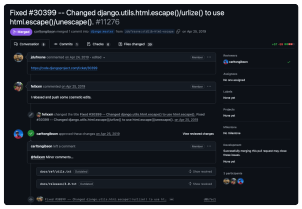
CR-BENCH: EVALUATING THE REAL-WORLD UTILITY OF AI CODE REVIEW AGENTS

Kristen Pereira*, **Neelabh Sinha***, **Rajat Ghosh, Debojyoti Dutta**
Nutanix, Inc.
{kristen.pereira, neelabh.sinha,
rajat.ghosh, debojyoti.dutta}@nutanix.com

ABSTRACT

Recent advances in frontier large language models have enabled code review agents that operate in open-ended, reasoning-intensive settings. However, the lack of standardized benchmarks and granular evaluation protocols makes it difficult to assess behavior of code review agents beyond coarse success metrics, particularly for tasks where false positives are costly. To address this gap, we introduce `CR-Bench`, a benchmarking dataset, and `CR-Evaluator`, a fine-grained evaluation pipeline for code review agents. Using these tools, we conduct a preliminary study evaluating both a single-shot agent and a Reflexion-based agent across two frontier models. We find that code review agents can exhibit a low signal-to-noise ratio when designed to identify all hidden issues, obscuring true progress and developer productivity when measured solely by resolution rates. Our analysis identifies the hidden trade-off between issue resolution and spurious findings, revealing a frontier that constrains effective agent design. Together, `CR-Bench` and `CR-Evaluator` provide a timely foundation for studying and developing code review agents as LLM-based systems transition from controlled benchmarks to real-world software engineering workflows.

1 INTRODUCTION



(a) Pull Request (PR)

```
trail  = middle[len(stripped):] + trail
middle = middle[:len(stripped) - len(middle_unescaped)]
```

```
punctuation_count = len(middle_unescaped) - len(stripped) trail = middle[-punctuation_count:] + trail middle = middle[:-punctuation_count]
```

(b) Patch (r_δ)

'trim_punctuation() mis-slices the original (escaped) URL token when stripping trailing punctuation from the unescaped form. It computes `stripped` on `middle_unescaped` but then uses incorrect indices on `middle`:\n\n```py\ntrail = middle[len(stripped):] + trail\nmiddle = middle[:len(stripped) - len(middle_unescaped)]\n```\n\nBecause `len(middle_unescaped)` can differ from `len(middle)` (e.g., '&lt'), this moves the wrong substring into `trail`, leaving extra characters like 'lt' outside the link and corrupting output.'

(c) Review Comment (r_c)

Figure 1: Example of an instance from `CR-Bench` dataset: In a pull request, a comment should be made to address the specified bug. The fix is removing the lines of code marked in RED and adding the ones marked in GREEN.

Code review is a challenging task to automate (Yang et al., 2024; Bacchelli & Bird, 2013), as it lacks a well-defined and universally accepted evaluation signal, unlike tasks such as compilation or unit testing where objective pass/fail metrics exist and verification is possible. Further, code review inherently is a complex task due two reasons. First, the code suggestions made during review can be documentation-based, stylistic, refactoring, which are very subjective as per practices followed by different teams. Second, even the comments can be addressed in multiple ways, and there is no uniquely correct way of doing it.

*Equal contribution

Due to the complexity of the problem, there develops an inherent bias in code review agents (CodeRabbit, 2024; Anysphere, 2024; Google Cloud, 2024; Tang et al., 2024). We have observed that these agents often face a fundamental trade-off, they either prioritize precision and risk missing critical vulnerabilities, or prioritize recall at the cost of producing noisy and low-actionable feedback. Gathering feedback of live running agents like Coderabbit (CodeRabbit, 2024), Gemini code assist (Google Cloud, 2024) from engineers have revealed that more noise in comments hampers developer productivity, inhibiting adoption of agents in real-world code review process. But, avoiding them risks missing defects that might be critical. Consequently, the systematic development and comparison of code review agents necessitates a standardized and principled code review benchmark.

Existing code review benchmarking datasets often suffer from two primary limitations. First, they frequently combine objective logic errors with subjective stylistic preferences (Zeng et al., 2025b; Sghaier et al., 2025). While aspects like style, documentation and formatting are important, they can often better managed by static analysis tools or linting hooks rather than expensive LLM inference. Second, many benchmarks rely on synthetic or small-scale problems (Tufano et al., 2021; 2022; Thongtanunam et al., 2022) that fail to capture the multi-file dependencies and complexities of large-scale repositories. We argue that an effective code review agent must prioritize defect-identifying reviews. These are reviews that target functional, performance, reliability, or security regressions. We argue to focus on this subset for three primary reasons: (1) unaddressed defects directly introduce system faults, and are critical to identify; (2) defect detection is inherently more objective and thus better suited for automated evaluation; and (3) stylistic or structural changes are often subjective and can vary for different projects/teams/working groups.

To address these gaps, we introduce `CR-Bench` – a novel benchmarking dataset designed to evaluate the reasoning and defect-detection capabilities of code review agents. In addition, we create `CR-Evaluator`, an evaluation agent that takes a PR, the reviews generated by any black-box agent, and determines its performance, as well as how effective they will be for developers. For PRs in `CR-Bench`, it is important to identify the underlying bugs. But, a generalized code review agent will also give additional comments, which may, or may not be useful. Therefore, alongside standard metrics like precision, recall, and F1-score of finding a bug, we also introduce two additional metrics – *usefulness rate* and *signal-to-noise ratio*, which drive measuring utility and developer acceptance of code review agents running in production workflows. This will help in understanding their accuracy, trustworthiness, and factuality. We evaluate two agents using our framework – one is a single-shot LLM which takes the PR's diff in context and returns reviews on it. Another is a Reflexion (Shinn et al., 2023) agent that argues and iteratively builds/removes reviews on the PR before making suggestions. Using our experiments, we demonstrate that if we pressure an agent to identify more bugs (like Reflexion), the noise increases, and if we make it too relaxed (like single LM), some bugs are missed. A good agent should fall in a delicate sweet spot between this spectrum.

Our key contributions are as follows:

- `CR-Bench`, a benchmark for automated code review focusing on real-world, preventable defects, labeled with tags covering bug category, impact, and severity.
- `CR-Evaluator`, a verifier agent that can effectively evaluate a code review agent beyond its accuracy, to trustworthiness, developer acceptability and factuality.
- Evaluation of two complementary prompting paradigms with two state-of-the-art LLMs to demonstrating performance comparisons and key trade-offs between review coverage and integrity present in code review agents.

In the rest of the paper, we discuss these in detail. Section 2 discusses related work, Section 3 formulates the problem. Section 4 details the `CR-Bench` generation methodology, and Section 5 presents the `CR-Evaluator`. The experiments and results are discussed in Section 6 and 7 respectively. Finally, Section 8 concludes the paper and Section 9 discusses future work.

2 Related Work

Code Review Agents. Code review agents have evolved from simple code refinement (Tufano et al., 2021; 2022) and comment generation (Tang et al., 2024) to AI agents capable of multi-step reasoning, cross-file interaction, and tool usage. Modern commercial platforms like CodeRabbit (CodeR-

Method	# Tasks	Multiple Diffs	Full PR Context	Categorization	Defect-focused	Evaluation
Tufano et al., 2021	1718	✗	✗	✗	✗	Exact/BLEU
Tufano et al., 2022	147,533	✗	✗	✗	✗	Exact/BLEU
Thongtanunam et al., 2022	$17K$	✗	✗	✗	✗	Exact
Li et al., 2022	$54K$	✗	✗	✗	✗	CodeBLEU
Zeng et al., 2025b	1000	✓	✓	✓	✗	LLM Judge (P, R, F1)
CR-Bench (Ours)	584	✓	✓	✓	✓	LLM Judge (P, R, F1, U, SNR)
CR-Bench-Verified (Ours)	174	✓	✓	✓	✓	LLM Judge (P, R, F1, U, SNR)

Table 1: Comparison of Benchmarks for Automated Code Review Evaluation. The data instances in `CR-Bench` are more defect-focused and real bugs found existing in large-scale open-source repositories. Also, alongside measuring the performance traditionally with precision, recall, F1, we also introduce developer trust and factuality parameters with usefulness score (U), and signal-to-noise ratio (SNR).

abbit, 2024), and Cursor Bugbot (Anysphere, 2024) are code review products that actively analyze architectural impact, suggest complex changes, and even interact with external project management tools. Similarly, frontier models, including Gemini Code Assist (Google Cloud, 2024) and Claude Code (Anthropic, 2024) also have their code review tool. Open-source initiatives such as PR-Agent (Qodo AI, 2024) have further democratized these capabilities. These agents have moved from experimental laboratory tools to mainstream production components. With such wide usage, it is important to evaluate them correctly and determine best available option. This brings a need for good quality benchmark.

Automated Code Review Benchmarks. Early approaches attempted to solve code review by training RNNs (Tufano et al., 2021) and transformers models (Thongtanunam et al., 2022; Tufano et al., 2022) to translate buggy methods into fixed ones. However, these works operated at a restrictive method-level granularity, relying on code abstraction or truncation that omits critical cross-file context. They also evaluated performance using static text-similarity metrics (e.g., BLEU (Papineni et al., 2002)) that fail to capture functional correctness. CodeReviewer (Li et al., 2022) attempted to specifically pre-train a model optimized for diff-hunks, yet it remains limited to local context and static evaluation. Recently, SWR-Bench (Zeng et al., 2025b) advanced the field by providing full Pull Request (PR) contexts and evaluating comments using LLM-as-a-judge (Liu et al., 2023). But, it focuses primarily on identifying hit metrics by comparison against human-verified change-points, thus invariably incorporating changes which can be very subjective in nature when compared to objective errors that cause functional errors. In contrast, `CR-Bench` is the first benchmark to focus only on objective defect-detection with full PR context for agentic evaluation. Table 1 presents a novelty positioning table for `CR-Bench`. We discuss the formalism of our problem in next section.

3 FORMALISM

Code Review Task Definition. The *Code Review* is an asynchronous peer-evaluation mechanism aimed at assessing the code quality and recommending possible improvements for a base code state (B). It starts with a *Pull Request* (PR) – a proposal to integrate a sequence of code modifications into a target codebase. Formally, we define a PR as a tuple $(\mathcal{C}, \Delta, \mathcal{M})$, where $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ is a set of discrete *commits*, Δ represents the cumulative code diff (patch), and $\mathcal{M}$ denotes the associated metadata (e.g., title, description, and author). Each commit $c \in \mathcal{C}$ constitutes a cryptographic snapshot of the repository state, capturing atomic changes to the directory structure and file contents. The output for a code review process is a set of m review blocks: $\mathcal{R} = \{(r_n, r_\delta)_i \mid 1 \leq i \leq m\}$. r_n is a natural language recommendation, which identifies specific defects or suggests improvements, and r_δ provides a concrete code-level resolution to the identified issue. The task of a code review agent is to take a pair of (PR, B) as input and generate a set of m reviews, $\mathcal{R}^m$, as shown in Equation 1.

$$(PR, B) \xrightarrow{\text{Code Review Agent}} \mathcal{R}^m \tag{1}$$

Bias-Variance Trade-off. Code review tasks,including documentation checks, coding style enforcement, and structural consistency verification, often involve substantial subjectivity (Zeng et al., 2025a). In practice, a given change can admit multiple valid review interpretations, and this vari-

ability increases with the size and complexity of the underlying repository. Such subjectivity can introduce review noise, making reliable automation challenging. Consequently, designing automated code review agents is a non-trivial task. An overly conservative agent may fail to flag important issues, while an overly permissive agent may generate excessive or low-quality feedback. This reflects a pronounced bias–variance trade-off that represents a central design challenge for automated code review systems.

Scope. This work prioritizes defect-identifying reviews, those targeting functional, performance, reliability, or security regressions. We focus on this subset for three primary reasons: (1) unaddressed logic errors directly introduce system faults; (2) defect detection is inherently more objective and thus better suited for automated evaluation by LLM-based agents; and (3) stylistic or structural preferences are often subjective and better managed via linting tools or human judgment. Following section introduces the dataset to facilitate such study.

4 CR-BENCH: BENCHMARKING DATASET FOR CODE REVIEW

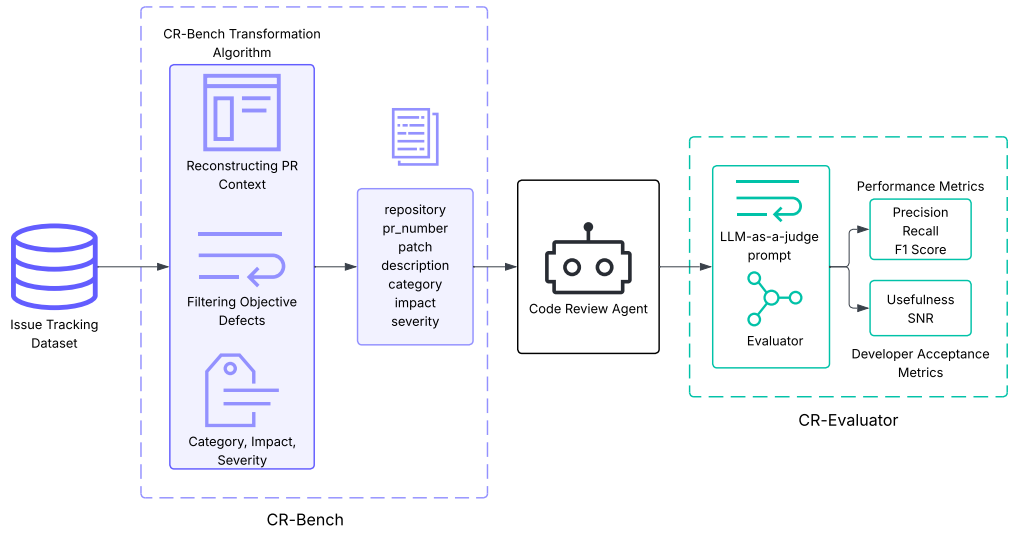


Figure 2: `CR-Bench` is a dataset for transforming real-world software defects into an objective code review benchmark with full Pull Request (PR) context and a multi-dimensional taxonomy of Category, Impact, Severity. `CR-Evaluator` is a method of evaluating code review agents and measuring them in terms of both performance and developer acceptance.

We offer `CR-Bench`, a code review dataset $(PR, B) \rightarrow \mathcal{R}^m$. We derived this dataset using SWE-Bench (Jimenez et al., 2024) as our foundation. SWE-Bench contains real GitHub issues at repository-scale along with the generated PRs that fix them. We tranformed SWE-Bench dataset into a code review dataset using Algorithm 1.

Unlike SWE-Bench, code review focuses on analyzing pull requests (PRs). A PR is a proposed code change that reviewers must evaluate without knowing in advance where bugs may exist or what type of problems might be introduced. The systematic transformation algorithm to convert SWE-Bench into a code review benchmark, is summarized in Figure 2 in addition to Algorithm 1.

During dataset construction, we apply several validation checks to ensure that the bugs included are realistically detectable through code review. We also introduce a taxonomy that labels each instance by bug category, impact, and severity. This allows us to measure how well models and agents perform across different types of bugs and levels of risk. We share two variations – `CR-Bench`, which is generated from SWE-Bench, and `CR-Bench-verified` from SWE-Bench-verified. `CR-Bench-verified` is also verified by us manually for quality.

4.1 TRANSFORMATION OF SWE-BENCH TO CR-BENCH

Algorithm 1 CR-Bench Transformation Algorithm

Require: SWE-Bench $\mathcal{S}$, LLM $\mathcal{L}$
Ensure: CR-Bench $\mathcal{D}$
1: $\mathcal{D} \leftarrow \emptyset$
2: **for** each $i \in \mathcal{S}$ **do**
3: $\quad \mathcal{C} \leftarrow$ blame(checkout($i.commit_{base}$))
4: $\quad \mathcal{PR} \leftarrow$ GitHubAPI($\mathcal{C}$)
5: $\quad$ **if** $|\mathcal{PR}| \neq 1$ **then**
6: $\quad\quad$ **continue**
7: $\quad$ **end if**
8: $\quad$ *detectable* $\leftarrow \mathcal{L}(\mathcal{P}^{\mathcal{C}}(i.patch))$
9: $\quad$ **if** $\neg$*detectable* **then continue**
10: $\quad$ **end if**
11: $\quad cmt \leftarrow$ Paraphrase($i.desc$)
12: $\quad tags \leftarrow \mathcal{L}(P^{T}(i.desc, i.patch))$
13: $\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{\mathcal{PR}, i.patch, cmt, tags\}$
14: **end for**
15: **return** $\mathcal{D}$

To create `CR-Bench`, we start with the SWE-Bench dataset and follow Algorithm 1. For each instance, we first check out to the initial state using the base commit, and execute `git blame`[1] on the lines shown removed in SWE Bench's patch. This is to extract all the commit IDs that added those lines (Step 3). Using these commit IDs, we use the GitHub APIs to get the pull request (PR) [2] that was associated with that commit (Step 4). If there are multiple PRs, then, the issue couldn't have been identified in a single PR, so, we discard them (Step 5). From here, we have the PR, commit, and diff patch of the code review problem.

Using the remaining task instances, we want to know if the bugs could have been detected during the PR Review. We define detectable bugs as the ones which stem from a defect introduced that can be reasonably identified during the original code review through logic inspection, boundary testing, or adherence to API contracts. This is done by classification using an LLM and prompt $\mathcal{P}^{\mathcal{C}}$(Step 9), and non-preventable bugs are discarded. The result becomes our final set. The prompt for this step is given in Listing 1 in Appendix A.

As per the definition of code review, along with the PR and patch, we also need a review comment. To obtain this, we paraphrase the problem definition (using prompt listing 2, Appendix A) of the SWE-Bench dataset into a bug description (Step 11). This can either be used directly or further modified by the users to suit their evaluation style and conventions.

Our approach may not provide an exhaustive set of all issues for a given PR. But, we prioritize having high-quality, verified defects that are free from data instances which are falsely included.

4.2 TAXONOMY

For the instances in the candidate set, we tag the reviews into buckets that can enable performance analysis of LLMs and code review agents across these. We create three buckets – category, impact, and severity. Similar to (Islam et al., 2019), we define category as the root cause of the bug, and the impact as the subsequent effect the bug can potentially cause. We define severity as the extent of impact of the bug on the system. For the category, we use an accepted taxonomy (Beizer, 1984) and divide the tasks into – Structural Bugs, Interface, Integration and System (IIS) Bugs, Requirements, Features and Functionality (RFF) Bugs, Data Bugs, Concurrency Bugs, Memory Bugs, Security Bugs, and Coding Bugs. For impact, we use the ISO/IEC 25010 standard (International Organization for Standardization & International Electrotechnical Commission, 2023) and divide it

[1] https://git-scm.com/docs/git-blame

[2] https://docs.github.com/en/rest/commits/commits?apiVersion=2022-11-28#list-pull-requests-associated-with-a-commit

Taxonomy	Elements	Source
Root Cause Category	Structural, IIS, RFF, Data, Concurrency, Memory, Security, Coding	(Beizer, 1984)
Impact	Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, Portability	ISO Standard
Severity	Low, Medium, High	—

Table 2: The `CR-Bench` taxonomy elements for classifying code reviews (Note: RFF = Requirements, Features and Functionality, IIS = Interface, Integration and System).

into Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, and Portability. Severity is divided into Low, Medium and High.

In order to generate tags for the task instances, we use the problem definition ($desc$) and fix patch with a tag generation prompt $\mathcal{P}^{\mathcal{T}}$ (Listing 3, Appendix A) and make an LLM call to get the tags (Step 12). The prompt also elaborates the definition of each of the items in all three buckets. Statistics of the generated dataset is given in Section 7.1. The next section describes the evaluation methodology to evaluate any code review agent using this dataset.

5 CR-EVALUATOR

`CR-Evaluator` uses an LLM-as-a-judge (Liu et al., 2023) approach that employs a zero-shot classification prompt to categorize every review generated by a candidate review model against the historical gold standard defect. `CR-Evaluator` is presented with the known bug description, the specific files modified in the eventual fix, and the candidate review model's comment. It then performs a discrete classification into one of three categories:

- **Bug Hit.** This category corresponds to reviews that accurately identify or directly relate to the specific logic error described in the ground truth.
- **Valid Suggestion.** This category corresponds to reviews with constructive feedback such as stylistic improvements, performance optimizations, or edge-case handling. These reviews are technically sound and significant, but not related to the primary defects in the ground truth set.
- **Noise.** This category corresponds to reviews that are factually incorrect, irrelevant to code changes, or insignificant, often indicating a model hallucination.

These three categories together form the total generated review. They are mutually exclusive and collectively exhaustive representations of the review space. By aggregating these classifications across the entire corpus, `CR-Evaluator` generates a multi-dimensional performance profile for any black-box review agent. The first profile metric is *Recall* , which measures the bug coverage rate.

$$\text{Recall} = \frac{\text{Total Bug Hits}}{\text{Total Bugs}} \tag{2}$$

To assess the predictive accuracy of a code review agent, we use Precision:

$$\text{Precision} = \frac{\text{Total Bug Hits}}{\text{Total Reviews}} \tag{3}$$

The harmonic mean of these two values provides the overall F1 Score:

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

To factor in Valid Suggestions as a dissertate for a code agent, we modify the traditional precision metric and introduce a new metric, *Usefulness Rate* in Equation 5:

$$\text{Usefulness Rate} = \frac{\text{Total Bug Hits} + \text{Total Valid Suggestions}}{\text{Total Reviews}} \tag{5}$$

Clearly, Usefulness Rate $>$ Precision. It accounts for the benefits of Code Review agents beyond the bug discovery.

Among three review categories, Bug Hit and Valid Suggestion clearly measure the beneficial signals, while Noise measure the undesirable part. A good code agent should maximize the $(\text{Bug Hit} + \text{Valid Suggestion})$ count and minimize the Noise count. This relative conflict can measured by *signal-to-noise ratio (SNR)*:

$$\text{SNR} = \frac{\text{Total Bug Hits} + \text{Total Valid Suggestions}}{\text{Total Noise Count}} \tag{6}$$

A high SNR serves as a primary proxy for developer trust by quantifying the ratio of actionable signal to distracting hallucinations. It is a critical diagnostic for identifying agents/model that achieve high recall through high-volume output. A low ratio can trigger developer fatigue and eventual tool abandonment in production workflows.

6 EXPERIMENTS

Using the `CR-Evaluator`, we evaluate two agents and do a comparative study of their performance. The two agents are:

1. **Single-shot LM** (A_1): This agent follows a direct zero-shot strategy where the model identifies potential bugs, logic issues, and vulnerabilities in a single pass based on the PR diff and description. It provides a baseline for the model's raw intuition by requiring concise, JSON-formatted feedback on specific file paths and line numbers without further iteration (prompt given in listing 4, Appendix B). This single-shot agent is loosely based on (Qodo AI, 2024).
2. **Reflexion Agent** (A_2): Utilizing the Reflexion framework (Shinn et al., 2023), this agent performs an initial analysis followed by an iterative self-improvement loop where it is explicitly prompted to discover missed bugs (false negatives) and refine existing comments. This multi-stage approach prioritizes thoroughness and diagnostic accuracy, tasking the agent to re-examine the code to identify overlooked logic errors, security flaws, or resource leaks before finalizing the review (prompt given in listing 5, Appendix B).

We run both the agents using GPT-5.2 (Singh et al., 2025; OpenAI, 2025) and GPT-5-mini (Singh et al., 2025) on `CR-Bench-verified` version of the dataset, and compare their performance as per metrics defined in section 5. For verification's LLM-as-a-judge, we use Claude-Sonnet-4.5 (Anthropic, 2025) using the prompt detailed in listing 6 in Appendix C.

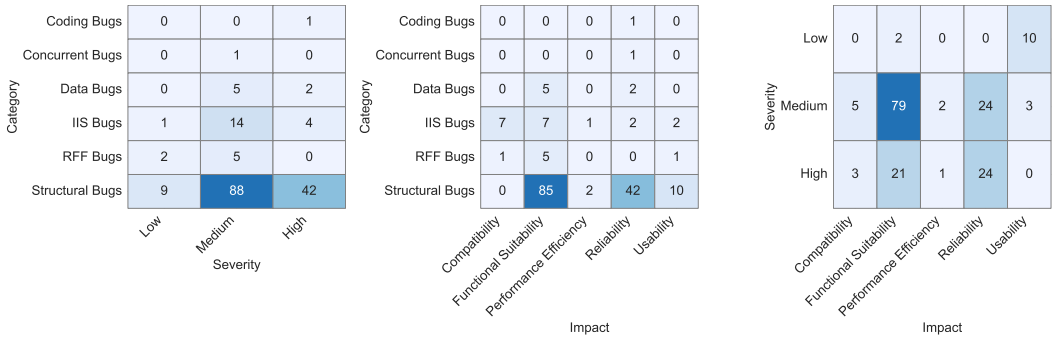
7 RESULTS

7.1 CR-BENCH STATISTICS

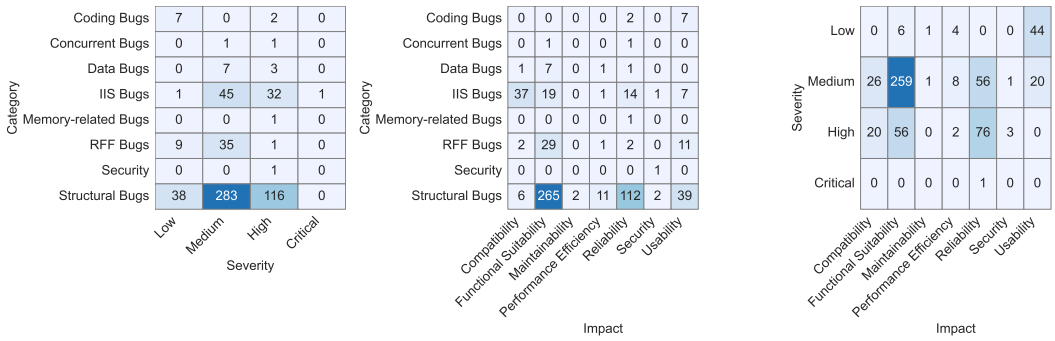
Dataset	# Lines fixed	# PR Comments	PR Description Length
`CR-Bench`	10.28	41.03	906.63
`CR-Bench-verified`	8.69	35.83	893.59

Table 3: Average statistics per instance for `CR-Bench` and its verified subset. All values represent means calculated across the respective dataset instances.

The final corpus `CR-Bench-verified` and `CR-Bench` of consists of 174 and 584 high-fidelity PR tasks. The dataset is anchored by major frameworks like django/django, mathematical libraries like sympy/sympy, and specialized tools such as astropy/astropy and scikit-learn/scikit-learn. This shows that these reviews are rooted in mature, long-lived codebases. To summarize the quantitative



(a) Distribution of category, severity, and impact for the `CR-Bench-verified` dataset ($N = 174$).



(b) Distribution for the full `CR-Bench` dataset ($N = 584$), showing increased taxonomic diversity.

Figure 3: Comparative analysis of bug category, severity, and impact distributions between the verified subset and the full `CR-Bench` corpus (Note: RFF = Requirements, Features and Functionality, IIS = Interface, Integration and System).

characteristics of the `CR-Bench` and `CR-Bench-verified` datasets, table 3 aggregates the average metrics related to patch complexity, pull request (PR) discussion volume, and context length. These statistics underscore the enterprise-scale nature of the benchmark, highlighting the significant surface area agents must navigate to identify defects.

As per Figure 3a, `CR-Bench-verified` is heavily weighted toward Structural Bugs, accounting for 79.9% of the corpus. These represent complex errors in code arrangement or logic flow that often elude simple static analysis. The primary impacts are Functional Suitability and Reliability. The rigor of `CR-Bench-verified` is further evidenced by its severity, where 93.1% of the defects are classified as Medium or High severity. High-severity defects are predominantly linked to Reliability and Functional Suitability, representing critical failures that would likely result in system regressions if merged. Thus, while 174 instances may seem less compared to other datasets in Table 1, we focus more on quality and stressed evaluation of logical boundaries. While these are quality enforced, there are some categories and impact missing in this subset.

The expansion into the full `CR-Bench` further reinforces these findings while increasing the statistical power of the benchmark. As shown in Figure 3b, this larger set maintains the high-stakes nature of the evaluation, with 90.2% of defects classified as Medium, High, or Critical severity. We see instances of impacts like Maintainability, Security existing in this version of the dataset, that were missing before. We also see some minor readjustments to proportions of different categories, impacts, and bugs. This ensures that the benchmark provides a meaningful assessment of an AI agent's ability to safeguard production-grade software.

7.2 Performance Analysis

Comparison Across Agents. The comparative performance of the Single-shot and Reflexion agents illustrates a clear trade-off between discovery coverage and the density of useful feedback. The

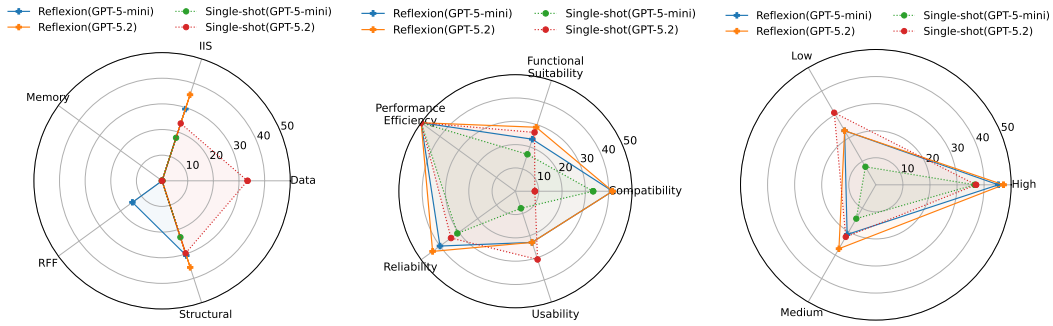
Agent	Model	Recall	Prec.	F1	Usefulness	SNR
Single-shot	GPT-5.2	27.01%	3.56%	6.30%	**83.63%**	**5.11**
	GPT-5-mini	18.39%	3.51%	5.90%	74.29%	2.89
Reflexion	GPT-5.2	**32.76%**	**5.10%**	**8.83%**	66.10%	1.95
	GPT-5-mini	27.59%	3.19%	5.72%	47.72%	0.91

Table 4: Comparative performance of code review agents on `CR-Bench-verified` on two techniques with two models. Best results are marked in **BOLD**.

Single-shot approach demonstrates superior signal integrity, particularly with GPT-5.2. With an SNR of 5.11, it can maintain high developer trust by minimizing the frequency of false alarms. However, this precision comes at the cost of Recall (27.01%), as the single-pass nature of the agent often overlooks subtle, cross-file errors. Conversely, the Reflexion agents instructively searches for false negatives, which boosted GPT-5.2's Recall to 32.76% (an increase from 27.01%). This technique is most preferable for security-critical audits or high-risk refactoring where catching the needle in the haystack is prioritized over time efficiency. But this comes with a drop of 1.95 in SNR. Moreover, Single-shot agents have lower precision, but higher usefulness compared to Reflexion agents. It could be because of the deep architecture of Reflexion, focusing on bug discovery.

Effect of Model Scale. The results also highlight a significant divergence in how model scale affects signal stability. GPT-5.2 possesses the semantic robustness to maintain an SNR of 1.95 even under the pressure of reflexion. In contrast, the SNR of GPT-5-mini agent drops to 0.91, indicating a possible struggle with logical grounding during reflexion. While GPT-5-mini achieves a respectable SNR of 2.89 in a single pass, its SNR collapses to 0.91 with the reflexion agent. This suggests a Reflexion agent with smaller LLMs has possibly higher hallucination. Due to iterative probing to identify more issues, the small model possibly suffers from a bias of not being able to disagree that more issues don't exist, and responds with noise.

7.3 Recall Distribution Across Taxonomy Components



(a) Recall by Category (b) Recall by Impact (c) Recall by Severity

Figure 4: PR Bug Recall analysis across Category, Impact, and Severity for `CR-Bench-verified` (Note: RFF = Requirements, Features and Functionality, IIS = Interface, Integration and System).

Assuming the false negative rate is a leading performance indicator for code review agents, we conducted deep analysis on the recall distribution.

Recall by Root Cause Category. The distribution of recall across root cause category suggests that all four agents are effective at identifying Structural and IIS (Interface, Integration, and System) defects. Recall of memory issues is zero for all agents, which feels correct as these are determined generally from system traces during execution. The Reflexion framework provides its most significant gains in the Structural and Data categories. For GPT-5.2, the iterative reflexion process allows it to re-read complex logic flows, successfully uncovering deeper algorithmic errors that were missed in the initial heuristic pass, as evidenced by the expanded radar profile in Figure 4a. Conversely,

GPT-5-mini exhibits a severe performance ceiling in RFF, IIS, and Structural bugs, but with reflexion. The performance degrades with single-shot. A surprising behavior is for Data bugs, where only the performance of single-shot GPT 5.2 is acceptable.

Recall by Possible Downstream Impact. When mapped to ISO/IEC 25010 impact standards, a clear pattern emerges regarding the type of quality an AI agent can protect. All configurations show peak performance in identifying defects with Performance efficiency and Reliability impacts. These bugs often have distinct code signatures such as inefficient loops or missing error handlers, that align well with the pattern-matching strengths of frontier LLMs like GPT-5.2. However, there is a dip in catching Usability and Functional suitability issues. These defects often require external environmental knowledge or upstream dependency context that is not fully contained within the PR diff, highlighting a fundamental limitation in current closed-context code review agents, lacking access to the broader system state. On compatibility issues, all agents except single-shot GPT 5.2 perform well.

Recall by Operational Severity. On operational severity metrics, all agents demonstrate significantly higher recall for High-Severity defects compared to Low or Medium ones. The GPT-5.2 Reflexion agent achieves its highest discovery rate in the High-severity bucket, suggesting the need for reasoning depth for severe bugs. Conversely, Low-Severity defects show the lowest recall across all paradigms. This indicates that while agents are effective safety nets for major regressions, they are less reliable at identifying minor nit-pick logic flaws, which developers might find acceptable.

8 Conclusion

In this work, we introduced `CR-Bench`, a benechmark dataset designed to bridge the gap between synthetic benchmarks and the complexities of real-world automated code review. By transforming high-fidelity software failures from SWE-Bench into `CR-Bench`, we provided a rigorous, defect-identifying PR review corpus of 174 (verified) and 584 (standard) defects that challenges AI agents to perform blind audits without prior defect knowledge. Our development of `CR-Evaluator` further enables a nuanced assessment of agent performance, moving beyond binary traditional metrics like precision, recall, accuracy, to more practical metrics from a developers perspective like usefulness rate and signal-to-noise ratio (SNR).

Our systematic evaluation of code review agents uncovered a fundamental design trade-off. While the Reflexion paradigm significantly enhances discovery Recall, it simultaneously incurs a steep cost in signal integrity with high SNR. This effect is most pronounced in lightweight models like GPT-5-mini. Overall, `CR-Bench` and `CR-Evaluator` jointly lay the design foundation of real-world code review agent evaluation.

9 Future Work

The findings from our evaluation of `CR-Bench` provide a clear roadmap for the next generation of automated code review agents. In this work, we considered four LLM-based agents. In future work, we plan to evaluate a broader range of agents, encompassing additional LLMs and diverse agentic architectures. We will also explore the applicability of advanced post-training techniques, such as GRPO. Finally, we aim to broaden the scope of `CR-Bench` by incorporating additional programming languages and domain-specific frameworks. This expansion will allow us to investigate how language-specific constraints influence an agent's signal-to-noise ratio (SNR) and recall profiles.

REFERENCES

Anthropic. Claude code: Github actions documentation, 2024. URL https://code.claude.com/docs/en/github-actions. Accessed: 2024-05-20.

Anthropic. Claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, 2025. Large language model.

Anysphere. Cursor bugbot: Automated bug finding and fixing, 2024. URL https://cursor.com/bugbot. Accessed: 2024-05-20.

Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 712–721, 2013. doi: 10.1109/ICSE.2013.6606617.

B. Beizer. *Software System Testing and Quality Assurance*. Electrical-Computer Science and Engineering Series. Van Nostrand Reinhold, 1984. ISBN 9780442213060. URL https://books.google.com/books?id=zNAmAAAAMAAJ.

CodeRabbit. Coderabbit: Ai-powered code reviews, 2024. URL https://www.coderabbit.ai/. Accessed: 2024-05-20.

Google Cloud. Gemini code assist: Ai-powered assistance for software development, 2024. URL https://codeassist.google/. Accessed: 2024-05-20.

International Organization for Standardization and International Electrotechnical Commission. Systems and software engineering — systems and software quality requirements and evaluation (square) — product quality model, 2023. URL https://www.iso.org/standard/78176.html.

Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics, 2019. URL https://arxiv.org/abs/1906.01388.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Codereviewer: Pre-training for automating code review activities. *arXiv preprint arXiv:2203.09095*, 2022.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: NLG evaluation using gpt-4 with better human alignment. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2511–2522, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.153. URL https://aclanthology.org/2023.emnlp-main.153/.

OpenAI. Gpt-5.2. https://openai.com/index/introducing-gpt-5-2/, 2025. Large language model.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Pierre Isabelle, Eugene Charniak, and Dekang Lin (eds.), *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pp. 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040/.

Qodo AI. Pr-agent: Ai-powered tool for automated pull request analysis, feedback, and suggestions, 2024. URL https://github.com/qodo-ai/pr-agent. Accessed: 2024-05-20.

Oussama Ben Sghaier, Martin Weyssow, and Houari Sahraoui. Harnessing large language models for curated code reviews, 2025. URL https://arxiv.org/abs/2502.03425.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, Akshay Nathan, Alan Luo, Alec Helyar, Aleksander Madry, Aleksandr Efremov, Aleksandra Spyra, Alex Baker-Whitcomb, Alex Beutel, Alex Karpenko, Alex Makelov, Alex Neitz, Alex Wei, Alexandra Barr, Alexandre Kirchmeyer, Alexey Ivanov, Alexi Christakis, Alistair Gillespie, Allison Tam, Ally Bennett, Alvin Wan, Alyssa Huang, Amy McDonald Sandjideh, Amy Yang, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrei Gheorghe, Andres Garcia Garcia, Andrew Braunstein, Andrew Liu, Andrew Schmidt, Andrey Mereskin, Andrey Mishchenko, Andy Applebaum, Andy Rogerson, Ann Rajan, Annie Wei, Anoop Kotha, Anubha Srivastava, Anushree Agrawal, Arun Vijayvergiya, Ashley Tyra, Ashvin Nair, Avi Nayak, Ben Eggers, Bessie Ji, Beth Hoover, Bill Chen, Blair Chen, Boaz Barak, Borys Minaiev, Botao Hao, Bowen Baker, Brad Lightcap, Brandon McKinzie, Brandon Wang, Brendan Quinn, Brian Fioca, Brian Hsu, Brian Yang, Brian Yu, Brian Zhang, Brittany Brenner, Callie Riggins Zetino, Cameron Raymond, Camillo Lugaresi, Carolina Paz, Cary Hudson, Cedric Whitney, Chak Li, Charles Chen, Charlotte Cole, Chelsea Voss, Chen Ding, Chen Shen, Chengdu Huang, Chris Colby, Chris Hallacy, Chris Koch, Chris Lu, Christina Kaplan, Christina Kim, CJ Minott-Henriques, Cliff Frey, Cody Yu, Coley Czarnecki, Colin Reid, Colin Wei, Cory Decareaux, Cristina Scheau, Cyril Zhang, Cyrus Forbes, Da Tang, Dakota Goldberg, Dan Roberts, Dana Palmie, Daniel Kappler, Daniel Levine, Daniel Wright, Dave Leo, David Lin, David Robinson, Declan Grabb, Derek Chen, Derek Lim, Derek Salama, Dibya Bhattacharjee, Dimitris Tsipras, Dinghua Li, Dingli Yu, DJ Strouse, Drew Williams, Dylan Hunn, Ed Bayes, Edwin Arbus, Ekin Akyurek, Elaine Ya Le, Elana Widmann, Eli Yani, Elizabeth Proehl, Enis Sert, Enoch Cheung, Eri Schwartz, Eric Han, Eric Jiang, Eric Mitchell, Eric Sigler, Eric Wallace, Erik Ritter, Erin Kavanaugh, Evan Mays, Evgenii Nikishin, Fangyuan Li, Felipe Petroski Such, Filipe de Avila Belbute Peres, Filippo Raso, Florent Bekerman, Foivos Tsimpourlas, Fotis Chantzis, Francis Song, Francis Zhang, Gaby Raila, Garrett McGrath, Gary Briggs, Gary Yang, Giambattista Parascandolo, Gildas Chabot, Grace Kim, Grace Zhao, Gregory Valiant, Guillaume Leclerc, Hadi Salman, Hanson Wang, Hao Sheng, Haoming Jiang, Haoyu Wang, Haozhun Jin, Harshit Sikchi, Heather Schmidt, Henry Aspegren, Honglin Chen, Huida Qiu, Hunter Lightman, Ian Covert, Ian Kivlichan, Ian Silber, Ian Sohl, Ibrahim Hammoud, Ignasi Clavera, Ikai Lan, Ilge Akkaya, Ilya Kostrikov, Irina Kofman, Isak Etinger, Ishaan Singal, Jackie Hehir, Jacob Huh, Jacqueline Pan, Jake Wilczynski, Jakub Pachocki, James Lee, James Quinn, Jamie Kiros, Janvi Kalra, Jasmyn Samaroo, Jason Wang, Jason Wolfe, Jay Chen, Jay Wang, Jean Harb, Jeffrey Han, Jeffrey Wang, Jennifer Zhao, Jeremy Chen, Jerene Yang, Jerry Tworek, Jesse Chand, Jessica Landon, Jessica Liang, Ji Lin, Jiancheng Liu, Jianfeng Wang, Jie Tang, Jihan Yin, Joanne Jang, Joel Morris, Joey Flynn, Johannes Ferstad, Johannes Heidecke, John Fishbein, John Hallman, Jonah Grant, Jonathan Chien, Jonathan Gordon, Jongsoo Park, Jordan Liss, Jos Kraaijeveld, Joseph Guay, Joseph Mo, Josh Lawson, Josh McGrath, Joshua Vendrow, Joy Jiao, Julian Lee, Julie Steele, Julie Wang, Junhua Mao, Kai Chen, Kai Hayashi, Kai Xiao, Kamyar Salahi, Kan Wu, Karan Sekhri, Karan Sharma, Karan Singhal, Karen Li, Kenny Nguyen, Keren Gu-Lemberg, Kevin King, Kevin Liu, Kevin Stone, Kevin Yu, Kristen Ying, Kristian Georgiev, Kristie Lim, Kushal Tirumala, Kyle Miller, Lama Ahmad, Larry Lv, Laura Clare, Laurance Fauconnet, Lauren Itow, Lauren Yang, Laurentia Romaniuk, Leah Anise, Lee Byron, Leher Pathak, Leon Maksin, Leyan Lo, Leyton Ho, Li Jing, Liang Wu, Liang Xiong, Lien Mamitsuka, Lin Yang, Lindsay McCallum, Lindsey Held, Liz Bourgeois, Logan Engstrom, Lorenz Kuhn, Louis Feuvrier, Lu Zhang, Lucas Switzer, Lukas Kondraciuk, Lukasz Kaiser, Manas Joglekar, Mandeep Singh, Mandip Shah, Manuka Stratta, Marcus Williams, Mark Chen, Mark Sun, Marselus Cayton, Martin Li, Marvin Zhang, Marwan Aljubeh, Matt Nichols, Matthew Haines, Max Schwarzer, Mayank Gupta, Meghan Shah, Melody Huang, Meng Dong, Mengqing Wang, Mia Glaese, Micah Carroll, Michael Lampe, Michael Malek, Michael Sharman, Michael Zhang, Michele Wang, Michelle Pokrass, Mihai Florian, Mikhail Pavlov, Miles Wang, Ming Chen, Mingxuan Wang, Minnia Feng, Mo Bavarian, Molly Lin, Moose Abdool, Mostafa Rohaninejad, Nacho Soto, Natalie Staudacher, Natan LaFontaine, Nathan Marwell, Nelson Liu, Nick Preston, Nick Turley, Nicklas Ansman, Nicole Blades, Nikil Pancha, Nikita Mikhaylin, Niko Felix, Nikunj Handa, Nishant Rai, Nitish Keskar, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Oona Gleeson, Pamela Mishkin, Patryk Lesiewicz, Paul Baltescu, Pavel Belov, Peter Zhokhov, Philip Pronin,

Phillip Guo, Phoebe Thacker, Qi Liu, Qiming Yuan, Qinghua Liu, Rachel Dias, Rachel Puckett, Rahul Arora, Ravi Teja Mullapudi, Raz Gaon, Reah Miyara, Rennie Song, Rishabh Aggarwal, RJ Marsan, Robel Yemiru, Robert Xiong, Rohan Kshirsagar, Rohan Nuttall, Roman Tsiupa, Ronen Eldan, Rose Wang, Roshan James, Roy Ziv, Rui Shu, Ruslan Nigmatullin, Saachi Jain, Saam Talaie, Sam Altman, Sam Arnesen, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Sarah Yoo, Savannah Heon, Scott Ethersmith, Sean Grove, Sean Taylor, Sebastien Bubeck, Sever Banesiu, Shaokyi Amdo, Shengjia Zhao, Sherwin Wu, Shibani Santurkar, Shiyu Zhao, Shraman Ray Chaudhuri, Shreyas Krishnaswamy, Shuaiqi, Xia, Shuyang Cheng, Shyamal Anadkat, Simón Posada Fishman, Simon Tobin, Siyuan Fu, Somay Jain, Song Mei, Sonya Egoian, Spencer Kim, Spug Golden, SQ Mah, Steph Lin, Stephen Imm, Steve Sharpe, Steve Yadlowsky, Sulman Choudhry, Sungwon Eum, Suvansh Sanjeev, Tabarak Khan, Tal Stramer, Tao Wang, Tao Xin, Tarun Gogineni, Taya Christianson, Ted Sanders, Tejal Patwardhan, Thomas Degry, Thomas Shadwell, Tianfu Fu, Tianshi Gao, Timur Garipov, Tina Sriskandarajah, Toki Sherbakov, Tomer Kaftan, Tomo Hiratsuka, Tongzhou Wang, Tony Song, Tony Zhao, Troy Peterson, Val Kharitonov, Victoria Chernova, Vineet Kosaraju, Vishal Kuo, Vitchyr Pong, Vivek Verma, Vlad Petrov, Wanning Jiang, Weixing Zhang, Wenda Zhou, Wenlei Xie, Wenting Zhan, Wes McCabe, Will DePue, Will Ellsworth, Wulfie Bain, Wyatt Thompson, Xiangning Chen, Xiangyu Qi, Xin Xiang, Xinwei Shi, Yann Dubois, Yaodong Yu, Yara Khakbaz, Yifan Wu, Yilei Qian, Yin Tat Lee, Yinbo Chen, Yizhen Zhang, Yizhong Xiong, Yonglong Tian, Young Cha, Yu Bai, Yu Yang, Yuan Yuan, Yuanzhi Li, Yufeng Zhang, Yuguang Yang, Yujia Jin, Yun Jiang, Yunyun Wang, Yushi Wang, Yutian Liu, Zach Stubenvoll, Zehao Dou, Zheng Wu, and Zhigang Wang. Openai gpt-5 system card, 2025. URL https://arxiv.org/abs/2601.03267.

Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. CodeAgent: Autonomous communicative agents for code review. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11313, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.632. URL https://aclanthology.org/2024.emnlp-main.632/.

Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: automated code transformation to support modern code review process. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pp. 237–248, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510067. URL https://doi.org/10.1145/3510003.3510067.

Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, pp. 163–174. IEEE Press, 2021. ISBN 9781450390859. doi: 10.1109/ICSE43902.2021.00027. URL https://doi.org/10.1109/ICSE43902.2021.00027.

Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pp. 2291–2302, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510621. URL https://doi.org/10.1145/3510003.3510621.

Zezhou Yang, Cuiyun Gao, Zhaoqiang Guo, Zhenhao Li, Kui Liu, Xin Xia, and Yuming Zhou. A survey on modern code review: Progresses, challenges and opportunities, 2024. URL https://arxiv.org/abs/2405.18216.

Zhengran Zeng, Ruikai Shi, Keke Han, Yixin Li, Kaicheng Sun, Yidong Wang, Zhuohao Yu, Rui Xie, Wei Ye, and Shikun Zhang. Benchmarking and studying the llm-based code review, 2025a. URL https://arxiv.org/abs/2509.01494.

Zhengran Zeng, Ruikai Shi, Keke Han, Yixin Li, Kaicheng Sun, Yidong Wang, Zhuohao Yu, Rui Xie, Wei Ye, and Shikun Zhang. Benchmarking and studying the llm-based code review, 2025b. URL https://arxiv.org/abs/2509.01494.

A APPENDIX: CR-BENCH PROMPTS

```
system = """You are a software engineering expert analyzing code changes.
Your task is to determine if a change is of the kind that might have been
     prevented by reviewing the PR/commit that made that change more
    carefully or NOT.
Name these PREVENTABLE and UNPREVENTABLE.
PREVENTABLE: Code changes that fix incorrect behavior, errors, crashes,
    or unexpected results introduced by a previous code change (PR/commit
    ).
UNPREVENTABLE: Everything else including but not limited to: New features
    , refactoring, documentation, performance improvements, or
    enhancements.

Respond with JSON only:
{ "category": "PREVENTABLE" or "UNPREVENTABLE", "confidence": 0.0 to 1.0,
     "reasoning": "brief explanation" }"""

user = """Problem Statement:
{problem_statement}

Patch:
{patch}

Is this a bug fix that might have been introduced by a previous code
    change (PR/commit)?"""
```

Listing 1: Prompt for classifying a SWE Bench bug as detectable/not detectable.

```
system="""You are a software engineering expert writing technical bug
    descriptions.

Given a problem statement and the code fix patch, write a concise
    technical description of the bug.

**Format Requirements:**
- Start with a clear statement of what's wrong
- Include specific code snippets showing the problematic pattern
- Explain why it's a bug (runtime impact, incorrect behavior)
- Keep it under 100 words
- Use technical language
- Focus on the root cause, not symptoms

Respond with ONLY the bug description (no JSON, no extra formatting)."""

user="""Bug Description Task

## Problem Statement
{row['problem_statement']}

## Code Fix
File: {file_path}
Lines: {line_start}-{line_end}

```diff
{patch}
```

Write a concise technical description of this bug."""
```

Listing 2: Prompt for paraphrasing the issue description.

```
system = """You are a software engineering expert analyzing bug reports
    to classify them according to a predefined bug taxonomy used in
    empirical software engineering research.
```

```
Analyze the bug from the issue description (problem statement) and the
    code fix patch, and provide a classification using the following
    fields.

---

### 1. **category**

Choose EXACTLY ONE of the following bug types:

#### **1. Requirements, Features, and Functionality Bugs**

**Source:** Ambiguous, incomplete, misunderstood, or changing
    requirements/specifications.
**Types:**

* **Requirements bugs:** Missing, ambiguous, or inconsistent requirements
    .
* **Feature bugs:** Wrong, missing, or extra features; undesired
    enhancements.
  **Detection/Remedy:** Formal specification languages, functional
      testing (transaction flow, domain, logic, state-based testing).

#### **2. Structural Bugs**

**Source:** Code control flow, sequence, and logical structure.
**Types:**

* **Control & sequence bugs:** Missing paths, unreachable code, improper
    loop nesting, misused GOTO/labels.
* **Logic bugs:** Incorrect logical expressions; Boolean errors.
* **Processing bugs:** Arithmetic, algorithm, type conversion, overflow
    errors.
* **Initialization bugs:** Uninitialized variables or improper first-loop
     values.
* **Data-flow anomalies:** Using uninitialized or stale data; not storing
    /modifying data correctly.
  **Detection/Remedy:** Structural/path testing, unit testing, domain
      testing.

#### **3. Data Bugs**

**Source:** Specification, format, initial values, and usage of data
    objects.
**Types:**

* **Dynamic data bugs:** Temporary data issues, shared memory residues.
* **Static data bugs:** Fixed content/parameters; preprocessing errors.
* **Role-based bugs:** Errors in data used as control, parameter, or
    information.
* **Attributes, structure, content bugs:** Incorrect semantics, memory
    allocation, or representation.
  **Detection/Remedy:** Validate all data declarations, centralize shared
       resources, defensive coding, compile-time checks.

#### **4. Coding Bugs**

**Source:** Programming errors, including wild-card or arbitrary mistakes
    .
**Types:**

* **Syntax errors:** Usually caught by compilers.
* **Documentation bugs:** Misleading comments or manuals.
```

```
  **Detection/Remedy:** Proper code review, accurate documentation,
      automated syntax checking.

#### **5. Interface, Integration, and System Bugs**

**Source:** Interactions between components, systems, or hardware.
**Types:**

* **External interface bugs:** Wrong device or protocol handling.
* **Internal interface bugs:** Improper module communication.
* **Hardware bugs:** Misunderstood device behavior, I/O errors.
* **OS bugs:** Misuse or assumption of OS services.
* **Software architecture bugs:** Emergent behavior due to module
    interactions or stress.
* **Integration bugs:** Incompatibilities between integrated modules.
* **System bugs:** Complex interaction of multiple components, rare but
    expensive.
  **Detection/Remedy:** Design for modularity, stress testing,
      integration testing, specialist interfaces.

#### **6. Test and Test Design Bugs**

**Source:** Errors in testing processes or test code.
**Types:**

* **Test execution bugs:** Incorrect test implementation.
* **Test design bugs:** Flawed test scenarios or criteria.
  **Detection/Remedy:** Test debugging, automation of test execution and
      design, test quality assurance.

#### **7. Memory-related Bugs**

**Source:** Improper memory handling.
**Types:**

* **Buffer overflow:** Access beyond allocated memory.
* **Stack smashing:** Overwriting function return addresses.
* **Memory leak:** Lost pointers to allocated memory.
* **Uninitialized read:** Accessing memory before initialization.
* **Double free:** Freeing memory twice.
  **Detection/Remedy:** Careful memory management, automated memory
      checking tools.

#### **8. Concurrent Bugs**

**Source:** Multi-threading or multi-process synchronization issues.
**Types:**

* **Data race bugs:** Conflicting access to shared memory.
* **Atomicity bugs:** Interrupted sequences of operations.
* **Deadlock:** Processes waiting indefinitely for shared resources.
  **Detection/Remedy:** Proper synchronization, lock ordering, deadlock
      detection mechanisms.

---

### 2. **severity**

Classify the runtime impact of the bug:

* **Critical:** System crash, severe malfunction, data loss, or serious
    security risk.
* **High:** Core functionality broken or system unusable in common
    scenarios.
```

```
* **Medium:** Partial malfunction, degraded behavior, or non-critical
    failures.
* **Low:** Minor issues, cosmetic problems, edge cases, or test-only
    failures.

---

### 3. **impact**

Choose EXACTLY ONE aspect of the software that the bug directly
    compromises:

#### **1. Functional Suitability**

* **Description:** Bugs that cause incorrect, missing, or inappropriate
    behavior relative to requirements.
* **Examples:** Wrong outputs, missing features, incorrect calculations.

#### **2. Performance Efficiency**

* **Description:** Bugs that degrade system performance or resource
    utilization.
* **Examples:** Slow response, high CPU/memory usage, poor throughput.

#### **3. Compatibility**

* **Description:** Bugs that interfere with interoperability or co-
    existence with other systems, devices, or software.
* **Examples:** Integration failures, protocol mismatches, API
    incompatibilities.

#### **4. Usability**

* **Description:** Bugs that make the system confusing, hard to use, or
    increase user errors.
* **Examples:** Misleading error messages, poor workflow, inaccessible UI
     elements.

#### **5. Reliability**

* **Description:** Bugs that cause system instability, crashes, incorrect
     state, or inability to recover from failures.
* **Examples:** Crashes, deadlocks, memory leaks, data corruption.

#### **6. Security**

* **Description:** Bugs that expose the system to threats, unauthorized
    access, or data compromise.
* **Examples:** Buffer overflows, improper access control, injection
    vulnerabilities.

#### **7. Maintainability**

* **Description:** Bugs that make code harder to understand, modify, or
    extend.
* **Examples:** Poorly structured code, missing documentation, complex
    interdependencies.

#### **8. Portability**

* **Description:** Bugs that prevent software from being installed,
    executed, or transferred across different environments.
* **Examples:** OS-specific assumptions, hardware incompatibilities,
    environment-specific failures.
```

```
---

### 4. **reasoning**

Concisely explain the reasoning behind the **category**, **impact**, and
    **severity** of the bug.

---

### Output format

Respond ONLY with valid JSON in the following format:

'''json
{
  "category": "...",
  "severity": "...",
  "impact": "...",
  "reasoning": "..."
}
'''

Do not include explanations, markdown, or any additional text outside the
     JSON."""

user = """# Bug Classification Task

## Problem Statement
{problem_statement}

## Code Fix
File: {file_path}
Lines: {line_start}-{line_end}

'''diff
{correct_fix}
'''

Analyze this bug and provide classification as JSON."""
```

Listing 3: Prompt for cateogorizing PR and bugs into category, impact, severity.

B Appendix: Code Review Agent Prompts

```
SYSTEM_PROMPT = """You are an expert code reviewer analyzing pull
    requests for potential bugs and issues.

Your task is to review the provided code changes and identify:
1. Potential bugs or errors
2. Logic issues
3. Edge cases not handled
4. Security vulnerabilities
5. Performance problems

For each issue found, provide:
- file: The file path
- line: The line number (approximate)
- comment: Clear description of the issue
- severity: "high", "medium", or "low"

Output your review as a JSON array of comments. Example:
[
 {
```

```
   "file": "src/module.py",
   "line": 42,
   "comment": "Potential null pointer dereference. Variable 'user' may be
       None.",
   "severity": "high"
 }
]

If no issues found, return an empty array: []

Focus on actual bugs, not style or formatting issues. Do not be verbose,
    just provide the review comments in concise manner that descirbes the
     issue completely."""


USER_PROMPT_TEMPLATE = Template("""# Pull Request Review

**Repository:** {{ repo }}
**PR Number:** #{{ pr_number }}
**Title:** {{ title }}
**Description:** {{ description }}

## Diff
```diff
{{ diff }}
```

Please review the above changes and identify any potential bugs or issues
    .""")
```

Listing 4: Prompt for Single-shot agent.

```
SYSTEM_PROMPT = """You are an expert code reviewer specializing in bug
    detection and security analysis.

Review the provided code changes and identify ONLY actual bugs and issues
    , NOT style or formatting concerns.

## Focus Areas:
1. **Bugs**: Logic errors, incorrect implementations, type mismatches
2. **Edge Cases**: Unhandled null/undefined, empty arrays, boundary
    conditions
3. **Security**: SQL injection, XSS, authentication bypasses, data leaks
4. **Concurrency**: Race conditions, deadlocks, thread safety issues
5. **Performance**: Memory leaks, infinite loops, inefficient algorithms

## Output Format:
Return a JSON array. Each issue must include:
- file: Exact file path from the diff
- line: Line number where issue occurs
- comment: Specific, actionable description of the bug (not generic
    warnings)
- severity: "high" (crashes/security), "medium" (data corruption/
    incorrect behavior), "low" (minor issues)

## Quality Standards:
- Be SPECIFIC: Point to exact problematic code, not general concerns
- Be CERTAIN: Only report issues you're confident are real bugs
- Explain WHY: Include the consequence of the bug
- Suggest FIX: Briefly mention how to resolve it

Return [] if no bugs found.

Example:
[
```

```
  {
    "file": "src/auth.py",
    "line": 23,
    "comment": "Missing null check before accessing 'user.email'. Will
        throw AttributeError if user is None. Add: if user is None:
        return error",
    "severity": "high"
  }
]"""


USER_PROMPT_TEMPLATE = Template("""# Pull Request to Review

**Repository:** {{ repo }}
**PR #{{ pr_number }}:** {{ title }}

{{ description }}

## Code Changes
```diff
{{ diff }}
```

Analyze the diff above and identify concrete bugs. Focus on what could
    actually break or cause incorrect behavior.""")

REFLECTION_SYSTEM_PROMPT = """You are performing iterative self-
    improvement using the Reflexion framework.

Your goal: Produce a MORE COMPLETE and ACCURATE review by finding bugs
    you missed.

Reflexion Strategy:

PRIMARY FOCUS: Find Missed Bugs (False Negatives)
Your previous review likely MISSED important bugs. Search for:
- Null/undefined dereferences you overlooked
- Logic errors in conditionals or loops
- Off-by-one errors, incorrect operators
- Unhandled exceptions or error cases
- Security vulnerabilities (injection, auth bypass)
- Resource leaks, race conditions, memory issues

SECONDARY: Remove Only OBVIOUS False Positives

When in doubt, KEEP the issue. Err on the side of caution.

TERTIARY: Improve Existing Issues
- Make vague comments more specific
- Adjust severity if clearly wrong
- Add fix suggestions

Output Requirements:
- Return the FULL JSON array with ALL valid issues
- Prioritize ADDING missed bugs over removing uncertain ones
- Each comment must point to specific problematic code
- Be thorough - it's better to report a potential bug than miss a real
    one

Return [] only if you're absolutely certain no bugs exist."""
```

```
REFLECTION_USER_PROMPT_TEMPLATE = Template("""# Reflexion - Iteration {{
    iteration }}: Find What You Missed

**Repository:** {{ repo }} | **PR:** #{{ pr_number }}

## Code to Review
```diff
{{ diff }}
```

Your Previous Review (Iteration {{ prev_iteration }})
You found {{ num_comments }} issue(s):
```json
{{ previous_comments }}
```

Reflexion Task: FIND MISSED BUGS

PRIMARY: What Bugs Did You Miss?

Carefully re-examine the diff. Look for bugs you

Output Your Complete Refined Review

Return a JSON array with:
1. All valid issues from previous review (unless clearly false positive)
2. NEW bugs you discovered in this reflection
3. Improved comments and severity levels

Format:
```json
[
  {
    "file": "path/to/file.py",
    "line": 42,
    "comment": "Specific bug: what breaks, why, and how to fix",
    "severity": "high|medium|low"
  }
]
```

Return [] only if absolutely no bugs exist.""")
```

Listing 5: Prompt for Reflexion agent.

C Appendix: CR-Evaluator Prompts

```
"""You are evaluating code review comments against a known bug.

## Known Bug Description:
{bug_description}

## Files Changed in Fix:
{patch_files}

## Review Comment to Evaluate:
File: {file}
Line: {line}
Comment: {comment}
```

```
## Task:
Classify this review comment into ONE of these categories:

1. **BUG_HIT**: The comment identifies or relates to the same issue
    described in the bug description
2. **VALID_SUGGESTION**: The comment makes a valid point (style,
    performance, maintainability, edge case, etc.) but is NOT about the
    known bug
3. **NOISE**: The comment is incorrect, irrelevant, or not actionable

Respond with ONLY a JSON object:
{{"classification": "BUG_HIT|VALID_SUGGESTION|NOISE", "reason": "brief
    explanation"}}
"""
```

Listing 6: Prompt for CR-Evaluator.