherry-pick, branch sync, release
bookkeeping, or changelog-only cleanup.
- unrelated_reference: points to another project or is incidental
and unrelated to this repo's target change.
- unclear: insufficient evidence.

When the source or target is a merged PR, judge from its PR title/
body; a commit subject like "Merge pull request #X ..." is just a
merge message. Judge only from the provided evidence (commit
messages, PR title/body, review comments). The matched_token field
is the exact reference text found in the source. Do not assume a
defect that the evidence does not describe. Return concise evidence
 and reason grounded only in the provided context.
```

USER PROMPT

```
{
  "relationship": "change_references_change",
  "matched_token": "<exact SHA or \#PR token found in source text
>",
  "is_revert_like": <true|false>,
  "source": {
    "type": "<pull_request|commit>",
    "pr": {
      "number": <int or null>,
      "title": "<PR title>",
      "body": "<PR body>",
      "labels": ["<label>", "..."],
      "merged_at": "<ISO timestamp or null>",
      "author_login": "<login or null>",
      "html_url": "<url or null>"
```

```
    },
    "commit_sha": "<40-char SHA>",
    "commit": {
      "subject": "<commit subject>",
      "message": "<full commit message>",
      "timestamp": "<Unix timestamp or null>",
      "author_name": "<name or null>"
    }
  },
  "target": {
    "pr": {
      "number": <int or null>,
      "title": "<PR title>",
      "body": "<PR body>",
      "labels": ["<label>", "..."],
      "merged_at": "<ISO timestamp or null>",
      "author_login": "<login or null>",
      "html_url": "<url or null>"
    },
    "commit_sha": "<40-char SHA>",
    "commit": {
      "subject": "<commit subject>",
      "message": "<full commit message>",
      "timestamp": "<Unix timestamp or null>",
      "author_name": "<name or null>"
    }
  }
}
```

E.2 LLM TAXONOMY LABELING PROMPTS

Mentions Break Cause

SYSTEM PROMPT

```
You label ONE field from a fixing artifact.
For a pull_request source, use its parent PR title/body and PR
patch. For a commit source, use its commit subject/message and
commit-specific patch, plus parent PR title/body when supplied; a
parent PR patch is never supplied for a commit source.
The context names exactly one TARGET causing PR. A fixing artifact
can mention several prior changes; ignore every change except that
target.

Field: mentions_break_cause
Does the fixing artifact explicitly say why the TARGET causing PR
introduced the break?

- mentions: the title, body, or patch comments state a causal story
 about what the target causing PR did wrong (missed case, inverted
condition, broken invariant, wrong assumption, incomplete migration
, etc.)
- does_not_mention: the artifacts describe the symptom, the repair,
 or that a regression exists, but never explain why that causing PR
 broke
- unknown: empty/boilerplate text, or you cannot tell whether an
explanation is about the named causing PR versus some other change

Decision rules:
- Identity is not explanation. "Follow up to #1234", "regression in
 abcdef", or "test failures introduced in ..." is does_not_mention
unless it also says why that change was wrong.
```

```
- Naming the causing PR or commit is not enough.
- Do not infer a cause from the patch alone. The author must state
it.
- If the only explanation is about a different PR than the target,
choose does_not_mention.
- Prefer does_not_mention over guessing. Use unknown only when the
artifacts are unusable or the target is ambiguous.

Return label, confidence in [0,1], a 1-3 sentence reason, and 1-3
evidence excerpts from the fixing artifact.

Reference identification:
The input's matched_token is the exact reference extracted from the
 fixing artifact that links it to target_causing_pr_number. It may
be a PR reference, or a commit SHA mapped to that PR (including a
merge commit or a commit within the PR). Use matched_token to
locate the passage about the specified target; the fixing text need
 not contain the rolled-up target PR number itself.
The reference identifies which earlier change is being discussed,
not why it broke or why it was not caught. Apply the same evidence
requirements and label definitions above. Do not treat the token,
its mapping to a PR, or this instruction as explanatory evidence.
If the token is absent from the supplied excerpts, do not invent
the missing passage or its explanation. Ignore explanations about
unrelated changes.
```

USER PROMPT

```
{
  "repo": "<owner/name>",
  "target_instruction": "Label only with respect to causing PR #<n>
 ('<causing PR title>'). This fixing artifact may mention other
prior changes; ignore those.",
  "matched_token": "<exact reference extracted from the fixing
artifact>",
  "target_causing_pr_number": <int>,
  "target_causing_pr_title": "<causing PR title>",
  "fixing_source_type": "<pull_request|commit>",
  "fixing_parent_pr_number": <int or null>,
  "fixing_parent_pr_title": "<parent PR title>",
  "fixing_parent_pr_body": "<parent PR body, truncated to 2500
chars>",
  "fixing_commit_sha": "<40-char SHA or null>",
  "fixing_commit_subject": "<commit subject, commit sources only>",
  "fixing_commit_message": "<commit message, truncated to 2500
chars>",
  "has_fixing_text": <true|false>,
  "files_changed": ["<path>", "... up to 40 paths ..."],
  "additions": <int>,
  "deletions": <int>,
  "patch_excerpt": "<fixing patch text, truncated to 4000 chars>"
}
```

Break Reason Types

SYSTEM PROMPT

```
You label ONE field from a fixing artifact.
For a pull_request source, use its parent PR title/body and PR
patch. For a commit source, use its commit subject/message and
commit-specific patch, plus parent PR title/body when supplied; a
parent PR patch is never supplied for a commit source.
```

```
The context names exactly one TARGET causing PR. These artifacts
already state why that causing PR introduced the break. Classify
the stated mechanism(s).

Field: break_reason_types
Mark every label that the author explicitly states. This is multi-
label.

- incomplete_implementation: the intended change is a set of
matching pieces; some were done and some were not (sibling dtype/
caller/backend/constructor/platform; leftover reference after a
correct removal; added a feature and never did the matching update)
. Seen in this cohort: "I forgot to change the test accordingly"
after moving a warper; HGBT docs listed losses and left quantile
out. If the difference between incomplete_implementation and
incorrect_logic is not obvious, mark both.
- incorrect_logic: they edited the place the change belonged; the
intended rule is right; the implementation of that rule is wrong (
wrong formula, inverted boolean, off-by-one, wrong default,
overzealous rewrite, copy-paste into the wrong field, typo at that
site, wrong quantity). Seen in this cohort: header links used --
body-bg instead of --header-link-color. If the difference between
incomplete_implementation and incorrect_logic is not obvious, mark
both.
- false_assumption: the code did what they intended; they state a
false belief about the environment, a dependency, a platform, or an
 invariant
- erroneous_removal: they removed, disabled, or omitted something
that still needed to exist; the story is the missing entity, not a
leftover sibling of a finished change
- contract_change: they changed a caller-visible contract and a
caller that was still supposed to work broke (signature, renamed
public field, hard error where a warning or supported path existed)
- name_collision: colliding or wrong binding (method vs property,
property called as a function, forwarded kwarg the parent does not
accept, import that makes a later lookup hit the wrong object)
- other: they state a real mechanism that is none of the labels
above
- unknown: no usable stated mechanism for the named causing PR

Decision rules:
- A label is on only if the author states that mechanism. Do not
infer from the patch.
- Identity is not a mechanism. "Follow up to #1234" or "regression
in abcdef" without why is unknown.
- Why it was not caught is out of scope.
- Product vs CI/docs/build is not a separate label. Same mechanisms
 apply.
- unknown must be the only label when it applies. Prefer other over
 unknown when the mechanism is clear but unlisted.
- incomplete_implementation vs incorrect_logic: mark the one the
text clearly chooses. If the difference is not obvious, mark both.
Do not pick one at random. Seen as straddles: keep-alive enabled
without discarding unread POST bodies; added 1-d num_frames and
_pad still used shape[1].
- incorrect_logic vs false_assumption: wrong quantity/formula at
the intended site is incorrect_logic even if a coincidence hid it.
Mark false_assumption only if they also state a false belief about
the world.
- leftover sibling of a finished change is
incomplete_implementation; deleting the needed thing itself is
erroneous_removal.
```

```
Return labels, confidence in [0,1], a 1-3 sentence reason, and 1-3
evidence excerpts from the fixing artifact.

Reference identification:
The input's matched_token is the exact reference extracted from the
 fixing artifact that links it to target_causing_pr_number. It may
be a PR reference, or a commit SHA mapped to that PR (including a
merge commit or a commit within the PR). Use matched_token to
locate the passage about the specified target; the fixing text need
 not contain the rolled-up target PR number itself.
The reference identifies which earlier change is being discussed,
not why it broke or why it was not caught. Apply the same evidence
requirements and label definitions above. Do not treat the token,
its mapping to a PR, or this instruction as explanatory evidence.
If the token is absent from the supplied excerpts, do not invent
the missing passage or its explanation. Ignore explanations about
unrelated changes.
```

USER PROMPT

```
{
  "repo": "<owner/name>",
  "target_instruction": "Label only with respect to causing PR #<n>
 ('<causing PR title>'). This fixing artifact may mention other
prior changes; ignore those.",
  "matched_token": "<exact reference extracted from the fixing
artifact>",
  "target_causing_pr_number": <int>,
  "target_causing_pr_title": "<causing PR title>",
  "fixing_source_type": "<pull_request|commit>",
  "fixing_parent_pr_number": <int or null>,
  "fixing_parent_pr_title": "<parent PR title>",
  "fixing_parent_pr_body": "<parent PR body, truncated to 2500
chars>",
  "fixing_commit_sha": "<40-char SHA or null>",
  "fixing_commit_subject": "<commit subject, commit sources only>",
  "fixing_commit_message": "<commit message, truncated to 2500
chars>",
  "has_fixing_text": <true|false>,
  "files_changed": ["<path>", "... up to 40 paths ..."],
  "additions": <int>,
  "deletions": <int>,
  "patch_excerpt": "<fixing patch text, truncated to 4000 chars>"
}
```

Mentions Why Uncaught

SYSTEM PROMPT

```
You label ONE field from a fixing artifact.
For a pull_request source, use its parent PR title/body and PR
patch. For a commit source, use its commit subject/message and
commit-specific patch, plus parent PR title/body when supplied; a
parent PR patch is never supplied for a commit source.
The context names exactly one TARGET causing PR. A fixing artifact
can mention several prior changes; ignore every change except that
target.

Field: mentions_why_uncaught
Does the fixing artifact explicitly say why the break from the
TARGET causing PR was not caught earlier?
```

```
- mentions: the title, body, or patch comments state a detection-
gap story (missing test, review miss, only on a rare input/platform
, CI did not run this path, no coverage for X, slipped through
because ...)
- does_not_mention: the artifacts may add tests or describe the bug
, but never say why it escaped earlier detection
- unknown: empty/boilerplate text, or you cannot tell whether a
detection-gap claim is about the named causing PR

Decision rules:
- Adding a test is not enough. The author must say the gap existed
or why the break was missed.
- "Add regression test" or "update tests" without saying they were
missing/insufficient is does_not_mention.
- Do not infer a missed-test story from the patch. The author must
state it.
- If the only detection-gap claim is about a different PR than the
target, choose does_not_mention.
- Prefer does_not_mention over guessing. Use unknown only when the
artifacts are unusable or the target is ambiguous.

Return label, confidence in [0,1], a 1-3 sentence reason, and 1-3
evidence excerpts from the fixing artifact.

Reference identification:
The input's matched_token is the exact reference extracted from the
 fixing artifact that links it to target_causing_pr_number. It may
be a PR reference, or a commit SHA mapped to that PR (including a
merge commit or a commit within the PR). Use matched_token to
locate the passage about the specified target; the fixing text need
 not contain the rolled-up target PR number itself.
The reference identifies which earlier change is being discussed,
not why it broke or why it was not caught. Apply the same evidence
requirements and label definitions above. Do not treat the token,
its mapping to a PR, or this instruction as explanatory evidence.
If the token is absent from the supplied excerpts, do not invent
the missing passage or its explanation. Ignore explanations about
unrelated changes.
```

USER PROMPT

```
{
  "repo": "<owner/name>",
  "target_instruction": "Label only with respect to causing PR #<n>
 ('<causing PR title>'). This fixing artifact may mention other
prior changes; ignore those.",
  "matched_token": "<exact reference extracted from the fixing
artifact>",
  "target_causing_pr_number": <int>,
  "target_causing_pr_title": "<causing PR title>",
  "fixing_source_type": "<pull_request|commit>",
  "fixing_parent_pr_number": <int or null>,
  "fixing_parent_pr_title": "<parent PR title>",
  "fixing_parent_pr_body": "<parent PR body, truncated to 2500
chars>",
  "fixing_commit_sha": "<40-char SHA or null>",
  "fixing_commit_subject": "<commit subject, commit sources only>",
  "fixing_commit_message": "<commit message, truncated to 2500
chars>",
  "has_fixing_text": <true|false>,
  "files_changed": ["<path>", "... up to 40 paths ..."],
  "additions": <int>,
  "deletions": <int>,
```

```
  "patch_excerpt": "<fixing patch text, truncated to 4000 chars>"
}
```

Why Uncaught Types

SYSTEM PROMPT

```
You label ONE field from a fixing artifact.
For a pull_request source, use its parent PR title/body and PR
patch. For a commit source, use its commit subject/message and
commit-specific patch, plus parent PR title/body when supplied; a
parent PR patch is never supplied for a commit source.
The context names exactly one TARGET causing PR. The binary mention
 tag is noisy: many fixing artifacts only describe the bug.
Classify the stated detection-gap mechanism(s), if any.

Field: why_uncaught_types
Mark every label that the author explicitly states. This is multi-
label.

- missing_or_insufficient_test: the author says the failing case/
path/model was not tested, the test was missing, wrong, overwritten
, or existing tests do not catch this case. Seen in this cohort: "
ogrid was not tested"; Flash Attention 2 + Pixtral "current tests
doesn't catch this case"; ViTImageProcessorFast "accidentally not
added to the test suite".
- suite_did_not_run: an existing CI/test/docs/nightly job did not
execute (skipped, not triggered, fetcher miss, docs-only skip,
merged before CI finished, not retested after the last change).
Seen in this cohort: "CircleCI did not run and I assumed it was
fine"; "I didn't run slow tests before merging"; "gh-22449 was not
retested after the merge"; "tox -e docs is not invoked in CI".
- environment_gap: the suite that ran never hits the failing
environment (GPU, Oracle, 32-bit, Windows, missing dep/backend,
conda vs pip, PyPy, free-threaded). Seen in this cohort: CPU push
CI missed a GPU-only reproducer; "Oracle CI wasn't run"; "main CI
does not have apex installed"; failure "does not run on 32-bit
machines".
- signal_lost: a check ran but the failure was hidden (CI stayed
green, warning swallowed, logger cache, follow-on errors hide the
cause). Seen in this cohort: bash -e missed so CI stayed green
while pytest failed; GitHub Actions reported green when pytest
failed; a concat warning shadowed by another FutureWarning.
- human_miss: the author or reviewer says they forgot, didn't
realize, or missed it in review, and that human-process miss is the
 stated story. Seen in this cohort: "I didn't realize before
merging that we were first uploading the web directory"; "missed
this detail when reviewing #19183".
- other: they state a real detection-gap that is none of the labels
 above. Seen as residuals: a benchmark added only after the change;
 test tolerance/config too aggressive (PT/TF scale_embedding);
warning-filter or lint-config mismatch; pytest yield-collection
quirk.
- unknown: no usable stated detection-gap for the named causing PR.
 The text only describes the bug or the repair, or says it was "
missed somehow" / "CI didn't catch it" with no mechanism.

Decision rules:
- A label is on only if the author states that mechanism. Do not
infer from the patch.
- Adding a test is not a detection-gap. The author must say the gap
 existed or why the break was missed.
```

```
- Why the causing PR broke (the bug mechanism) is out of scope.
- Product vs CI/docs/build is not a separate label. Same mechanisms
 apply.
- unknown must be the only label when it applies. Prefer unknown
over other when no mechanism is stated. Do not use other as a
default. Do not mark unknown together with any other label.
- "CI/tests didn't catch it", "was missed somehow", "previous PR
was wrong", or "thanks X for spotting this" with no further
mechanism is unknown.
- Changing an existing test's tolerance/config because it was too
sensitive is other, not missing_or_insufficient_test. Seen as a
straddle: speech-to-text PT/TF equivalence failed because
scale_embedding amplified diffs.
- A benchmark added after the change is other, not
missing_or_insufficient_test.
- Warning-filter / lint-config / expected-string drift is other,
not signal_lost, unless they say the job stayed green or a warning
was swallowed.
- suite_did_not_run vs environment_gap: mark the one the text
clearly chooses. If they say they did not run the GPU/slow job and
CPU CI ran, mark both. Do not pick one at random.
- If they state both a missing test and that CI skipped the file,
mark both.
- Do not also mark human_miss when they attribute the miss to a
test/CI/environment mechanism. "I forgot to add the test" is
missing_or_insufficient_test only. "I didn't run the tests" is
suite_did_not_run only. "I forgot to check missing backends" is
environment_gap, not human_miss.

Return labels, confidence in [0,1], a 1-3 sentence reason, and 1-3
evidence excerpts from the fixing artifact.

Reference identification:
The input's matched_token is the exact reference extracted from the
 fixing artifact that links it to target_causing_pr_number. It may
be a PR reference, or a commit SHA mapped to that PR (including a
merge commit or a commit within the PR). Use matched_token to
locate the passage about the specified target; the fixing text need
 not contain the rolled-up target PR number itself.
The reference identifies which earlier change is being discussed,
not why it broke or why it was not caught. Apply the same evidence
requirements and label definitions above. Do not treat the token,
its mapping to a PR, or this instruction as explanatory evidence.
If the token is absent from the supplied excerpts, do not invent
the missing passage or its explanation. Ignore explanations about
unrelated changes.
```

USER PROMPT

```
{
  "repo": "<owner/name>",
  "target_instruction": "Label only with respect to causing PR #<n>
 ('<causing PR title>'). This fixing artifact may mention other
prior changes; ignore those.",
  "matched_token": "<exact reference extracted from the fixing
artifact>",
  "target_causing_pr_number": <int>,
  "target_causing_pr_title": "<causing PR title>",
  "fixing_source_type": "<pull_request|commit>",
  "fixing_parent_pr_number": <int or null>,
  "fixing_parent_pr_title": "<parent PR title>",
  "fixing_parent_pr_body": "<parent PR body, truncated to 2500
chars>",
```

```
  "fixing_commit_sha": "<40-char SHA or null>",
  "fixing_commit_subject": "<commit subject, commit sources only>",
  "fixing_commit_message": "<commit message, truncated to 2500
chars>",
  "has_fixing_text": <true|false>,
  "files_changed": ["<path>", "... up to 40 paths ..."],
  "additions": <int>,
  "deletions": <int>,
  "patch_excerpt": "<fixing patch text, truncated to 4000 chars>"
}
```

PR Complexity

SYSTEM PROMPT

```
You label the inherent complexity of an original merged pull
request.
You are given only the PR title/body and the merge diff for that PR
. You are NOT given whether it later broke, whether it was
corrected, or how long anything took.

Label `complexity` for the complexity of understanding and reviewing
 the code change:
- simple: localized, straightforward, or mostly mechanical; a
reviewer can understand the intent and risk quickly.
- moderate: requires understanding one component, a meaningful
behavior change, or several local interactions.
- complex: requires reasoning across components, public semantics,
non-obvious invariants, large design tradeoffs, or subtle
interactions.

Return the single best label, confidence from 0 to 1, and one
concise reason citing evidence from the PR text or diff.
```

USER PROMPT

```
{
  "pull_request": {
    "title": "<PR title>",
    "body": "<PR body, truncated to 2500 chars>",
    "files_changed": ["<path>", "... up to 60 paths ..."],
    "additions": <int>,
    "deletions": <int>,
    "patch_excerpt": "<merge patch text, truncated to 6000 chars>"
  }
}
```

F TAXONOMY EXAMPLES

Each box is one correction from the mention-gated Figure 3 cohort on which the correcting PR named a single mechanism. Title, body, and diff are truncated to the lines that state that mechanism.

F.1 ISSUE SOURCE

Issue Source: Incomplete implementation

CAUSING PR

Link: https://github.com/kubernetes/kubernetes/pull/88372

```
Title: Support injecting errors for FakeImageService

We want FakeImageService to support injecting errors similar to
FakeRuntimeService. These injected errors will be returned on the
next call to the given function.

Diff (truncated):
 type FakeImageService struct {
        FakeImageSize uint64
        Called        []string
+       Errors        map[string][]error
        Images        map[string]*runtimeapi.Image
 }
+func (r *FakeImageService) InjectError(f string, err error) {
+       r.Errors[f] = append(r.Errors[f], err)
+}
```

CORRECTING PR

Link: https://github.com/kubernetes/kubernetes/pull/88437

```
Title: Fix initialization bug in FakeImageService

When adding this functionality in #88372, I forgot to allocate a
map for the Errors field when constructing the object. As a result,
 trying to actually use the InjectError method failed (as I noticed
 when I started trying to write tests to actually use InjectError).

Diff (truncated):
 func NewFakeImageService() *FakeImageService {
        return &FakeImageService{
                Called: make([]string, 0),
+               Errors: make(map[string][]error),
                Images: make(map[string]*runtimeapi.Image),
        }
 }
```

Issue Source: Incorrect logic

CAUSING PR

Link: https://github.com/huggingface/transformers/pull/31546

```
Title: Add torch_empty_cache_steps to TrainingArguments

Following up on https://github.com/huggingface/transformers/pull
/31530, add `torch.cuda.empty_cache()` as an optional `
TrainingArgument` to the training loop every N step, such that those
 who need it to avoid OOMs can use it.

Diff (truncated):
+        if self.torch_empty_cache_steps is not None:
+            if not (isinstance(self.torch_empty_cache_steps, int)
or self.torch_empty_cache_steps > 0):
+                raise ValueError(
+                    f"`torch_empty_cache_steps` must be an integer
bigger than 0, got {self.torch_empty_cache_steps}."
+                )
```

CORRECTING PR

Link: https://github.com/huggingface/transformers/pull/36734

```
Title: Fix `TrainingArguments.torch_empty_cache_steps` post_init
check

PR #31546 introduced the `torch_empty_cache_steps` attribute to `
TrainingArguments`, requiring it to be either a positive integer or
`None`. This constraint is enforced via an explicit check in `
TrainingArguments.__post_init__()`.

However, the current condition incorrectly uses:

`not ([is int] OR [> 0])`

instead of:

`not ([is int] AND [> 0])`

as intended to trigger a `ValueError`.

Diff (truncated):
         if self.torch_empty_cache_steps is not None:
-            if not (isinstance(self.torch_empty_cache_steps, int)
or self.torch_empty_cache_steps > 0):
+            if not (isinstance(self.torch_empty_cache_steps, int)
and self.torch_empty_cache_steps > 0):
                 raise ValueError(
```

Issue Source: False assumption

CAUSING PR

Link: https://github.com/kubernetes/kubernetes/pull/59655

```
Title: Check etcd port instead of process name

currently the pgrep check does not allow any etcd process to exist
other than the one we intend to start in our scripts. All we need
to know is if the port is free. So let us use netstat to check if
anyone is using that port.

Diff (truncated):
-  # validate it is not running
-  if pgrep -x etcd >/dev/null 2>&1; then
+  # validate etcd port is free
+  if netstat -nat | grep "[\.:]${ETCD_PORT:?} .*LISTEN" >/dev/null
 2>&1; then
```

CORRECTING PR

Link: https://github.com/kubernetes/kubernetes/pull/59852

```
Title: Check if netstat or iproute2 is available

In https://github.com/kubernetes/kubernetes/pull/59655, I
incorrectly assumed that the kubetest image would have net-tools.
It doesn't. This checks and uses `ss` from iproute2, *or* `netstat`
from net-tools. The kubetest image does appear to have iproute2.

Diff (truncated):
-  if netstat -nat | grep "[\.:]${ETCD_PORT:?} .*LISTEN" >/dev/null
 2>&1; then
+  local port_check_command
+  if command -v ss &> /dev/null && ss -Version | grep 'iproute2'
&> /dev/null; then
+    port_check_command="ss"
+  elif command -v netstat &>/dev/null; then
+    port_check_command="netstat"
+  else
+    kube::log::usage "unable to identify if etcd is bound to port
${ETCD_PORT}. unable to find ss or netstat utilities."
+    exit 1
+  fi
```

Issue Source: Erroneous removal

CAUSING PR

Link: https://github.com/numpy/numpy/pull/21887

```
Title: BLD: Allow GCC compile on mingw-w64-based systems

Macro __MINGW32__ always defined for Mingw-w64 compilers (32- or
64-bit).

Diff (truncated):
 #include <inttypes.h>

-#ifdef _WIN32
+#if defined(_WIN32) && !defined (__MINGW32__)
 #include <stdlib.h>
 #define inline __forceinline
 #endif
```

CORRECTING PR

Link: https://github.com/numpy/numpy/pull/22242

```
Title: BLD: add back stdlib.h include in pcg64.h

This seems to have been removed by accident in #21887. stdlib.h is
required for _rotr64() further down.

Diff (truncated):
 #include <inttypes.h>

-#if defined(_WIN32) && !defined (__MINGW32__)
+#ifdef _WIN32
 #include <stdlib.h>
+#endif
+
+#if defined(_WIN32) && !defined (__MINGW32__)
 #define inline __forceinline
 #endif
```

Issue Source: Contract change

CAUSING PR

Link: https://github.com/pytest-dev/pytest/pull/3560

```
Title: Add a default value to CallInfo.result

Fixes #3554

Diff (truncated):
          self.when = when
          self.start = time()
+         self.result = None
          try:
              self.result = func()
```

CORRECTING PR

Link: https://github.com/pytest-dev/pytest/pull/3562

```
Title: Revert change of Callinfo.result default value

As discussed in #3560, this should not go to master because this
breaks the API.

Diff (truncated):
          self.when = when
          self.start = time()
-         self.result = None
          try:
              self.result = func()
```

Issue Source: Name collision

CAUSING PR

Link: https://github.com/huggingface/transformers/pull/18499

```
Title: Update feature extractor methods to enable type cast before
normalize

At the moment, the return type of our feature extractors isn't
always as expected or sometimes fails if a `do_xxx` config flag is
set to `False`. This PR introduces the necessary changes to the `
ImageFeatureExtractionMixin` methods such that we can modify the
feature extractor calls to fix this.

Diff (truncated):
+    def rescale(self, image: np.ndarray, scale: Union[float, int])
 -> np.ndarray:
+        """
+        Rescale a numpy image by scale amount
+        """
+        self._ensure_format_supported(image)
+        return image * scale
```

CORRECTING PR

Link: https://github.com/huggingface/transformers/pull/18677

```
Title: Rename method to avoid clash with property

Fixes a bug when the `self.rescale` method was being called, but `
self.rescale` was set as a property. This arose after a refactoring
of the feature extractors in my PR: https://github.com/huggingface/
transformers/pull/18499, causing an issue in any model that had `
rescale` as a property in its config. Renaming considered safe as
the new `rescale` method was introduced yesterday.

Diff (truncated):
-    def rescale(self, image: np.ndarray, scale: Union[float, int])
 -> np.ndarray:
+    def rescale_image(self, image: np.ndarray, scale: Union[float,
 int]) -> np.ndarray:
         """
         Rescale a numpy image by scale amount
         """
```

F.2 REASON NOT CAUGHT

Reason Not Caught: Missing / insufficient test

CAUSING PR

Link: `https://github.com/rails/rails/pull/18322`

```
Title: Add ActiveModel::Errors#details

To be able to return type of validator, one can now call details on
 Errors instance.

Diff (truncated):
     def delete(key)
       messages.delete(key)
+      details.delete(key)
```

CORRECTING PR

Link: `https://github.com/rails/rails/pull/19465`

```
Title: Fix ActiveModel::Errors#delete return value to stay backward
 compatible

Rails 5.0 changes to ActiveModel::Errors include addition of
details that also accidentally changed the return value of delete.
Since there was no test for that behavior it went unnoticed. This
commit adds a test and fixes the regression.

Diff (truncated):
     def delete(key)
-      messages.delete(key)
       details.delete(key)
+      messages.delete(key)
     end
```

Reason Not Caught: Suite did not run

CAUSING PR

Link: https://github.com/scikit-learn/scikit-learn/pull/16193

```
Title: Fix pre-build checks to handle compilers from build_ext
options

There are two ways to specify the compiler to build the project:
- CC=<compiler> python setup.py build_ext -i
- python setup.py build_ext --compiler=<compiler> -i
Currently the checks before building sklearn ignore the compiler
specified through the second option.

Diff (truncated):
+def _get_compiler():
+    """Get a compiler equivalent to the one that will be used to
build sklearn"""
+    cmd_opts = dist.command_options.get('build_ext')
 def compile_test_program(code, extra_preargs=[], extra_postargs
=[]):
-    ccompiler = new_compiler()
+    ccompiler = _get_compiler()
```

CORRECTING PR

Link: https://github.com/scikit-learn/scikit-learn/pull/17887

```
Title: BLD fix CI

#16193 broke the CI

We merged it while the CI did not run on it for a long time. Take
home message for myself: "Never assume unexpected errors won't
happen"

Diff (truncated):
 import tempfile
 import textwrap
+import setuptools  # noqa
 import subprocess
```

Reason Not Caught: Environment gap

CAUSING PR

Link: https://github.com/huggingface/transformers/pull/18961

```
Title: Add AnyPrecisionAdamW optimizer

Add `AnyPrecisionAdamW` optimizer from `torchdistx`

Diff (truncated):
         optim_test_params.append(
             (
-                OptimizerNames.ADAMW_APEX_FUSED,
+                TrainingArguments(OptimizerNames.ADAMW_APEX_FUSED,
 output_dir="None"),
                 apex.optimizers.FusedAdam,
                 default_adam_kwargs,
```

CORRECTING PR

Link: https://github.com/huggingface/transformers/pull/20454

```
Title: [trainer] apex test fix

there was a small mistake in a test of https://github.com/
huggingface/transformers/pull/18961, this PR fixes it.

the main CI doesn't have apex installed that's why it missed it.

Diff (truncated):
         optim_test_params.append(
             (
-                TrainingArguments(OptimizerNames.ADAMW_APEX_FUSED,
 output_dir="None"),
+                TrainingArguments(optim=OptimizerNames.
ADAMW_APEX_FUSED, output_dir="None"),
                 apex.optimizers.FusedAdam,
                 default_adam_kwargs,
```

Reason Not Caught: Signal lost

CAUSING PR

Link: https://github.com/pandas-dev/pandas/pull/23924

```
Title: TST/CI: Unify CI test scripts (single and multi) and
simplify the system

The goal of this PR is:
- Avoid repetitive code by merging `ci/script_single.sh` and `ci/
script_multi.sh`

Diff (truncated):
diff --git a/ci/run_tests.sh b/ci/run_tests.sh
new file mode 100755
--- /dev/null
+++ b/ci/run_tests.sh
+#!/bin/bash
+
+if [ "$DOC" ]; then
+    echo "We are not running pytest as this is a doc-build"
+    exit 0
+fi
```

CORRECTING PR

Link: https://github.com/pandas-dev/pandas/pull/24075

```
Title: CI/TST: Making ci/run_tests.sh fail if one of the steps fail

Looks like when simplifying the running of the tests in the CI
(#23924), I missed the `-e` in the bash header. And that makes the `
ci/run_tests.sh` exit with status code 0, even if the calls to
pytests fail.

This left the CI in green, even when tests fail for the last 3 days
 (sorry about that). I think nothing is broken.

Diff (truncated):
 #!/bin/bash

+set -e
+
 if [ "$DOC" ]; then
     echo "We are not running pytest as this is a doc-build"
     exit 0
```

Reason Not Caught: Human miss

CAUSING PR

Link: https://github.com/rust-lang/rust/pull/142416

```
Title: Assorted bootstrap cleanups (step 2)

Very small improvements designed towards making bootstrap tests
less hacky/special, and towards making it possible to run bootstrap
 tests in parallel.

Diff (truncated):
 #![allow(dead_code)]

+#[cfg(test)]
+mod tests;
+
 use std::env;
```

CORRECTING PR

Link: https://github.com/rust-lang/rust/pull/144445

```
Title: Fix ./x check bootstrap (again)

Redoes #134883 and reverts a commit from #142416. Unfortunately I
missed this during review.

- Commit 1: Reverts the nested tests module and re-applies #134883.
- Commit 2: Check ./x check bootstrap in pr-check-1 to catch
obvious problems like this.

Diff (truncated):
-#[cfg(test)]
-mod tests;
+// If we were to declare a tests submodule here, the shim binaries
 that
+// include this module via #[path] would fail to find it, which
breaks
+// ./x check bootstrap.
```