arXiv:2611.29162v1 [cs.SE] 22 Nov 2026

修复还是删除？外部反馈让低成本 AI 编码代理修复规则违规

Laura Wynter*新加坡管理大学计算与信息系统学院，新加坡；以及 CodeAssure Labs，新加坡

2026年11月

摘要

许多缺陷违反了项目规则，而这些违规无法仅从代码本身看出，例如分层约束或必需测试。我们研究大语言模型编码代理在外部检查器报告违规及其位置和修复目标后，能否更好地修复此类违规。由 AI 代理驱动的三个模型（小型、中型和大型模型）针对十个开源项目中的 25 个拉取请求（其中植入了 31 个缺陷）以及 5 个正确对照案例开展工作，每个案例分别独立运行五次，并在获得商业检查器 CodeAssure（CA）的发现结果后运行五次。我们还将结果与 GitHub Copilot 代码审查进行比较，并将 CodeAssure 的检测能力与九种开箱即用的免费静态分析工具进行比较，其中包括 Semgrep 和 SonarQube Community。只有当修复后的代码也能成功编译且保留其功能时，我们才将一次运行计为修复成功。CA 发现结果将最小模型的修复率从 26% 提高到 69%，将中型模型的修复率从 46% 提高到 90%，将最大模型的修复率从 45% 提高到 93%（拉取请求级别 $p < 0.001$）；总体而言，所有运行的修复率从 39% 提高到 84%。所有 Java、TypeScript 和 Python 修复均成功编译。我们发现，智能体经常通过删除新添加的文件来消除缺陷。根据缺陷是否消失来衡量的指标会把删除功能也算作成功。最大模型单独运行时，在 125 次运行中有 44 次如此操作。CA 包含一项检查已删除文件的检查器，大体上可以防止这种情况。在关闭该检查的消融实验中，375 次运行中的文件删除次数从 9 次升至 65 次，修复率则降至小型模型 54%、中型模型 70% 和大型模型 85%。CA 工作流的表现也优于 Copilot 代码审查（$p < 0.001$）。九种免费工具合计检测出 31 个缺陷中的 8 个，而 CA 检测出了全部 31 个。借助 CA，中型模型在 90% 的运行中修复成功；相比之下，最大模型单独运行时的修复率为 45%，而前者的平均单次运行成本约为后者的五分之一（0.17 美元，对比 0.91 美元）。

1 引言

基于大语言模型（LLM）的编码代理如今能够在很少监督的情况下执行多步骤软件任务。它们会读取代码仓库、编辑文件、运行命令并提交。SWE-bench 等基准[10] 用于衡量智能体能否解决问题，使项目的测试通过。然而，变更中的许多要求并未通过测试覆盖。成熟的项目都有架构规则（例如，哪些层可以

依赖哪些层）、约定（例如，服务代码中不得输出控制台信息，或存储层中不得使用原始线程）以及流程要求（例如，新命令必须附带相应测试）。这类规则往往只写在贡献者指南中，甚至完全没有文档记录；其随时间逐渐弱化的现象已在软件架构研究中得到充分探讨[3]。

若要求智能体审查自身的变更，它就必须从代码中推断这些规则。先前研究表明，缺乏外部信号的自我审查并不可靠。大语言模型往往无法纠正自身推理 [9], ，而代码生成中的自我修复效果取决于反馈质量 [20]。确定性工具可以提供这种信号。架构一致性检查由来已久 [19], ，近期研究则将静态分析结果与大语言模型修复相结合 [24]。

本文报告了一项受控研究，考察接收此类反馈的编码代理。具体而言，我们在十个广泛使用的开源项目的拉取请求（PR）中植入缺陷；这些项目使用 C++、Rust、Python、

*通讯作者：lwynter@smu.edu.sg

TypeScript 和 Java 编写。随后，我们使用 AI 编码代理中的三种不同规模的模型修复这些缺陷，并比较有无外部检查器发现结果的情况。我们将这一工作流与 GitHub Copilot 代码审查进行比较，并进一步将检查器的检测结果与开箱即用的免费静态分析工具的检测结果进行比较。我们提出了四个问题。

- **RQ1.** 外部架构反馈能否提高编码代理修复规则违规的比率？其效果是否取决于模型规模？
- **RQ2.** 智能体会如何失败？特别是，它们多么频繁地通过删除包含缺陷的功能来移除缺陷？删除检查能否防止这种情况？
- **研究问题3.** 外部反馈能否节约成本？
- **RQ4.** 广泛使用的免费静态分析工具在开箱即用的情况下，是否已经能够检测出这些缺陷？

在回答上述问题时，我们的贡献如下。

- 证据表明，采用定位式外部反馈的工作流（此处的反馈来自一个商业检查器），可将修复率提高43至48个百分点；当修复后的代码还必须能够编译时，仅使用静态评分器可提高47至58个百分点。
- 功能删除的测量：智能体删除拉取请求所添加文件的情况。这是一种常见策略，而简单的成功指标会将其计为修复。消融实验表明，CA 的删除检查在很大程度上避免了这种情况：关闭该检查后，375 次运行中的删除次数从 9 次升至 65 次，修复率则下降 8 至 21 个百分点；在三个模型中，有两个模型的这一降幅在拉取请求层面具有统计显著性。
- 有证据表明，单独工作的智能体在被要求审查不含缺陷的更改时，通常仍会编辑其代码。在 CA 工作流中，对于这些更改，由于没有报告任何可采取行动的发现，智能体没有进行代码编辑。
- 在相同拉取请求上，与默认配置下的 GitHub Copilot 代码审查进行比较。
- 将九种免费静态分析工具直接按默认设置运行后的检测结果进行比较（Semgrep、SonarQube Community，以及各语言的标准代码检查器和编译器）；这些工具合计检测出 31 个植入缺陷中的 8 个。
- 一项成本分析表明，在获得外部反馈并进行删除检查的情况下，中型模型修复的运行次数是单独工作的最大模型的两倍，而其每次运行的平均成本约为后者的五分之一。
- 对 851 次评分修复进行的构建与功能检查表明，当代码必须通过编译时，这些优势依然成立：在有 CA 发现结果的情况下，84% 的运行会生成能够构建并保留其功能的修复，而智能体单独运行时这一比例为 39%。所有 Java、TypeScript 和 Python 修复均能通过编译；在 CA 工作流中，没有任何 Java 或 TypeScript 修复以任何方式失败。

第 2 节回顾相关工作，第 3 节介绍方法。第 4 节呈现结果，包括对 CA 的删除检查进行的消融实验。第 5 节讨论结果中的主要观察。第6 节将 CA 工作流与 GitHub Copilot 代码审查进行比较，第 7 节比较免费静态分析工具的检测结果。第 8 节讨论有效性威胁，第 9 节作出结论。

2 相关工作

智能体基准测试。 SWE-bench [10] 及其人工验证子集 [21] 通过运行项目测试来评估问题解决能力。SWE-agent 等智能体脚手架 [28] 和 Agentless [27] 提升了这些基准测试上的表现。我们的设置有两点不同：缺陷是测试通常无法检测到的规则违规；我们还会衡量智能体在不包含缺陷的更改上的行为。定位更改必须涉及的文件是智能体工作的重要部分，而智能体探索代码仓库的方式会影响其完成这项工作的效果 [5]。定位反馈会直接向智能体提供这些信息。

自我纠错与反馈。 当反馈具有信息量时，自我改进可以提升输出质量[17, 22], ，但没有外部信号的内在自我纠错往往会失败，或导致结果变差 [9]。对于代码，Olausson等人 [20] 发现，修复效果受反馈质量限制。汇总多条推理轨迹是提高可靠性的另一种途径，还能保留可能被多数投票舍弃的正确少数观点 [26]。我们的 Alone 工作流是一种内在自我审查。我们的 +CA 工作流则加入了外部定位反馈，否则模型就必须自行推断这些信息。

静态分析 wit　　大语言模型修复。 Wadhwa等人将静态分析警告与大语言模型配对，提出　　r

并使用大语言模型排序器解决代码质量问题。Blyth 等人 [1] 将代码检查器和安全扫描器的警告反馈给大语言模型，以改进生成的代码，使其超越功能正确性；CodeCureAgent [11] 使用智能体对静态分析警告进行分类和修复。Su 等人 [23] 发现，大语言模型能够很好地检测代码中可推断出的明确架构决策违规，但对隐含的架构决策违规检测能力较弱。从反射模型 [19] 到在持续集成中运行的规则库，架构一致性检查能够检测已声明模块依赖关系中的违规。我们研究当此类发现结果通过工具接口传递给自主智能体，并任由该智能体据此采取行动时会发生什么。

向智能体提供项目知识。 项目规则可以通过多种方式传达给智能体。可以通过微调将规则融入模型 [14], ，但规则一旦发生变化，保持模型更新的成本就很高。也可以将规则写成自然语言指令或技能文件，供智能体在运行时读取。Gloaguen 等人 [8] 发现，AGENTS.md 等代码库上下文文件通常并不能提高任务成功率，反而会使推理成本增加 20% 以上，不过智能体确实会遵循其中的指令。这类声明式技能在相关知识能够被可靠检索时，可能优于手工编码的工作流 [16]。在本研究中，规则由智能体之外的机制检查，并将违规情况报告给智能体。

规范博弈。 优化器严格按照字面要求实现目标，却偏离其本意，这种现象已有充分记录 [12]。观察发现，编码代理会删除或修改失败测试，或硬编码预期输出，而不是解决根本问题 [6, 29]。为了消除发现项而删除某项功能，也是一个例子。检查器的目标（不存在违规）实现了，但开发者的意图（功能正常运行且不存在违规）却没有实现。在强化学习中，可以从学习策略外部施加约束，而不是将其交由策略自行处理，例如将其与随机规划结合 [7]。外部检查对编码代理也发挥着类似作用，而保留功能必须成为一项约束。因此，CA 会检查功能是否被移除（第 3.3节）。

3 方法

3.1 拉取请求与植入缺陷

我们使用了十个广泛使用的开源项目的分支。按语言划分，11 个拉取请求使用 C++，9 个使用 Rust，6 个使用 Python，2 个使用 TypeScript，另有 2 个使用 Java。每个拉取请求都添加了一个小型且合理的功能，编写风格与项目一致。我们在其中 25 个拉取请求中植入了缺陷，并保留了 5 个正确的拉取请求。表格 1 总结了缺陷类别。五个多缺陷拉取请求共包含 11 个缺陷，分别位于不同的文件中，因此单次编辑不会意外修复其中两个缺陷。二十个拉取请求来自同一作者早先开展的一项单次运行研究，另有十个是为本研究编写的。

表格 1：按缺陷类别划分的拉取请求。

类别	描述	示例	PRs
代码可见	仅凭代码即可检测	重命名函数，但调用方仍留在原处	3
约定	关于项目特定规则 代码可以在何处执行何种操作	背后 服务代码中的控制台输出，原始存储层中的线程	8
架构规则	已声明的分层或流程 约束	数据层模块导入 Web 路由，缺少必需参数的命令 test	9
多个	两个或三个缺陷，来自不同类别		5
干净	无缺陷		5

对于每个缺陷，我们都在创建拉取请求之前编写了评分器规范。其缺陷谓词仅测试植入缺陷是否存在，并且我们确认它能在拉取请求的头部提交中发现该缺陷，而不会在其基准提交中发现。第 3.4 节中的运行结果为拉取请求新增的文件添加了规则，而这些文件在基准提交中并不存在。

3.2 智能体与模型

所有运行均使用 Claude Code（版本 2.1.285）作为非交互模式下的 AI 编码代理，整个过程中没有人工参与。该智能体可以读取和编辑文件并使用 Shell。未安装构建工具链，因此智能体无法编译代码或运行测试，修复结果通过检查最终提交（第 3.4节）进行评判。项目的规则文件对智能体而言是只读的。我们测试了三个规模和价格不同的模型：最小的模型是 Claude Haiku 4.5，中等模型是 Claude Sonnet 5.5，最大的模型是 Claude Fable 5.1（API 标识符 `claude-haiku-4-5-20251001`、 `claude-sonnet-5-5` 和 `claude-fable-5-1`）。

3.3 实验设计

主要实验比较了两种工作流： *Alone* 和 *+CA*。两者的区别在于要求智能体执行的任务以及它接收的信息。为衡量 CA 的删除检查所起的作用，我们还运行了 +关闭该检查的 CA，作为一项消融实验（第 4.2 节）。第 6 节加入了与 GitHub Copilot 代码审查的比较，第 7 节则比较了单独检测与免费静态分析工具的效果。

Alone。 智能体获得了一个位于拉取请求分支上的代码仓库克隆，并被要求根据项目的约定和结构审查拉取请求的更改，修复该拉取请求引入的任何问题，保持更改精简且符合惯例，然后提交并推送。在这些运行开始前，我们从代码仓库及其历史记录中彻底移除了检查器的所有痕迹。十个项目中有四个附带由其维护者编写的智能体指令文件（AGENTS.md 或 CLAUDE.md），在两种工作流中我们都保留了这些文件。其中一些文件描述了我们的缺陷所违反的规则，有一例还明确指出了这一点（禁止在某个目录中使用 `unsafe` code）。在这些项目中，单独工作的智能体可以读到部分相关规则。

+CA。 智能体被指示修复 CodeAssure（CA）在拉取请求中报告的发现项；它通过模型上下文协议（MCP）获取这些发现项。每个发现项都指出发生项目规则违规的位置，并说明修复目标，但不提供补丁。首次尝试后，如果仍有可采取行动的发现项，智能体还可以再进行一轮修复。CA 是由作者所在公司开发的商业检查器（第 8节）。除人工智能生成的代码外，它还会检查员工构建的智能体和自动化程序，以及连接到组织系统的人工智能助手，以识别安全风险和组织自身的规则违规。本研究仅使用其对人工智能生成的代码进行的检查。由于已知智能体会通过删除或编辑待测试的代码来满足检查要求（第 2节），因此，当拉取请求新增的文件被删除时，CA 也会向智能体提供反馈。该设计并不特定于 CA；任何向智能体报告已定位的规则级发现项的检查器，都可以采用相同方式进行评估。

每次运行都在拉取请求的独立副本上进行，使用一个新分支，该分支的头部提交是在原始头部提交之上添加的空提交，因此每次运行都从相同的代码树开始。30 个拉取请求中的每一个，都针对模型和配置的 9 种组合分别运行 5 次（Alone、 +CA，以及关闭删除检查的 +CA），共进行 1,350 次运行，其中 1,125 次针对有缺陷的拉取请求，225 次针对干净的拉取请求。同一配置下针对同一拉取请求的运行会连续安排。

3.4 结果指标

评分。 智能体进行最后一次提交后，评分器会针对每个植入缺陷应用相应的规范。研究团队独立于检查器实现了评分器，且评分器不使用检查器。评分器会给出三种结果之一。

- 已修复。 所有缺陷谓词均通过，且拉取请求新增的每个非测试文件仍然存在并且仍包含代码。

- 功能已移除。 每个缺陷谓词都通过了，但拉取请求新增的至少一个非测试文件已被删除，或其中的代码已被清空，无论该文件是否包含该缺陷。当文件中除了空行、注释、花括号以及导入、包含或命名空间行之外不再有任何内容时，该文件即视为空。若 Git 报告某文件已被重命名（相似度至少为 50%），或代码树中的其他位置存在同名文件，则该文件不算被删除。对于两个功能是重命名的拉取请求，撤销该重命名也算作功能已移除。在之后对评分结果进行审查后，我们新增了有关空文件和撤销重命名的规则，并据此对所有运行结果重新评分。其中九次运行的结果从已修复变为功能已移除。“功能已移除”是我们对删除或清空某个

新添加文件，或撤销重命名的简写。一个仍然包含代码的文件存续下来，并不能说明该功能仍然有效（第 8节）。

- 未修复。 至少有一个缺陷谓词未通过，无论是否同时删除了文件。因此，一个仍有一个缺陷未解决且删除了文件的多缺陷运行不算已修复。

只有在多缺陷拉取请求中的所有缺陷都已修复时，该拉取请求才算已修复，因此本文中的所有比率都是运行的占比，而不是 31 个单独缺陷的占比。我们报告两个指标。 修复率是我们的主要指标，表示已修复运行所占的比例。缺陷消除率表示已修复或功能已移除的运行所占的比例，也就是说，缺陷通过任何途径消失的运行所占比例。二者的差异来自智能体移除功能的运行，而这并不是可接受的修复。评分器不会构建代码，因此评分后，我们还对每个已修复运行进行了构建检查，并确认其保留了相应功能（第 4.6节）。 构建检查修复率表示已修复且通过该检查的运行所占比例，是我们的主要指标。如果正文未加限定地给出修复率，则指静态评分器的修复率。

对于无缺陷拉取请求，我们将最终提交分为未作改动（没有更改，或仅更改测试或注释）、代码编辑或功能移除。

成本。 一次运行的成本是 Claude Code 报告的智能体成本；若运行包含两个会话，则将两者的成本相加；此外，还包括 CA 工作流中检查器的用量。基础设施、许可证及其他费用不计入其中。每次更改的成本 是有缺陷拉取请求各次运行的平均成本。模型的公开应用程序编程接口（API）价格采用 2026年11月 的标价。

一致性。 为检验评分器，我们将其对同一最终提交给出的缺陷消除判定与检查器自身的判定进行比较。

3.5 统计分析

报告比率时，同时给出计数和 95% Wilson 得分区间 [25]，区间基于某一配置的 125 次运行。针对同一个拉取请求的五次运行难度相同，且连续安排，因此并非相互独立；我们将拉取请求视为推断单位。因此，我们还报告通过聚类自助法重采样拉取请求得出的 95% 区间 [4],，这能更好地反映对新拉取请求的不确定性。成本区间采用相同的自助法。所有自助法均使用一个固定种子抽取 10,000 个重采样样本，因此，同一配置若出现在两个表格中，其区间也相同。如果某一配置的每次运行均成功，则聚类自助法区间会收敛为一个点。这反映的是样本情况，并不代表我们对新拉取请求的结果确定无疑。

配对比较首先对每个拉取请求的多次运行取平均值。我们报告每个拉取请求的平均差值及其配对自助法区间，并将针对拉取请求进行的精确双侧配对置换（符号翻转）检验作为主要检验。当差异涉及的拉取请求较少时，该检验无法得到很小的 p 值。例如，若有四个拉取请求存在差异，可能达到的最小 p 值为 0.125。作为次要指标，我们报告精确 McNemar 检验 [18] ，该检验针对按拉取请求和重复次数匹配的运行进行。由于它将各次运行视为相互独立，因此会夸大显著性。评分器与检查器之间的一致性用 Cohen κ系数 [2]概括。

4 结果

4.1 修复率（RQ1）

表2给出了主要结果。加入 CA 发现结果后，每个模型的构建检查修复率都提高了：Haiku 从 26%（125 次运行中的 32 次）升至 69%（86 次），Sonnet 从 46%（57 次）升至 90%（113 次），Fable 从 45%（56 次）升至 93%（116 次）。仅使用静态评分器时，修复率分别从 28%（35 次）升至 86%（107 次）、从 47%（59 次）升至 94%（118 次），以及从 45%（56 次）升至 100%（125 次）。正如 25 个拉取请求所预期的那样，聚类自助法区间比 Wilson区间更宽，但对于任何模型，两种工作流之间的区间都不重叠。控制每个拉取请求难度的配对比较见表3。

表2：对25个有缺陷的拉取请求进行的结果，每种配置运行125次。构建检查通过：已修复且能够编译并保留功能的运行（第 4.6 节），并给出 Wilson 95%区间和聚类自助法95%置信区间。静态：仅使用静态评分器时的修复率，并给出 Wilson 95%区间。已移除：功能已移除的运行。成本：每次更改的平均成本，并给出聚类自助法95%置信区间。

配置	构建检查通过	（聚类）	静态	缺陷已消除 rate	移除后的每次更改成本	
仅使用 Haiku 4.5	26% [19, 34]	[11, 41]	28% [21, 36]	38% [30, 47]	13	$0.18 [0.14, 0.23]
Haiku 4.5 +CA	69% [60, 76]	[54, 82]	86% [78, 91]	91% [85, 95]	7	$0.42 [0.31, 0.53]
Sonnet 5.5 Alone	46% [37, 54]	[28, 63]	47% [39, 56]	72% [64, 79]	31	$0.06 [0.06, 0.07]
Sonnet 5.5 + CA	90% [84, 94]	[79, 99]	94% [89, 97]	96% [91, 98]	2	$0.17 [0.14, 0.19]
Fable 5.1 Alone	45% [36, 54]	[28, 62]	45% [36, 54]	80% [72, 86]	44	$0.91 [0.76, 1.08]
Fable 5.1 + CA	93% [87, 96]	[83, 100]	100% [97, 100]	100% [97, 100]	0	$1.73 [1.32, 2.18]

表3给出了静态评分器下的配对比较；第 4.6 节针对构建检查通过率重复了这些比较。在该指标下，CA 发现结果使每个模型的修复率提高了 43 至 48 个百分点（$p < 0.001$）。在静态评分器下，对于每个模型，CA 发现结果都提高了修复率（拉取请求级别 $p < 0.001$）和缺陷消除率（$p \leq 0.016$）。与 Alone 工作的 Fable 相比，加入 CA 发现结果的 Sonnet 也具有更高的修复率（$p< 0.001$），但缺陷消除率并未显著提高（$p = 0.062$）。对于加入 CA 发现结果的 Haiku 与 Alone 的 Fable，结论同样如此（$p = 0.001$，$p = 0.27$）。当修复还必须能够编译时，加入 CA 发现结果的 Haiku 仍领先于 Alone 的 Fable，但优势幅度已不再显著（第 4.6节）。

表3：针对25个有缺陷的拉取请求进行的配对比较。平均差值：每个拉取请求的平均差值及配对自助法95%区间。“仅A／仅B”统计按拉取请求和重复次数匹配的运行次数：A 已修复而 B 未修复（对于缺陷消除率，即缺陷在 A 中消除而在 B 中未消除）。p：针对拉取请求进行的精确双侧配对置换检验（主要检验）。次要的运行级麦克尼马尔检验中，除 Haiku + CA 相较于 Fable Alone 的缺陷消除率外（$p = 0.029$），所有比较的 p 值均为 < 0.001。

A 与 B	修复率：平均差值 [95%]	仅 A / B only	p	缺陷消除率：仅 A / 仅 B	p
Haiku +CA 与 Haiku Alone	+58分 [38, 74]	77 / 5	< 0.001	72 / 6	< 0.001
Sonnet + CA 与 Sonnet Alone	+47分 [28, 66]	62 / 3	< 0.001	33 / 3	0.014
Fable + CA 对比 Fable Alone	+55分 [38, 72]	69 / 0	< 0.001	25 / 0	0.016
Sonnet + CA 对比 Fable Alone	+50分 [33, 66]	62 / 0	< 0.001	25 / 5	0.062
Haiku + CA 对比 Fable Alone	+41分 [20, 60]	59 / 8	0.001	25 / 11	0.27

按缺陷类别。表 4 显示，效果主要集中在代码未揭示规则的情况下。单独工作时，模型在 15 次代码可见运行中修复了 14 次或 15 次。在架构规则方面，模型单独工作时，在 45 次运行中修复了 5 次（Haiku）至 19 次（Sonnet），而结合 CA 发现后则修复了 35 次至 45 次。与

CA 发现方面，Sonnet 和 Fable 已修复全部40次约定类运行中的问题，Haiku 则修复了38次。在多缺陷拉取请求中，模型单独运行时，在25次运行中有4至8次修复了所有缺陷；结合 CA 发现时，这一数字为22至25次。

表 4：按缺陷类别统计的已修复运行次数。

配置	代码可见	约定	架构 rule	多个
Haiku 4.5 Alone	15/15	11/40	5/45	4/25
Sonnet 5.5 Alone	15/15	20/40	19/45	5/25
单独使用 Fable 5.1	14/15	21/40	13/45	8/25
Haiku 4.5 +CA	11/15	38/40	35/45	23/25
Sonnet 5.5 + CA	15/15	40/40	41/45	22/25
Fable 5.1 + CA	15/15	40/40	45/45	25/25

单独工作时，各模型的表现并不按规模排序。Sonnet 在架构规则运行中修复的次数多于 Fable（45 次中为 19 次，对比 13 次），主要得益于删除。Fable 在 45 次架构规则运行中有 24 次删除了该功能。就缺陷消除率而言，两者相近（ 45 次中分别为 36 次和 37 次）。

对于代码可见缺陷，使用 CA 发现结果的 Haiku 修复了 15 次运行中的 11 次，而单独使用时修复了 15 次中的 15 次。四次失败都发生在同一个拉取请求中；在该请求中，检查器报告缺陷已消除，而评分器却没有。第 5 节将解释原因。

4.2 功能删除与删除检查消融（RQ2）

智能体单独工作时，删除该功能的情况很常见（表2，“已删除”列）。Fable 在 125 次运行中有 44 次（35%）这样做，Sonnet 有 31 次（25%），Haiku 有 13 次。删除主要发生在架构规则和多缺陷拉取请求中，其中植入缺陷通常位于拉取请求新增的文件里。当承载缺陷的代码是新代码时，删除它从字面上满足了“修复此拉取请求引入的任何问题”这一指令，也是获得干净审查结果的最短途径。使用会报告已删除功能的 CA 后，删除次数降至 7、2 和 0。

删除检查的消融实验。 为衡量删除检查的作用，我们关闭该检查，运行了 +CA 工作流；使用相同的 30 个拉取请求、每个模型运行五次，以及相同的智能体、指令、会话限制和评分器（表5）。关闭检查后，CA 的其他发现几乎没有减少 Sonnet 的删除次数（运行 29 次，而单独运行时为 31 次），但增加了 Haiku 的删除次数（从 13 次增至 19 次）；三个模型的删除次数总计从 375 次运行中的 9 次升至 65 次。

表5：删除检查的消融实验： +CA 启用检查（如表2所示）与关闭检查时，在25个有缺陷的拉取请求上的表现（每种配置运行125次）。修复率：先给出 Wilson 95%区间，再给出聚类自助法95%置信区间。成本：每次更改的平均成本，包括检查器，并附聚类自助法95%置信区间。

配置	修复率	（聚类）	缺陷消失 rate	已删除	2nd 会话	每次更改的成本
Haiku 4.5 + CA, 取消勾选	**71%** [63, 78]	[57, 84]	**86%** [79, 91]	19	33	**$0.37** [0.28, 0.46]
Haiku 4.5 +CA	**86%** [78, 91]	[75, 94]	**91%** [85, 95]	7	41	**$0.42** [0.31、0.53]
Sonnet 5.5 + CA, 勾选	**74%** [65, 81]	[58, 87]	**97%** [92, 99]	29	19	**$0.14** [0.12, 0.15]
Sonnet 5.5 + CA	**94%** [89, 97]	[86, 100]	**96%** [91, 98]	2	46	**$0.17** [0.14, 0.19]
Fable 5.1 + CA, 勾选	**86%** [79, 91]	[73, 98]	**100%** [97, 100]	17	7	**$1.22** [1.01, 1.45]
Fable 5.1 + CA	**100%** [97, 100]	[100, 100]	**100%** [97, 100]	0	18	**$1.73** [1.32, 2.18]

关闭检查后，所有模型的修复率都下降了：Haiku 下降 14 个百分点（95% 区间[3, 27], $p = 0.036$），Sonnet 下降 21 个百分点（[7, 35], $p = 0.010$），Fable 下降 14 个百分点（[2, 27], $p = 0.12$）；这些结果来自与完整 CA 的配对比较。三个模型的配对区间均不包含零，且以拉取请求为单位的检验对 Haiku 和 Sonnet 达到显著性。Fable 的差异来自少数几个拉取请求，因此该检验的 p 值很难达到更低水平。所有模型的缺陷消除率变化都很小且不显著，这表明该检查会将删除转化为修复，而不是留下未修复的缺陷。该检查也有成本：更多运行需要第二个会话（Fable 为 18 次，而非 7 次），启用检查后的每次更改成本比关闭检查时高 13% 至 42%。该检查实现了预期目的：智能体会保留功能并重写代码，而不是将其删除。在构建检查下，关闭检查时 Haiku、Sonnet 和 Fable 的修复率分别为 54%、70% 和 85%；该检查对 Haiku（提高 15 个百分点，$p = 0.023$）和 Sonnet（提高 21 个百分点，$p = 0.010$）的影响仍然显著。对 Fable 而言，影响降至 8 个百分点（$p = 0.25$），因为某个项目中的一些重写无法通过编译（第 4.6 节）。

在仍以功能已移除告终的 9 次启用 CA 的运行中，检查器有 6 次报告了删除，但智能体在剩余会话中没有恢复该功能。其余 3 次是 Haiku 的运行。其中两次，检查器没有报告删除，我们正在调查原因。第三次，智能体清空了文件，而非将其删除；删除检查不涵盖这种情况。启用 CA 且缺陷仍然存在的 16 次运行分为三组。在 10 次运行中，智能体没有添加项目要求为新命令或组件编写的测试。4 次是第 5 节所述拉取请求中的 Haiku 运行，检查器只报告了 4 个失效调用中的 1 个。其余 2 次是 Rust 拉取请求中的 Haiku 运行，都留下了软件包之间被禁止的依赖关系。

同一作者此前进行的一项单次运行研究，在另一组 33 项更改和较早版本的 Sonnet 模型上使用同一检查器，报告的修复率为 91% 至 100%。在该研究中，删除文件需要交互式批准，而无人值守的运行无法提供批准，因此智能体不得不修复代码。这或许可以解释两项研究结果差异的一部分，但我们没有在两项研究共同涉及的拉取请求上对此进行测试。

4.3 无缺陷拉取请求

表6展示了智能体在五个无缺陷拉取请求上的行为。当被要求审查并修复一项正确的更改时，单独工作的智能体通常会编辑代码：Haiku 在64%的运行次数中进行了编辑，Sonnet 为100%，Fable 为80%，平均修改8到22行。Fable 还在5次运行中删除了该功能。在无缺陷拉取请求上进行的所有 75 +CA 运行中，检查器均未报告任何可采取行动的问题；在每次运行中，智能体都保持更改不变，就像关闭删除检查时一样。这些运行的成本也低得多，因为智能体无事可做。

表6：5个无缺陷拉取请求的结果，每种配置的运行次数为25次。

配置	保持不变	不必要 代码编辑	功能 已移除	平均更改行数 更改	每次更改的成本
仅 Haiku 4.5	9	16（64%）	0	7.8	$0.11
仅 Sonnet 5.5	0	25（100%）	0	8.5	$0.06
Fable 5.1 Alone	0	20（80%）	5	21.6	$0.73
每个模型 + CA	25	0	0	0.0	$0.03 至 $0.27

4.4 评分器有效性

在 Alone 工作流和消融实验（第 4.2 节）针对有缺陷的拉取请求进行的 750 次运行中，有 748 次同时获得了评分器判定和检查器判定；研究结束后，我们对每次运行的最终提交都运行了检查器。在两次独立运行中，有一种规则类型的检查器判定未被收集。评分器与检查器在 94.8% 的运行中判定一致（Cohen κ系数为 0.84）。有 24 次运行中，检查器报告缺陷已消除，而评分器没有。这里有 14 次是由于检查器的两条规则存在疏漏，这些疏漏是在本研究期间发现的。其余 10 次是 Haiku 的运行：4 次针对第 5 节讨论的拉取请求，另外 6 次针对两个我们仍在调查的 Rust 拉取请求。有 15 次运行中，评分器报告缺陷已消除，而检查器没有；其中 4 次是由于检查器的一条规则出错。两者判定不一致时，我们报告评分器的判定。

一致性表明由同一团队构建的两种工具彼此一致，但不能说明其中任何一种是正确的。对最终差异进行盲审会是更有力的检验，这留待未来工作完成。

4.5 成本与工作量（研究问题3）

表7总结了工作量和成本。计入检查器后，Haiku 每次更改的成本从 $0.18 上升至 $0.42，Sonnet 从 $ 0.06 上升至 $0.17，Fable 从 $0.91 上升至 $1.73。Haiku 的智能体步骤数从 43 增至 60，Sonnet 从 12 增至 16，Fable 从 22 增至 34。Haiku 的运行中有 33% 需要第二个会话，Sonnet 的运行中有 37% 需要第二个会话，Fable 的运行中有 14% 需要第二个会话。增加的成本中，13% 至 42% 来自删除检查（第 4.2 节）。

表7：有缺陷的拉取请求中每次更改所需的工作量和成本。

模型	步骤数，单独 / 使用	使用第二个会话的运行 会话	每次更改的成本，单独 / 搭配
Haiku 4.5	43 / 60	125 个中的 41 个	$0.18 / $0.42
Sonnet 5.5	12 / 16	125 个中的 46 个	$0.06 / $0.17
Fable 5.1	22 / 34	18/125	$0.91 / $1.73

按成本效益计算，CA 发现会改变排名。仅统计同时能够通过构建的修复，Sonnet 在有 CA 发现的情况下修复了 90% 的运行，每次更改的平均成本为 $0.17；而 Alone 的 Fable 修复了 45%，每次为 $0.91（配对差异为 46 个百分点，$p < 0.001$）。绝对结果最佳的是有 CA 发现的 Fable，修复率为 93%，每次更改的成本为 $1.73。

4.6 构建和功能检查

评分器会读取最终提交，但不会构建该提交（第 3.4 节）。为了了解有多少修复同时能够通过构建，在对所有运行进行评分后，我们构建了主研究和 Copilot 对比中每个被评分为已修复的缺陷拉取请求的最终提交（共 851 次运行）。Rust 项目使用`cargo check`进行检查，C++ 项目使用 CMake 构建，Java 使用 `mvn test-compile`，TypeScript 使用 `tsc`；Python 文件则使用 Ruff 进行编译，并检查未定义的名称。navigation2 拉取请求需要 ROS 2 安装，因此对于其 26 次已修复的运行，我们只检查代码使用的每个项目头文件和符号是否存在。为了区分智能体的更改与拉取请求自身的状态，我们还构建了基础提交和拉取请求的头部提交，并且只统计头部提交中不存在的错误。四个拉取请求头部提交由于与植入缺陷无关的原因，本身无法通过构建；在其中一个项目中，拉取请求添加的文件不属于构建内容，因此没有统计从这些头部提交继承的错误。我们还检查了每次已修复运行的差异，以查找存根和被原地移除的功能，而第 3.4 节的文件级规则无法检测到这些问题。

CA 工作流的优势在必须确保修复也能通过编译时依然成立。对于 CA 发现，375 次运行中有 315 次（84%）生成的修复同时通过了构建与功能检查；而同一批智能体单独工作时，375 次运行中只有 145 次（39%）通过。Sonnet 和 Fable 在有 CA 发现时，这一比例为 250 次中的 229 次（92%）。在所有工作流中，Java、TypeScript 和 Python 的每项修复都通过了编译；在 CA 工作流中，Java 或 TypeScript 的修复也没有任何一项未通过检查（各 15 次运行中均为 0 次；表 8）。

在 851 次修复中，有 75 次未通过：50 次无法编译，全部发生在 Rust 和 C 中++；21 次 Python 修复删除了拉取请求所新增的行为，却保留了相关文件（验证器从模式类中删除，且其他地方没有任何代码应用它）；还有 4 个 Rust 和 C 存根++。编译失败发生在智能体没有编译器可用的情况下。其中，Rust 的修复失败占比最高（占该语言修复的 14%）：智能体调用了不存在的函数，或违反了其线程安全规则。原地删除发生在两个 Python 拉取请求中，这两个请求植入的缺陷都是从服务层导入模式层内容。删除验证器符合分层规则，而删除检查只会查找被删除或清空的文件，因此无法检测到这种情况。大多数失败来自最小的模型：在 CA 工作流中，Haiku 配合 CA 发现造成了 35 次失败中的 21 次。表格9 按配置比较了这两种衡量方式。

表 8：按语言划分的构建与功能检查：由静态评分器评分的修复，以及未通过检查的修复，涵盖所有工作流，以及仅进行删除检查的 +CA。

语言	已检查	无法编译	原地移除或存根	使用 +CA 失败
Java	51	0	0	15 个中的 0 个
TypeScript	30	0	0	15 个中的 0 个
Python	202	0	21	70 个中的 8 个
C++	305	16	2	137 个中的 8 个
Rust	263	34	2	113 个中的 19 个
总计	851	50	25	350 个中的 35 个

采用更严格的衡量方式后，所有主要结果仍然成立。CA 发现提高了每个模型的修复率（Haiku +43 个百分点，Sonnet [23, 62]，个百分点，Fable +45 [26, 64] 个百分点；拉取请求级别 p +48 [31, 65].001）。配合 CA 发现的 Sonnet 修复的运行次数仍是 Alone 的两倍（90% 对 45%，$p < 0.001$），成本约为其五分之一。对于两种模型，CA 工作流的表现仍优于 Copilot 代码审查（$p < 0.002$）；删除检查仍显著提高了 Haiku（$p \leq 0.023$）和 Sonnet（$p = 0.010$）的修复率。两项次要比较则不那么明确。配合 CA 发现的 Haiku 仍优于 Alone 的 Fable（$= 0$ 个百分点 +24 [−1,），但差异不再显著（p 47].075）。对于 Fable，删除检查促使智能体保留文件，而不是将其删除；但某个项目中有 9 次重写无法编译，因此其效果降至 8 个百分点（$= 0$，p [−2, 19].25）。

表 9：静态评分器下以及经过构建与功能检查后的 25 个有缺陷的拉取请求的修复率（主要指标）。每种配置的运行次数为 125 次，Copilot 代码审查为 72 次。

模型	工作流	静态评分器	构建检查通过
Haiku 4.5	Alone	35（28%）	32（26%）
	+CA	107 (86%)	86 (69%)
	+CA，关闭删除检查	89 (71%)	67 (54%)
	Copilot 审查	29（40%）	23（32%）
Sonnet 5.5	Alone	59（47%）	57（46%）
	+CA	118（94%）	113（90%）
	+CA，关闭删除检查	92（74%）	87（70%）
	Copilot 审查	33（46%）	33（46%）
Fable 5.1	Alone	56（45%）	56（45%）
	+CA	125 (100%)	116 (93%)
	+CA，删除检查已关闭	108 (86%)	106 (85%)

该检查用于确认修复能够编译并保留其功能；它不测试行为。大多数植入的功能都很小，没有可观察的行为可供测试，因此我们没有运行项目测试。

5 讨论

规则发现与修复 最大的提升来自那些只存在于项目自身惯例和结构中的规则。独自工作时，智能体必须先发现这类规则，再判断改动是否违反了规则。CA 发现替它完成了这两个步骤。剩下的工作是按照明确的目标修复已定位的违规，而三个模型在大多数运行中都做到了这一点。这与反馈质量限制自我修复能力的观察结果相符 [20], ，也表明，对于这类任务，模型之间的大部分差异在于规则发现，而非违规修复。

模型规模与审查质量 独自工作时，最大模型的修复率并未超过中等规模模型，而且它删除功能的次数最多。能力更强的智能体也可能更愿意采取大幅度行动，包括移除它认为有问题的代码，但我们没有对此进行测试。在我们的数据中，模型规模无法预测智能体审查自身改动的效果，而 CA 发现则提升了每个模型的表现。其他研究通过改变模型本身（例如知识蒸馏），使较小的模型也能胜任 [13]，或通过组合较小的领域专家 [15]。而在这里，改变的是模型的输入。

对删除进行评分 在关闭删除检查的消融实验中，如果指标只询问缺陷是否消失，报告的成功率会是 86% 至 100%，而静态修复率为 71% 至 86%。对于 Fable 单独运行的情况，该指标会报告 80%，而非 45%，从而将 Fable 单独运行排在 Sonnet 单独运行（72%）之上，这与它们按修复率排名的顺序相反。我们建议，评估智能体修复时，应将删除作为单独的结果报告；反馈系统则应像 CA 那样，将移除标记文件视为一项独立发现。若没有这项发现，CA 的其他发现几乎无法减少 Sonnet 的删除行为，对 Haiku 的删除行为则毫无影响（第 4.2节）。

编译型和解释型语言 构建检查将这些语言分为两类。在 Java、TypeScript 和 Python 这几种人工智能生成的代码最常见的语言中，所有修复都通过了编译；其余失败的情况是 Python 修复：它们通过删除相关行为，而非迁移该行为，满足了分层规则。在 Rust 和 C++ 中，缺少编译器的智能体——尤其是最小的模型——编写的修复虽然达到了修复目标，却无法构建。由此可在反馈循环中加入两项改进：对于编译型语言，将编译器错误反馈给智能体；并将删除检查的范围从已删除文件扩展到已删除函数和验证器。

没有缺陷的变更 对于无缺陷拉取请求，独自工作的智能体几乎总会修改一些内容。它们似乎把“审查并修复任何问题”这一指令理解成存在问题的信号。检查器未报告任何问题时，智能体便会停止，从而避免不必要的编辑，也降低了成本。这一比较无法区分发现结果的影响与指令差异的影响。独自工作的智能体被要求广泛审查变更，而 +CA 智能体被要求修复发现项，但当时并无发现项。我们没有测试在两个智能体收到相同审查指令时，发现结果是否能避免不必要的编辑。

检查器的失败案例 在 24 次运行中，检查器漏掉了评分器发现的一处缺陷。第 4.4节介绍了检查器两条规则的不足。另一个案例进一步说明，检查器的遗漏会如何降低智能体的结果。

海句在代码可见缺陷上的表现较差，原因在于一项拉取请求。该请求将共享函数的一个新参数设为必填，却没有为该函数已有的四处调用添加新实参。只有四处调用全部修正，这处缺陷才算已修复。检查器只向智能体报告了四处调用中的一处，没有报告另外三处。在五次 +CA 运行中，有四次海句修正了被报告的调用后便停止了。此后检查器未再报告任何发现项，因此运行结束时仍有三处调用存在问题。独自工作时，海句在五次运行中都修复了该缺陷：要么修正全部四处调用，要么为新参数提供默认值，使已有调用继续有效。无论是否有 CA 发现，Sonnet 和 Fable 都在每次运行中为该参数提供了默认值，只需修改函数本身一次，就能修复全部四处调用。有 CA 发现时，海句只修复了被指出的那处调用，将发现结果视为完整清单。

依赖检查器的智能体会漏掉检查器遗漏的内容，而在这里，最小的模型将这些发现结果视为完整结果。在已部署的循环中，遗漏一条规则或一个调用点，就意味着智能体停止时仍有缺陷存在。两项改动或许可以降低这一风险。检查器可以报告每个受影响的位置，包括那些确定性较低的位置；同时，智能体的指令可以要求它在发现结果之外寻找相关问题，以便外部发现补充其自身的审查，而不是取代审查。研究结束后，检查器进行了修改，现在会报告此拉取请求中的全部四个调用。

6 与 GitHub Copilot 代码审查的比较

主要实验将外部发现与完全没有帮助的情况进行比较。一种常见的替代方案是对拉取请求进行 AI 审查。因此，我们使用 GitHub Copilot 代码审查，为 Haiku 和 Sonnet 运行了一个工作流，处理相同的 30 个拉取请求，并使用相同的评分器。

GitHub Copilot 代码审查（+Copilot）。对于每个拉取请求和每次重复运行，GitHub Copilot 都以默认的“平衡”工作强度和默认配置审查拉取请求。它看到的代码仓库与其他工作流相同，包括维护者自己的指令文件和贡献者指南。我们没有添加自定义审查指令（一个 `.github/copilot-instructions.md` 文件），团队可以用它来为 Copilot 指定自己的规则。我们将其审查评论交给智能体，由智能体通过一次会话处理这些评论。这反映了常见做法，即一个拉取请求接受一次 AI 审查。为控制成本，Copilot 只运行了三次，而非五次。

表 10：在 25 个有缺陷的拉取请求上，Alone、 +Copilot 和 +CA 的表现；以及在 5 个无缺陷拉取请求上出现不必要代码编辑的运行情况。修复率：Wilson 95%区间。这些工作流的差异不仅在于提供的帮助。 +Copilot：一次会话、无自定义审查指令、重复运行三次；针对 24 个有缺陷的拉取请求（每个模型运行 72 次）和 5 个无缺陷的拉取请求（每个模型运行 15 次），另有一个拉取请求被排除（见正文）。+CA：最多两次会话。因此，表格比较的是实际运行的工作流。

模型	工作流	修复率	已移除	干净：代码编辑	与CA配对比较
Haiku 4.5	Alone	28% [21, 36]	13	16 / 25	77 / 5，$p < 0.001$
Haiku 4.5	+ Copilot	40% [30, 52]	13	12 / 15	38 / 5，$p < 0.001$
Haiku 4.5	+ CA	86% [78, 91]	7	0 / 25	
Sonnet 5.5	Alone	47% [39, 56]	31	25 / 25	62 / 3，$p < 0.001$
Sonnet 5.5	+ Copilot	46% [35, 57]	9	12 / 15	37 / 1，$p < 0.001$
Sonnet 5.5	+ CA	94% [89, 97]	2	0 / 25	

对于这两个模型，CA 的修复率都高于单独工作和 Copilot 的审查（拉取请求级别 $p < 0.001$）。配对列给出了仅通过 CA 修复的运行次数与仅通过另一种工作流修复的运行次数之间的比较，以及拉取请求级别的 p 值。

Copilot 的审查并未显著改变修复率。在与智能体单独工作的配对比较中，对于 Haiku，有 17 次运行仅通过 Copilot 的审查得到修复，而有 9 次在没有 Copilot 的情况下得到修复（$p = 0.35$）；对于 Sonnet，这一数字分别为 8 次和 12 次（$p = 0.65$）。在无缺陷拉取请求中，两个模型都在 15 次运行中的 12 次根据 Copilot 的注释采取了行动并编辑了代码。Copilot 的注释涉及代码质量，并且至少有一次根据该问题对设计造成的影响，识别出了一个分层问题。

Copilot 反复未能审查某个拉取请求。其完整审查未能启动，因此退回到最轻量的设置，之后的请求则完全没有收到审查。因此，我们将该拉取请求从 Copilot 工作流及其配对比较中排除。Copilot 的运行（重复 1 至 3）与同一拉取请求上其他工作流的重复 1 至 3 进行了配对。作为敏感性检查，将被排除的拉取请求在三次运行中都计为已修复，或一次都不计为已修复，得到的 Haiku 修复率分别为 39% 至 43%，Sonnet 修复率分别为 44% 至 48%。这两个界限都不会改变其与单独工作（$p \geq 0.21$）或与 CA（$p < 0.001$）的比较结果。

Copilot 的审查具有通用性。在默认配置下，它会依据通用最佳实践进行审查，而 CA 发现结果带来的最大收益则来自项目特定规则（第 4 节）。可以通过自定义审查指令向 Copilot 提供此类规则，但这些规则必须先针对每个项目分别编写，而我们并未对此进行测试。

7 通过免费静态分析工具进行检测

我们考察了静态分析工具在没有专用检查器的情况下能否发现这些缺陷。为了区分 CodeAssure 检查器与 AI 代理本身的作用，我们仅比较了检测结果，没有运行 AI

智能体。我们在每个拉取请求上运行了九种广泛使用的免费工具，采用其默认或标准配置，且不使用项目特定规则：Semgrep 社区版 1.185.0，使用其公开的 `p/default` 和语言规则包；SonarQube 社区构建版 26.11，使用其默认的“Sonar way”配置文件；用于 Python 的 Pylint 4.1 和 Ruff；ESLint 9，采用其推荐的 TypeScript 规则；用于 C++ 的 clang-tidy 23 和 cppcheck 2.22（警告、风格、性能和可移植性检查）；用于 Java 的 PMD，采用其推荐的“quickstart”规则集；以及用于 Rust 的 `cargo clippy` 构建，采用默认 lint 规则，并且还会运行编译器。每种工具都会分析拉取请求所更改的文件，以及包含植入缺陷的所有文件。如果工具在缺陷所在文件中、距离缺陷三行以内报告发现项，或者针对缺失的配套测试在文件任意位置报告发现项，我们就认为它检测到了该缺陷。我们逐一手动检查了每个匹配项，并剔除了三个与缺陷无关的结果：Pylint 报告的未解析第三方导入，因为项目依赖项未安装；以及 Semgrep 在过时调用旁边（但并非针对该调用）报告的一条 `unsafe` 警告。对于 CA，我们使用了每个拉取请求上的首次发现结果，也就是智能体在每次运行开始时收到的结果。

表 11：开箱即用运行的工具对 31 个植入缺陷的检测情况。范围内：工具所分析语言中的缺陷。未涵盖：其他语言中的缺陷。没有任何工具（包括 CA）在 5 个无缺陷拉取请求所修改的行上报告发现项。

Tool	语言	范围内	已检测	未检出范围内	Not 已覆盖	Not 捕获了（共 31 项）
CA	全部五项	31	31[b]	0	0	0
Rust 构建和 Clippy	Rust	10	6	4	21	25
Semgrep	全部五个	31	2	29	0	29
Pylint	Python	7	1	6	24	30
SonarQube Community	Python, TypeScript、Java	9	1	8	22	30
Ruff	Python	7	0	7	24	31
ESLint	TypeScript	1	0	1	30	31
clang-tidy	C++	12	0	12	19	31
cppcheck	C++	12	0	12	19	31
PMD	Java	1	0	1	30	31
所有免费工具合计	全部五个	31	8	23	0	23

[b] 如果某个工具报告了受影响行中的至少一行，则视为检测到了该缺陷。有些检测并不完整：例如，CA 检测到拉取请求中 4 个失效调用里的 1 个（第 5 节）；Rust 构建检测到 3 个，它标记了依赖声明，却没有标记使用该依赖的代码；SonarQube 检测到 1 个。

这些免费工具总共检测到 31 个缺陷中的 8 个。Rust 构建检测到的最多。Rust 软件包之间人为植入的三个依赖会造成依赖循环，导致构建失败；此外，失效的重命名以及在禁止使用标准库的 crate 中使用标准库，都会导致编译失败。Tock 自身的 crate 设置也禁止`unsafe` 代码，Semgrep 的通用 Rust 规则同样会对此发出标记。Pylint 检测到一个 Python 拉取请求中的全部四个失效调用，而 CA 检测到其中一个，SonarQube 则检测到四个中的一个。对 Java 项目进行构建（我们没有运行构建）也会发现该项目中的失效重命名。如果某个项目的规则写入其构建设置中，构建就会强制执行这些规则。

其余 23 个缺陷仅由 CA 检测到。它们违反了 C++、Python、TypeScript 和 Java 项目特定的约定与分层规则，或违反了新组件必须随附测试的要求。没有任何免费工具默认包含这些规则，而 SonarQube Community 完全不分析 C、C++ 或 Rust。CA 配置了各个项目的规则，包括我们植入的规则类型，因此这项比较体现的是了解项目规则的价值，而不是 CA 是更好的通用分析器。Semgrep 和 SonarQube 等一些免费工具支持自定义规则，但必须有人先为每个项目编写这些规则。

8 有效性威胁

内部有效性。 真实项目的规则可能不够完整，这会降低发现结果带来的益处。评分器由同一团队独立于检查器实现，因此严格来说并非独立；它与检查器的一致性（κ系数为 0.84）表明结果一致，而非判断正确。研究没有对最终差异进行盲审。Alone 和 +CA 工作流的差异不止于发现结果。两者使用了不同的指令，因为各自反映了智能体在实际中的使用方式；此外，如果仍有发现结果， +CA 还允许进行第二个会话。Copilot 工作流使用 GitHub 自有模型进行审查，并且只进行一次修复会话，不同于 CA 允许的两次会话；该工作流在 25 个有缺陷的拉取请求中，有 24 个进行了三次重复运行。

构念效度。 根据第 3.4 节的规则，只有当每个植入缺陷都满足模式级规范，且拉取请求新增的每个非测试文件都得以保留并仍包含代码时，该次运行才算修复成功。这一静态度量本身并不能说明代码是否能够编译，也不能说明功能是否仍然正常，因此我们增加了构建与功能检查（第 4.6 节），并将构建检查修复率作为主要指标。所有主要比较在该指标下均成立，我们也报告了两种度量。所有 Java、TypeScript 和 Python 修复均成功编译。在 851 项修复中，有 75 项未通过检查：包括 Rust 和 C++中的编译失败，以及原位移除验证器的 Python 修复，这类情况主要来自最小的模型。该检查成功构建了 25 个有缺陷的拉取请求中的 24 个，但 navigation2 除外；它也没有运行项目测试，因此即使修复能够编译，语义上仍可能有误。四个拉取请求的头部提交无法单独编译；此外，在一个项目中，新增文件不属于构建的一部分，这限制了该检查对这些拉取请求所能说明的情况。

外部有效性。 我们研究了一个模型家族和一个智能体。第 7 节中的免费工具运行时未使用项目特定规则，而 clang-tidy 运行时未使用项目的构建配置，这限制了其可分析的内容。拉取请求相对简洁，包含一至三个中等规模的缺陷，且所涉及的变更经过编写，旨在模拟真实贡献。涵盖五种语言的十个项目在一定程度上限制了研究结果的普适性。

结论有效性。 在 25 个有缺陷的拉取请求中，拉取请求之间存在显著差异，这一点从聚类自助法区间中可以看出。将拉取请求视为分析单位的检验仍显示主要比较结果具有显著性，但删除检查消融对 Haiku 和 Sonnet 显著，而对 Fable 不显著（第 4.2节）。表格中报告的运行级麦克尼马尔检验将重复视为相互独立，因此夸大了显著性。模型行为会随版本变化，我们的结果适用于截至 2026年11月所指定的版本。

利益冲突。 作者隶属于 CodeAssure Labs，该机构开发了本研究评估的 CodeAssure 检查器。为降低偏差风险，缺陷及其评分规范均在任何智能体运行前编写，评分器不使用该检查器，每次运行均予以报告，并且每次运行的结果和分析代码均已公开（见数据可用性）。

9 结论

在我们的研究中，采用一种工作流，由单个商业检查器提供外部发现，定位每项规则违规并说明修复目标，使三个大语言模型编码代理对植入缺陷的修复率提高了 43 至 48 个百分点，达到 69% 至 93%；这里只统计能够编译且保留其功能的修复。在仅使用静态评分器时，所有 Java、TypeScript 和 Python 修复均能编译，修复率提高了 47 至 58 个百分点，达到 86% 至 100%。该工作流还使一个中型模型以极低的成本胜过最大模型。智能体经常通过删除拉取请求新增的文件来移除缺陷，而朴素成功指标会将此计为修复。CA 的删除检查基本上能避免这种情况：在关闭该检查的消融实验中，删除行为从 375 次运行中的 9 次上升至 65 次，修复率下降了 8 至 21 个百分点；对于 Haiku 和 Sonnet，这一下降在拉取请求层面具有统计显著性。该工作流的表现也优于采用默认配置的 GitHub Copilot 代码审查；九种开箱即用的免费静态分析工具合计仅检测出 31 个缺陷中的 8 个。我们建议，在评估编码代理时，应将删除行为单独评分，并将其作为发现项纳入反馈循环。

数据可用性

每次运行的结果（包括拉取请求的不透明标识符、缺陷类型、模型、工作流、重复次数、评分器结果、会话数、步骤数和总成本）、每种工具对每个植入缺陷的检测结果，以及一份可重新计算本文所有表格和测试的脚本，均已公开，网址为 https://github.com/CodeAssureLabs/external-feedback-coding-agents。这些资料可以用于核查统计数据，但不能用于重新评分各次运行。完整的逐次运行记录（检查器发现、智能体对话记录和差异）、评分器规范以及确切的智能体指令，可通过 hello@codeassurelabs.com 联系索取，并须遵守保密协议。本文不介绍检查器规则格式及其内部组件。

参考文献

[1] Scott Blyth、Sherlock A. Licorish、Christoph Treude 和 Markus Wagner。静态分析作为反馈循环：提升大语言模型生成的代码，使其超越正确性。*arXiv*预印本 *arXiv:2508.14419*，2025年。[2] Jacob Cohen。名义尺度的一致性系数。*Educational and Psychological Measurement*，20(1)：37–46，1960年。[3] Lakshitha de Silva 和 Dharini Balasubramaniam。控制软件架构侵蚀：一项综述。*Journal of Systems and Software*，85(1)：132–151，2012年。[4] Bradley Efron 和 Robert J. Tibshirani。*An Introduction to the Bootstrap*。Chapman and Hall，1993年。[5] Akeela Darryl Fattha、Kia Ying Chua、Lingxiao Jiang 和 Laura Wynter。大语言模型智能体中的探索结构：多文件变更定位。*arXiv*预印本 *arXiv:2606.11976*，2026年。[6] Jonathan Gabor、Jayson Lynch 和 Jonathan Rosenfeld。EvilGenie：奖励劫持基准。*arXiv*预印本 *arXiv:2511.21654*，2025年。[7] Supriyo Ghosh、Laura Wynter、Shiau Hong Lim 和 Duc Thien Nguyen。神经渐进式对冲：利用随机规划在强化学习中施加约束。载于*Proceedings of theThirty-Eighth Conference on Uncertainty in Artificial Intelligence (UAI)*，*Proceedings ofMachine Learning Research*第180卷，第707–717页。PMLR，2022年。[8] Thibaud Gloaguen、Niels Mündler-Sasahara、Mark Niklas Müller、Veselin Raychev 和 Martin Vechev。评估 AGENTS.md：代码仓库级上下文文件是否有助于编码代理？*arXiv*预印本 *arXiv:2602.11988*，2026年。[9] Jie Huang、Xinyun Chen、Swaroop Mishra、Huaixiu Steven Zheng、Adams Wei Yu、Xinying Song 和 Denny Zhou。大语言模型尚不能自我纠正推理。载于*International Conference onLearning Representations (ICLR)*，2024年。[10] Carlos E. Jimenez、John Yang、Alexander Wettig、Shunyu Yao、Kexin Pei、Ofir Press 和 Karthik Narasimhan。SWE-bench：语言模型能否解决真实世界的 GitHub 问题？载于*InternationalConference on Learning Representations (ICLR)*，2024年。[11] Pascal Joos、Islem Bouzenia 和 Michael Pradel。CodeCureAgent：静态分析警告的自动分类与修复。*arXiv*预印本 *arXiv:2509.11787*，2025年。[12] Victoria Krakovna、Jonathan Uesato、Vladimir Mikulik、Matthew Rahtz、Tom Everitt、Ramana Kumar、Zac Kenton、Jan Leike 和 Shane Legg。规范博弈：人工智能的巧思的另一面。DeepMind 博客，2020年。[13] Achintya Kundu、Fabian Lim、Aaron Chew、Laura Wynter、Penny Chong 和 Rhui Dih Lee。面向边缘应用高效蒸馏大语言模型。载于*Proceedings of the 2024 Conference of the North AmericanChapter of the Association for Computational Linguistics (NAACL), Industry Track*，2024年。[14] Rhui Dih Lee 和 Laura Wynter。协作式地向大语言模型添加新知识。*arXiv*预印本 *arXiv:2410.14753*，2024年。

[15] Rhui Dih Lee、Laura Wynter 和 Raghu Kiran Ganti。将大语言模型灵活且有效地混合到领域专家混合体中。arXiv预印本 arXiv:2408.17280，2024年。[16] M. Danish Lim、I. Danial Bin Sharudin、Wen Han Chen、Cedric Lim 和 Laura Wynter。面向知识支撑的工具使用工作流的 AI 代理声明式技能。arXiv预印本 arXiv:2606.06923，2026年。[17] Aman Madaan、Niket Tandon、Prakhar Gupta、Skyler Hallinan、Luyu Gao、Sarah Wiegreffe、Uri Alon、Nouha Dziri、Shrimai Prabhumoye、Yiming Yang 等。自我精炼：通过自我反馈进行迭代式精炼。载于神经信息处理系统进展（NeurIPS），2023年。[18] Quinn McNemar。关于相关比例或百分比之差的抽样误差的注记。《心理测量学》，12(2)：153–157，1947年。[19] Gail C. Murphy、David Notkin 和 Kevin Sullivan。软件反思模型：弥合源模型与高层模型之间的鸿沟。载于 ACM SIGSOFT软件工程基础研讨会第3届会议论文集，第18–28页，1995年。[20] Theo X. Olausson、Jeevana Priya Inala、Chenglong Wang、Jianfeng Gao 和 Armando Solar-Lezama。自我修复是代码生成的万能药吗？载于 International Conference on Learning Representations（ICLR），2024年。[21] OpenAI。推出 SWE-bench verified。https://openai.com/index/introducing-swe-bench-verified/，2024年。[22] Noah Shinn、Federico Cassano、Ashwin Gopinath、Karthik Narasimhan 和 Shunyu Yao。反思：采用语言强化学习的语言智能体。载于神经信息处理系统进展（NeurIPS），2023年。[23] Ruoyu Su、Alexander Bakhtin、Noman Ahmad、Matteo Esposito、Valentina Lenarduzzi 和 Davide Taibi。评估大语言模型检测架构决策违规的能力。arXiv预印本 arXiv:2602.07609，2026年。[24] Nalin Wadhwa、Jui Pradhan、Atharv Sonwane、Surya Prakash Sahu、Nagarajan Natarajan、Aditya Kanade、Suresh Parthasarathy 和 Sriram Rajamani。CORE：使用大语言模型解决代码质量问题。载于 ACM软件工程论文集（FSE），2024年。[25] Edwin B. Wilson。概率推断、继承定律与统计推断。《美国统计学会杂志》，22(158)：209–212，1927年。[26] Laura Wynter、Nirvik Sahoo 和 Paul Griffin。面向结构化假设空间推理的量子启发的轨迹增强证据选择。arXiv预印本 arXiv:2606.06941，2026年。[27] Chunqiu Steven Xia、Yinlin Deng、Soren Dunn 和 Lingming Zhang。Agentless：揭示基于大语言模型的软件工程智能体的奥秘。arXiv预印本 arXiv:2407.01489，2024年。[28] John Yang、Carlos E. Jimenez、Alexander Wettig、Kilian Lieret、Shunyu Yao、Karthik Narasimhan 和 Ofir Press。SWE-agent：智能体—计算机接口使自动化软件工程成为可能。载于神经信息处理系统进展（NeurIPS），2024年。[29] Ziqian Zhong、Aditi Raghunathan 和 Nicholas Carlini。ImpossibleBench：衡量大语言模型利用测试用例的倾向。arXiv预印本 arXiv:2510.20270，2025年。