Adversarial Review: Cooperative Code Review through Structured Disagreement

Anonymous Authors[1]

Abstract

Early multi-agent LLM systems often used role-separated teams that debate or aggregate independent outputs, yet scaling agent count yields diminishing returns on repository-level coding tasks. In comparison, recent work has shifted towards a main-agent-plus-subagents paradigm, where subagents act like tools rather than independent collaborators. We study whether this paradigm can support a middle ground: minimal agentic cooperation without the overhead of large multi-agent teams. We introduce Adversarial Review (AR), a cooperative code-review protocol in which a main agent works with a reviewer and a critic. The reviewer evaluates code, while the critic audits the review through structured disagreement before the main agent edits or commits. On LiveCodeBench, AR achieves the highest pass rate among tested methods, outperforming a six-agent baseline. On SWE-PRBench, naive AR exposes a false-consensus failure mode, where agents converge on agreement without sufficient evidence, but with textual constraints it achieves the highest F1 among tested methods. On SWE-bench Verified, AR also show improvements over the baselines on repository-level coding tasks. Overall, AR demonstrates that cooperative oversight is useful when disagreement is minimal, structured, and evidence-grounded.

1. Introduction

Early multi-agent LLM systems often framed agents as role-separated teams of largely independent peers, mirroring human divisions of labor by assigning agents to generate, review, debate, or aggregate outputs (Islam et al., 2024; Huang et al., 2023; Pan et al., 2025; Islam et al., 2025). However, recent work suggests that scaling the number of peer agents can yield diminishing, and sometimes negative, returns on complex repository-level coding tasks, partly because unconstrained communication introduces coordination overhead and failure modes of its own (Smit et al., 2023; Kim et al., 2025). In contrast, many production-grade agent systems have converged on a *main agent with tools* paradigm, where invoking a subagent is treated as another tool call rather than as collaboration with an independent peer (Anthropic, 2024).

We ask whether there is a productive middle ground between these two paradigms: can a main-agent-plus-subagents system support lightweight cooperation among independent agents without incurring the overhead of large role-separated teams? Our goal is to design a minimal protocol that preserves the benefits of agentic disagreement while constraining communication enough to remain useful on real coding tasks.

We introduce *Adversarial Review* (AR), a cooperative coding protocol built step by step from naive zero-shot code generation. Each step adds one design choice motivated by a measurable failure of the previous step. We first evaluate this construction on LiveCodeBench (LCB) (Jain et al., 2024). We then probe the resulting protocol in two directions: review quality on SWE-PRBench (Kumar, 2026) and repository-level repair on SWE-bench Verified (Jimenez et al., 2024). We use two execution modes. On LCB and SWE-PRBench, a Python orchestrator enforces matched-budget comparisons across methods. On SWE-bench Verified, we express each method as a portable pure-text `SKILL.md` protocol followed by autonomous agents such as Claude Code.

Our contribution is a constructive study of minimal agentic cooperation in real-world coding tasks. First, we show that a reviewer–critic loop can improve coding agents within the main-agent-plus-subagents paradigm. Second, we identify a core Cooperative AI failure mode: agents can optimize for agreement rather than truth. Third, we show that structured disagreement can remain useful even when implemented as a portable text protocol for autonomous agent systems.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

055
056
057
058
059
060
061
062
063
064
065
066
067
068
069
070
071
072
073
074
075
076
077
078
079
080
081
082
083
084
085
086
087
088
089
090
091
092
093
094
095
096
097
098
099
100
101
102
103
104
105
106
107
108
109

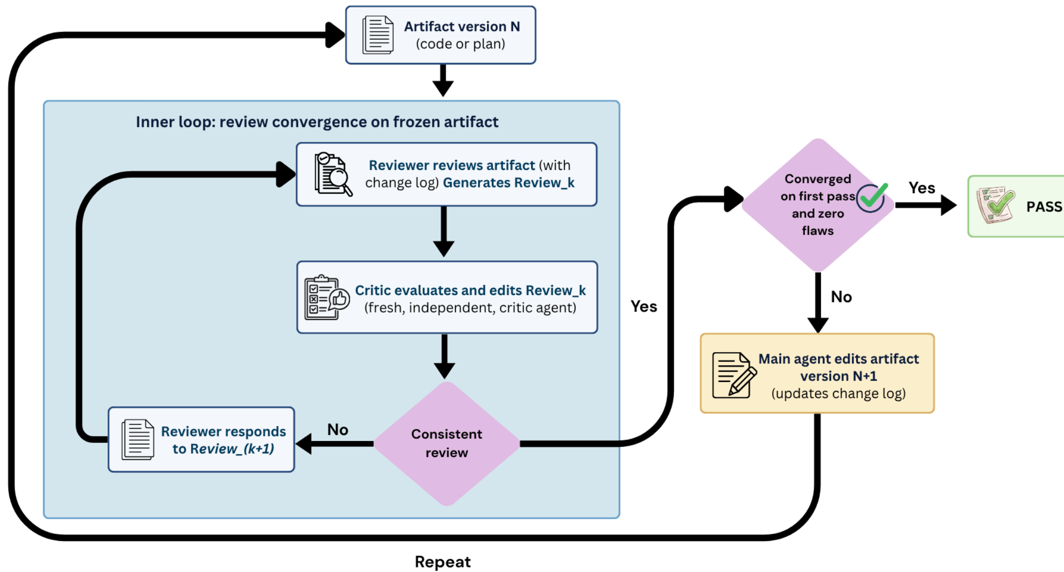


Figure 1. Workflow of Adversarial Review (AR). The main agent first produces artifact version N (code or plan). The protocol then enters an inner loop in which the artifact is frozen: reviewer R generates Review_k, critic C evaluates and may revise that review, and R responds until a consistent review is reached. If the review converges on the first pass and identifies no flaws, the artifact is accepted immediately. Otherwise, the main agent edits the artifact to produce version $N+1$, updates the change log, and the process repeats. The key separation is that the inner loop exchanges review text only, while artifact edits occur only in the outer loop.

2. Related Work

Self-refinement and external verification. Test-time refinement methods improve an initial model output by adding critique, feedback, or revision. Self-Refine uses the same model as generator, feedback provider, and refiner, keeping revision within a single model's judgment loop (Madaan et al., 2023). However, subsequent work shows that language models do not reliably self-correct reasoning errors without external feedback (Huang et al., 2024). AR builds on test-time refinement but makes two changes: it separates the reviewer and critic roles, and it makes disagreement an explicit part of the protocol rather than an optional self-check.

Multi-agent debate and structured disagreement. Multi-agent debate uses interaction among model instances to improve reasoning, factuality, or decision quality. Prior systems allow agents to exchange arguments before a judge or final answer is selected (Du et al., 2024; Liang et al., 2024). However, debate-style systems can be sensitive to prompt and protocol choices, and may fail to outperform cheaper non-debate baselines (Smit et al., 2023). MARS reduces this overhead by replacing round-table debate with independent reviewers whose comments are aggregated by a meta-reviewer (Wang et al., 2025). AR takes a different middle position. It preserves interaction, but restricts it to a single reviewer–critic channel. The goal is not open-ended debate, but evidence-grounded pushback before the main agent edits or commits.

Multi-agent systems for code generation and review. Several coding systems use role-separated agents to divide programming into specialized subtasks. MapCoder decomposes competitive-programming tasks into example recall, planning, code generation, and debugging (Islam et al., 2024). AgentCoder separates implementation, test design, and test execution (Huang et al., 2023). CodeSim uses simulation-driven planning and debugging to guide code generation (Islam et al., 2025). CodeCoR uses a self-reflective multi-agent framework to prune, refine, and repair generated code (Pan et al., 2025). For code review, CodeAgent studies communicative agents for automated review and adds a QA-Checker to keep generated review contributions aligned with the original question (Zhang et al., 2024). AR is intentionally smaller than these systems. Rather than adding many specialized coding roles, it asks whether a minimal reviewer–critic pair can provide useful oversight within a main-agent-plus-subagents workflow.

Agentic coding systems and portable protocols. Production agent systems increasingly rely on a main agent that calls tools, subagents, or reusable instruction bundles. Anthropic describes agentic systems as systems that reason, use tools, and adapt their process over multiple steps (Anthropic, 2024). In this setting, subagents are separate agent instances spawned for focused subtasks, while Skills are portable instruction bundles activated through a `SKILL.md` file (Anthropic, 2026; 2025). This motivates evaluating AR in two forms. The Python orchestrator isolates the protocol

under matched-budget comparisons. The pure-text SKILL form tests whether the same cooperative-review protocol can be followed by an autonomous coding agent without an external controller enforcing each step.

Benchmarks for coding and review. Coding-agent evaluation now spans algorithmic programming, repository-level repair, and code review. LiveCodeBench evaluates code generation using a contamination-resistant benchmark that is updated over time (Jain et al., 2024). SWE-PRBench evaluates AI code-review quality against pull-request feedback (Kumar, 2026). SWE-bench evaluates whether agents can resolve real GitHub issues by editing repositories and passing tests (Jimenez et al., 2024). Together, these benchmarks separate three claims that are often conflated: whether a method solves standalone programming problems, whether it helps with repository-level repair, and whether it produces useful review comments. We use this separation to evaluate AR as both a coding protocol and a cooperative-oversight protocol.

3. Constructive design progression on LiveCodeBench

This section builds the Adversarial Review protocol step by step. Each subsection adds one design choice. Each result reports two metrics on LCB: pass-rate over all 105 stdin-style tasks (pass/105) and pass-rate over the 57 hard-tagged subset (pass-on-hard/57). Full results are shown in Table 1. The hard tier is where the methods separate. We use Claude Sonnet 4.5 Medium Reasoning for all subsequent agent and subagent calls for all benchmarks.

3.1. Zero-shot → Self-Refine: adding verification

Zero-shot writes code with no verification step. So we add one. Self-Refine (Madaan et al., 2023) has the same agent self-critique its own code, then revise. On LCB, the improvements are barely noticeable. Zero-shot pass-rate is 77%. Self-Refine pass-rate is also 77%. Pass-on-hard also stays at 35/57. The reason is that the critic and the generator are the same model making the same mistakes. So the critic struggles to catch what the generator missed (Olausson et al., 2024). This motivates using a separate model call for the critic.

3.2. Self-Refine → Single-reviewer: external verification

Single-reviewer employs separate model call. Specifically, M writes the code, R reviews M's code, and M edits once. On LCB, pass-rate is 77%. Pass-on-hard is 36/57. This is a small change from Self-Refine and stays in the same cluster as Zero-shot. One issue may be that a single external reviewer has high variance. It can approve everything without checking, or flag many small nits to look thorough without targeting the real issue. This motivates running multiple independent reviewers.

3.3. Single-reviewer → Two-reviewers: independence reduces variance

Two independent samples should reduce variance. Two-reviewers-in-series has R_1 and R_2 review M's code independently, then M edits once based on the union. On LCB, pass-rate is 75%. Pass-on-hard is 34/57. This is within noise of Single-reviewer, Self-Refine, and Zero-shot. So independence alone gives little improvement. We read this as a four-method cluster at the same level. The four methods are Zero-shot, Self-Refine, Single-reviewer, and Two-reviewers. Naive verification cannot push past this level. This motivates pushing for diversity rather than more independent samples.

3.4. Two-reviewers → MARS: diversity, but with a known limit

Multi-agent debate (MAD) is the standard way to add diversity in multi-agent LLM systems (Du et al., 2024). However, MAD has two costs that grow with the number of agents and rounds. The first cost is communication overhead, because each agent must read every other agent's messages. The second cost is that pass-rate can even drop as the system scales (Smit et al., 2023; Kim et al., 2025).

MARS (Wang et al., 2025) reduces MAD-style communication costs by removing message passing between reviewers. Rather, reviewers work independently, each providing a decision, review comments, and a confidence score. A meta-aggregator combines their reviews at the end.

On LCB, MARS pass-rate is 82% and pass-on-hard is 39/57. So MARS is the first method whose pass-rate is clearly higher than Zero-shot, Self-Refine, Single-reviewer, and Two-reviewers. MARS gains +5pp on pass/105 over the four-method cluster and +3 hard tasks over Single-reviewer. But MARS has two issues. First, the gain is small relative to the token cost from using 5 agents per task (4 personas plus a meta-aggregator). Second, while MARS avoids MAD's communication overhead, all agent interactions are removed completely. This motivates a natural question: ***can we improve coding performance through agent interaction, yet without paying MAD's overhead, and with fewer agents?***

3.5. MARS → AR: interaction with two agents

The smallest interactive structure that still permits meaningful cooperation is one reviewer plus one critic of that reviewer. We call this **Adversarial Review (AR)**. As shown in Figure 1, the protocol operates over successive artifact versions produced by a main agent M (artifact is either code

Table 1. LCB results. The leader is bold. The first four methods cluster at the same pass-rate. MARS breaks out of the cluster. AR scores highest with the fewest agents among the methods that broke out.

Method	pass / 105	pass-on-hard / 57	# agents
Zero-shot	77%	35/57 (61%)	1
Self-Refine	77%	35/57 (61%)	1
Single-reviewer	77%	36/57 (63%)	2
Two-reviewers	75%	34/57 (60%)	3
MARS	82%	39/57 (68%)	6
AR	**87%**	**43/57 (75%)**	**3**

or a plan). At outer iteration N, M produces an artifact version N, together with a change log that records how the artifact evolved.

AR then enters an *inner loop* in which the artifact is never edited. Instead, an independent reviewer subagent R inspects the frozen artifact and generates a review Review_k. A fresh critic subagent C then evaluates that review and may revise or challenge it. If the review and critique are not yet consistent, the protocol returns to R, who responds with a revised review Review_{k+1}. This reviewer–critic exchange repeats until the review converges (or, for cost control, until a cap of 5 inner rounds is reached). Thus, the inner loop is a review procedure over a fixed artifact, not an editing procedure over the artifact itself. This means that we can evaluate the inner loop separately as a code-review method (see Section 4).

Once a consistent review is reached, the protocol exits the inner loop and applies an outer-loop decision rule. If R and C converged immediately and found no flaws, AR terminates and the artifact is accepted. This is the *first-pass termination rule*. Otherwise, the consistent review is returned to the main agent M, which edits the artifact to produce version $N{+}1$, updates the change log, and launches a new inner round of review on the updated artifact. In this way, AR alternates between (i) an inner loop that seeks consensus over review text of the artifact and (ii) an outer loop in which the main agent edits the artifact only after a stable review signal has been produced.

On LCB, AR has the highest pass-rate of all six methods we test (pass-rate is 87%. Pass-on-hard is 43/57). In particular, AR scores +5pp over MARS while using fewer agents: 2 reviewing roles versus MARS's 5.

Table 1 summarises all six methods. Evaluation on LCB first validates the middle ground we seek. AR shows that in the main-agent-plus-subagent paradigm, **two interacting agents can perform better than five non-interacting personas**.

3.6. The minimal interactive structure

AR has the smallest number of roles that still has interaction. AR has three roles: M, R, and C. R and C interact. This matches the main-plus-subagent shape used by production agent systems like Claude Code, Cursor, and Copilot (Anthropic, 2024). If we remove C, the loop becomes Single-reviewer. If we add more reviewers or more rounds, the loop becomes MARS-style or MAD-style.

AR has the highest LCB pass-rate, but we further probe the protocol in two directions. Section 4 looks at review quality of the inner loop on SWE-PRBench. Section 5 looks at real agentic scenarios on SWE-bench Verified.

4. Inner Loop review quality on SWE-PRBench

This section does two things. First, it shows that AR's LCB win does not transfer naively to review-quality benchmarks. Second, it finds the failure modes and fixes them with one prompt iteration. The fix points to a cooperative-AI lesson that applies beyond AR.

4.1. Experiment Setup

SWE-PRBench (Kumar, 2026) contains 100 real GitHub PR diffs. The agent's job is to review the diff. A GPT-5.2 judge matches the agent's comments against the actual human reviewer's comments. The judge agrees with human annotators at Cohen's kappa = 0.75. Cohen's kappa is a standard agreement score, where 1.0 is perfect agreement and 0 is chance agreement. The metric is F1 over matched comments.

We restrict the comparison to four methods: AR, MARS, Two-reviewers, and Single-reviewer. We exclude Zero-shot and Self-Refine, because they do not apply to a review-only benchmark. Zero-shot has no review step. Self-Refine has the agent critique its own work, but SWE-PRBench gives the agent a third party's PR, so there is no "own work" to refine. Evaluating the four methods answers whether AR's interactive R–C structure produces better reviews than methods that run reviewers in parallel.

4.2. AR underperforms naively

The naive AR protocol from Section 3.5 gives F1 = 0.457 on SWE-PRBench. This is the lowest of the four methods in the subset. The other three subset methods all reach F1 around 0.50. So review quality is a setting where AR's interactive R–C loop gives a lower F1 than methods without it. The rest of Section 4 finds out why and fixes it.

Table 2 shows the leaderboard. Two case studies in Section 4.3 surface two failure modes. Section 4.4 describes

Table 2. SWE-PRBench review-relevant subset. The leader is bold. v1 and v2 are prompt-level iterations of the same R+C structure. The number of agents and the way they connect did not change between v1 and v2. The prompt iteration is described in Section 4.4.

Method	F1	N
AR (v2)	**0.533**	100
Two-reviewers	0.503	100
MARS	0.501	100
Single-reviewer	0.495	100
AR (v1)	0.457	100

the prompt iteration that fixes both. The fixed protocol is labelled v2 and reaches F1 = 0.533, the highest in the subset.

4.3. Two failure modes (case studies)

We show two case studies. Each is one task from SWE-PRBench. We use verbatim trace excerpts.

Case A — Over-decomposition. v1 R writes a review with one real bug and two to three hedged concerns. v1 C agrees with all of R's flags and adds another speculative bug on top. A format-review step turns each flag into a separate comment. The judge marks 3 of 5 comments as fabricated, because the comments are too thin or too speculative. The mechanism is that the R–C loop adds findings without filtering them. R hedges often. C confirms most of R's hedges. So the result is too many thin comments, which the judge marks as fabricated. On this task AR (v1) has F1 = 0.250.

Case B — C yields to R. v1 R produces an APPROVE review. v1 C raises a real concern, which we verified manually. v1 R rebuts C with weak signals; R uses a file-level argument yet cites no code. v1 C gives in and the verdict flips to AGREE. So the real bug is dropped from the final review. Case B reveals a structural mechanism: the protocol pushes R and C to agree, so C tends to agree. R can end the disagreement by writing a confident-sounding rebuttal, even when R is wrong. What looks like "structured disagreement" becomes false agreement. On this task, AR (v1) has F1 = 0.286, while MARS catches the same bug at F1 = 0.667.

The two cases share one cause. In Case A, R adds many hedged findings, and C confirms them all. In Case B, R rebuts a real concern from C with confident-sounding text, and C drops a real concern right away. In both cases, a recurring theme stands out: *two cooperating LLM agents asked to agree on a joint output tend to agree with each other. They do not always find the truth.* This means that a collaborative agentic protocol must encode truth forcefully—any agent pushback must be evidence-grounded. Without grounded pushback, R can add weak findings and C will not stop them, and C can agree on R's rebuttal even when the rebuttal cites no code.

4.4. AR (v2)

We explicitly guide pushback constraints in text, producing AR (v2). Everything else stay the same. AR (v2) is still one R, one C, with inner and outer loops.

The textual constraint of AR (v2) specifies that all flags should be grounded in concrete evidence. Specifically, C used to choose between two verdicts (`AGREE` or `DISAGREE`). Now C must choose one of three:

- `AGREE` means C accepts R's review.
- `DISAGREE_EVIDENCE: <code citation>` means C cites specific code that contradicts a flag.
- `DISAGREE_CONCERN: <epistemic objection>` means C raises an objection that cannot point to contradicting code.

R's response rule depends on which kind of disagreement it sees. On `AGREE`, R keeps every flag unchanged. On `DISAGREE_EVIDENCE`, R revises the flag based on the cited code. On `DISAGREE_CONCERN`, R must cite code in the diff that confirms the bug (and keep the flag) or cite code that refutes the bug (and drop the flag).

4.5. Result

AR (v2) F1 = 0.533. This is the highest among the four-method subset. Recall is 0.596 (v1: 0.520). Precision is 0.640 (v1: 0.505). Since AR (v2) already achieves the highest F1 in the SWE-PRBench, we stop iterating intentionally to maintain the minimal protocol structure we motivated to design: one R, one C, with only changes in the prompts.

The case studies show that *when two LLM agents are asked to agree on a joint output, they tend to agree with each other. They do not always find the truth.* So the protocol must include explicit pushback. Without pushback, what looks like "structured disagreement" becomes false agreement.

5. Real-world agentic performance on SWE-bench Verified

5.1. Experimental Setup

SWE-bench Verified (Chowdhury et al., 2024) contains 500 real GitHub issues in open-source Python repos. The set is human-filtered for solvability. The agent must read the issue, navigate the repo, find the bug, and emit a patch. The metric is pass@1, measured through the official Docker harness.

The execution mode here is different from Section 3 and Section 4. We use Claude Code with full tool access. Claude Code loads the `adversarial-review` skill, which is

Table 3. SWE-bench Verified pass-rate of three methods: Zero-shot, MARS, and AR.

Method	pass-rate (%)	N
AR	**75.2%**	500
Zero-shot	71.6%	500
MARS	72.6%	500

simply a markdown document (SKILL.md) that records the AR protocol in pure text (rather than Python orchestration logic).

Evaluation of the skills form of AR has several desirable characteristics. First, the skills form, by design, let a main LLM agent drive the protocol itself. Specifically, the agent can decide when to invoke R and C as Task subagents, and it also decides when to edit. While our evaluation harness of AR for LCB and SWE-PRBench intentionally used Python orchestrating logic for fair comparisons in token cost, this leans more towards an agentic workflow, where the structure is rigid. On the other hand, a more general notion of LLM agent can decide when to invoke and how to structure its own workflow. Coding agents, which often need flexibility and innovative decision-making to solve complex, repository level coding tasks, fall under the latter notion of agents. Hence, the skills form of AR provides a natural way to evaluate the protocol on flexible coding agents. Additionally, the loose constraint of text (rather than code) intentionally puts the cooperative-AI claim to a strong test, because an external orchestrator does not force cooperation on the agent, we observe how an agent may organize cooperation through its given instruction solely.

We evaluate only three methods on SWE-bench Verified because full benchmark runs for each method estimate over 300 hours. We choose Zero-shot as the internal baseline, MARS as the strongest prior multi-agent baseline from LCB, and AR as our proposed method.

5.2. Result

As shown in Table 3, AR reaches 75.2% pass-rate on N=500. Zero-shot on the same Claude Code scaffold without the AR skill reaches 71.6%. MARS reaches 72.6%. Though notably, AR uses about 4.5× the tokens of Zero-shot (see Section 6), trading off token-efficiency for performance gains.

5.3. Case studies

We use two case studies to further analyze AR's results on SWE-bench Verified. The first case shows when AR helps: the reviewer–critic loop pushes the agent toward the root cause. The second case shows when AR hurts: the same loop amplifies a speculative concern and expands the patch beyond the issue scope. Both cases come from AR's skills-form evaluation.

Case 1 — Structured disagreement helps find the root cause. *Task:* `matplotlib__matplotlib-20826`. The issue occurs after `Axis.clear()` resets the per-tick keyword argument dictionaries. As a result, when a user calls `ax.clear()` on shared or subplot axes, tick-visibility settings such as labels hidden by `label_outer()` are lost.

Zero-shot and MARS both patch the caller: `lib/matplotlib/axes/_base.py`. Their patches re-apply tick visibility settings after the reset has already destroyed them. This fixes the symptom, but not the underlying cause. Zero-shot produces a 45-line patch, and MARS produces a 50-line patch. Both fail the hidden tests.

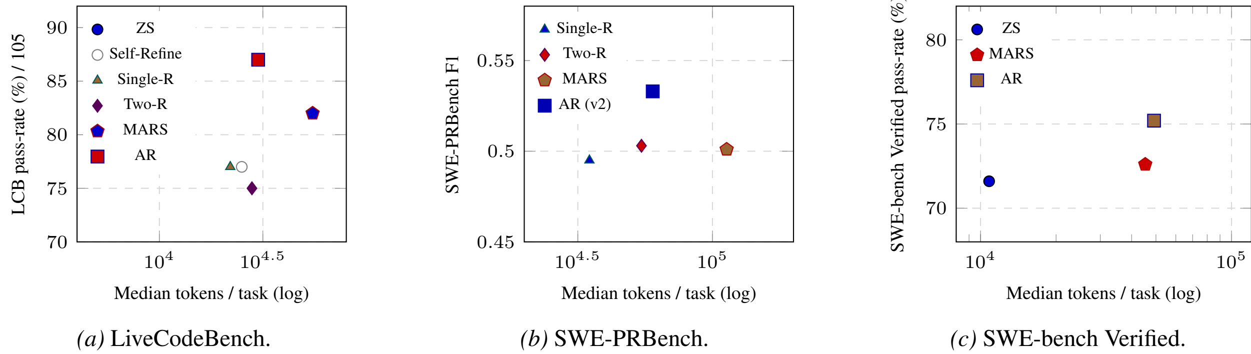
In comparison, AR instead patches the callee: `lib/matplotlib/axis.py`. Rather than re-applying visibility settings after the reset, AR removes the over-broad reset and preserves the relevant tick keyword state directly:

```
- self._reset_major_tick_kw()
- self._reset_minor_tick_kw()
+ self._major_tick_kw['gridOn'] = ...
+ self._minor_tick_kw['gridOn'] = ...
```

This difference matters. Zero-shot and MARS repair the visible symptom at the public-API layer. While AR's reviewer subagent R makes the same mistake in the initial round of reviews, the critic subagent C catches the mistake right away, and the agents converge to agreement immediately. As a result, AR actually repairs the state-destroying operation at the implementation layer. AR's patch is shorter, about 20 lines, and is the only patch that passes the hidden tests. We interpret this as a case where the reviewer–critic loop helps the main agent inspect one frame deeper in the call chain, moving from symptom repair to root-cause repair.

Case 2 — Structured disagreement amplifies scope creep. *Task:* `astropy__astropy-14182`. The issue asks the agent to add a `header_rows` argument to the RST writer. Zero-shot solves the task with a minimal 24-line patch that threads the parameter through `__init__` and `write`. The patch stays within the issue scope and passes the hidden tests.

AR produces a larger 32-line patch. It adds `header_rows`, but also removes a stable class variable, `start_line=3` in `SimpleRSTData`, and adds a new `read()` method. These extra changes were not required by the issue. The source of the over-edit is visible in the trace: during planning, the reviewer asks, "what if the user calls `read()` separately?" The critic does not reject this as out of scope. The main agent then implements the speculative change,



(a) LiveCodeBench. *(b)* SWE-PRBench. *(c)* SWE-bench Verified.

Figure 2. Per-task median tokens vs. quality on each benchmark. The two y-axes measure different things across panels (binary pass-rate vs. LLM-judge F1). The within-panel claim is the within-panel trade-off, not a cross-panel comparison. LCB token medians are estimated from per-method call counts; SWE-PRBench and SWE-bench Verified token medians are computed from per-task logs.

and the patch fails the hidden tests.

This case shows the main failure mode of AR on SWE-bench Verified. The reviewer–critic loop can improve root-cause analysis, but it can also turn plausible concerns into unnecessary edits. In other words, AR helps when disagreement redirects the agent toward the true bug, and hurts when disagreement expands the repair beyond the issue. This suggests that future versions of AR could make scope discipline as explicit as the evidence-grounded disagreement we encoded in AR (v2) (Section 4.4).

6. Cost-quality position across benchmarks

Though AR trades token cost for performance, as we see in Section 5.2, nevertheless AR is on the cost-quality Pareto frontier of all three benchmarks we evaluate. The Pareto frontier is the set of methods where no other method is both cheaper and better. Figure 2 plots per-task median tokens against performance on each benchmark.

The takeaway is that AR uses more tokens than Zero-shot on all three benchmarks. The extra tokens give AR Pareto-frontier performance on every panel. So no method we tested is both cheaper and better than AR.

7. Discussion and limitations

AR works because it makes cooperation narrow. The main result of this paper is not that adding more agents improves coding. In fact, our constructive progression shows the opposite: adding independent reviewers alone does not reliably improve LCB, and MARS requires five agents to obtain a smaller gain than AR. The useful part of AR is more specific. AR restricts cooperation to one narrow interface: a reviewer writes a review, and a critic audits that review. The artifact is frozen during this exchange. Edits occur only after the review stabilizes. This separation matters because it prevents the agents from jointly rewriting the solution while they are still disagreeing about what is wrong. In this sense, while AR is a small oversight mechanism inserted into a main-agent workflow, AR behaviorally resolves as a collaborative agent team with distinct role separation.

The results support a middle ground between tools and teams. Production coding agents often follow a main-agent-plus-tools pattern, where subagents are invoked as tools. Earlier multi-agent coding systems often follow a team pattern, where many role-separated agents generate, debate, or aggregate outputs. AR sits between these two designs. The main agent remains responsible for writing and editing the artifact, but the reviewer and critic are not merely passive tools. They interact with each other before the main agent acts. This design gives AR some benefits of role-separated cooperation without requiring a large round-table debate like MAD (Du et al., 2024; Smit et al., 2023; Wang et al., 2025). The SWE-bench Verified result is important for this reason: AR still improves performance when expressed only as a `SKILL.md` protocol, without a Python orchestrator forcing every step. This suggests that lightweight cooperative protocols can be portable across agentic systems when the roles and decision rules are simple enough to express in text.

Cooperation is not the same as trustworthiness. The SWE-PRBench results show why cooperative AI systems need to be audited, not merely benchmarked. AR v1 performs best on LCB but worst among the review-centered methods on SWE-PRBench. The case studies make the failure visible. In one case, the reviewer proposes too many weak findings and the critic confirms them. In another case, the critic raises a real concern, but then yields to a confident rebuttal from the reviewer. Both failures have the same structure: the agents reach agreement, but the agreement is not supported by enough evidence. We call this failure mode *false consensus*. False consensus is especially concerning because it can look like independent validation: two agents

appear to agree, but the agreement may only reflect conversational pressure to converge. Below we discuss how we could explicitly and transparently ground trustworthiness of collaborative agents in ways that may mitigate such false consensus.

Evidence-grounded disagreement is the key design principle. AR v2 improves SWE-PRBench performance by adding textual constraints rather than more agents. The critic must distinguish agreement, evidence-backed disagreement, and concern-based disagreement. The reviewer must then respond with code evidence instead of a plausible rebuttal. This turns disagreement into an auditable object. The trace shows whether a decision was supported by code, speculation, or premature agreement. This is the main trustworthy-AI lesson of AR: multi-agent oversight should not be judged only by whether agents converge. It should be judged by whether the path to convergence preserves dissent, evidence, and accountability. Our textual constraint forces an agent to discretize its disagreement into exactly one of three types, which makes agent disagreement clear-cut, explicit, and transparent. However, textual constraint may not be the best nor the final solution to enforcing semantic constraints in LLM agents. While the instruction-following capability of LLMs continue to grow along its general ability, current LLMs are still prone to forgetting or ignoring instructions, especially as the context window approaches full and past full. Further work on more powerful methods to guide LLMs is an orthogonal research direction to our work, yet nevertheless important to creating trustworthy AI.

Cost trade-off. AR is not a free improvement. It uses more tokens than Zero-shot on all three benchmarks. The cost-quality plots show that AR lies on the Pareto frontier among the methods we tested, but this does not mean AR should always be used. For easy tasks, the review loop may be wasted computation. For tasks where the issue scope is narrow, extra review may increase the chance of over-editing. AR is most justified when the failure cost is high, when the repository context is complex, or when root-cause localization is more important than minimizing token cost. This suggests a natural future direction: an adaptive version of AR that invokes the reviewer–critic loop only when the main agent is uncertain or when the patch touches high-risk code.

Our claims about agent cooperation are empirical, not formal. We do not prove equilibrium properties of AR. We also do not claim that reviewer–critic loops are universally better than independent reviewers, debate, or single-agent refinement. Our conclusions are based on three empirical probes: constructive progression on LCB, inner-loop review quality on SWE-PRBench, and skills-form repository repair on SWE-bench Verified. These settings are complementary, but they do not cover all coding tasks or all agent scaffolds. In particular, the Python-orchestrated experiments isolate the protocol under controlled budgets, while the SWE-bench Verified experiment tests a looser skills-form implementation. These two modes answer different questions. The first asks whether the mechanism works under controlled execution. The second asks whether the mechanism survives when delegated to an autonomous coding agent.

The evaluation is limited by benchmarks and judge design. LCB measures competitive-programming style correctness, SWE-PRBench measures similarity to human pull-request feedback, and SWE-bench Verified measures repository-level repair through hidden tests. Each benchmark captures only one view of quality. In particular, SWE-PRBench depends on an LLM judge that matches generated comments to human review comments. This makes it useful for comparing review methods, but it may penalize valid comments that differ from the human review or reward comments that match surface form without improving the patch. Similarly, SWE-bench Verified measures whether the final patch passes tests, but it does not fully measure maintainability, minimality, or long-term code quality. The case studies partly address this gap by examining mechanism, but larger and more comprehensive evaluations would be needed to validate AR as a deployed review assistant.

Future work should study adaptive and auditable cooperation. The next step of collaborative multi-agent systems may not simply be adding more agents or more rounds. The main open question is when cooperation should be invoked and how its traces should be audited. For our line of work, AR variants could add a scope-checking critic, a confidence gate that skips review on easy tasks, or a disagreement log that records which claims were supported by code evidence. More broadly, our results suggest that cooperative AI protocols should be designed around explicit failure modes. Agents should not merely be encouraged to agree. They should be required to preserve the evidence, uncertainty, and dissent that make agreement trustworthy.

8. Conclusion

We construct AR step by step on LCB. The progression goes Zero-shot, Self-Refine, Single-reviewer, Two-reviewers, MARS, AR. AR achieves the highest measured score among the methods we test on all three benchmark; yet AR is the smallest interactive structure in the main-plus-subagent paradigm. Rather than scaling agents or interaction rounds, we show through AR that improvement in collaborative agents may lie in carefully structured disagreement as well as explicit audits and guidance.

440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Anthropic. Building effective agents. https://www.anthropic.com/research/building-effective-agents, December 2024. Accessed: 2026-05-10.

Anthropic. Equipping agents for the real world with agent skills. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills, October 2025. Accessed: 2026-05-10.

Anthropic. Subagents in the SDK. https://code.claude.com/docs/en/agent-sdk/subagents, 2026. Accessed: 2026-05-10.

Chowdhury, N., Aung, J., Shern, C. J., Jaffe, O., Sherburn, D., Starace, G., Mays, E., Dias, R., Aljubeh, M., Glaese, M., Jimenez, C. E., Yang, J., Ho, L., Patwardhan, T., Liu, K., and Madry, A. Introducing SWE-bench verified. https://openai.com/index/introducing-swe-bench-verified/, August 2024. Updated February 24, 2025. Accessed: 2026-05-10.

Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., and Mordatch, I. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11733–11763. PMLR, 2024. URL https://proceedings.mlr.press/v235/du24e.html.

Huang, D., Zhang, J. M., Luck, M., Bu, Q., Qing, Y., and Cui, H. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation, 2023. URL https://arxiv.org/abs/2312.13010.

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., and Zhou, D. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IkmD3fKBPQ.

Islam, M. A., Ali, M. E., and Parvez, M. R. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4912–4944, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.269. URL https://aclanthology.org/2024.acl-long.269/.

Islam, M. A., Ali, M. E., and Parvez, M. R. CodeSim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 5128–5154, Albuquerque, New Mexico, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-naacl.285. URL https://aclanthology.org/2025.findings-naacl.285/.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. LiveCodeBench: Holistic and contamination-free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Kim, Y., Gu, K., Park, C., Park, C., Schmidgall, S., Heydari, A. A., Yan, Y., Zhang, Z., Zhuang, Y., Malhotra, M., Liang, P. P., Park, H. W., Yang, Y., Xu, X., Du, Y., Patel, S., Althoff, T., McDuff, D., and Liu, X. Towards a science of scaling agent systems, 2025. URL https://arxiv.org/abs/2512.08296.

Kumar, D. SWE-prbench: Benchmarking AI code review quality against pull request feedback, 2026. URL https://arxiv.org/abs/2603.26130.

Liang, T., He, Z., Jiao, W., Wang, X., Wang, Y., Wang, R., Yang, Y., Shi, S., and Tu, Z. Encouraging divergent thinking in large language models through multi-agent debate. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 17889–17904, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.992. URL https://aclanthology.org/2024.emnlp-main.992/.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Prasad Majumder, B., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Olausson, T. X., Inala, J. P., Wang, C., Gao, J., and Solar-Lezama, A. Is self-repair a silver bullet for code generation? In *International Conference on Learning Represen-*

tations, 2024. URL https://arxiv.org/abs/2306.09896.

Pan, R., Zhang, H., and Liu, C. CodeCoR: An LLM-based self-reflective multi-agent framework for code generation, 2025. URL https://arxiv.org/abs/2501.07811.

Smit, A., Duckworth, P., Grinsztajn, N., Barrett, T. D., and Pretorius, A. Should we be going MAD? a look at multi-agent debate strategies for LLMs, 2023. URL https://arxiv.org/abs/2311.17371.

Wang, X., Wang, J., Wang, Y., Dang, P., Cao, S., and Zhang, C. MARS: Toward more efficient multi-agent collaboration for LLM reasoning, 2025. URL https://arxiv.org/abs/2509.20502.

Zhang, K., Li, J., Li, G., Shi, X., and Jin, Z. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13643–13658, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.737. URL https://aclanthology.org/2024.acl-long.737/.