Rust 代码的多智能体协同方法级重构

Yijian Zheng、Yi Zhang、Juyuan Zhang、Di Cui* 西安电子科技大学计算机科学与技术学院，中国西安 710071 cuidi@xidian.edu.cn；25031212388@stu.xidian.edu.cn

arXiv:2611.29321v1 [cs.SE] 22 Nov 2026

摘要—**随着 Rust 代码库不断演进，开发者会通过重构代码来减少重复、简化复杂方法，并将相关功能集中在一起。开发者可以手动进行这些更改，也可以使用集成开发环境提供的辅助功能执行受支持的操作，然后通过编译器诊断信息和测试检查生成的代码。Rust 中的方法级重构必须在保留可观察行为的同时，妥善处理所有权、借用和生命周期。我们提出了一种基于大语言模型（LLM）的工作流，用于提取、内联和移动 Rust 方法。该工作流结合了候选生成、可配置的专家审查和基于容器的验证。在来自三个开源 Rust 项目的六项任务中，三种端到端配置均在首次尝试时通过了所有任务检查，分别使用了 6、13 和 31 次模型调用。为单独考察审查的效果，我们随后向四种配置提供了相同的六个参考补丁和六个刻意改变行为的变体。指定专家在容器验证前拒绝了全部六个行为变更补丁，并让全部六个参考补丁进入验证；与直接验证相比，Docker 调用次数从 24 次降至 12 次。固定专家组也提前发现了全部六个故障，但因未经测量的性能问题拒绝了两个参考补丁。在这组工作负载中，专家审查减少了执行工作量，但其对总耗时的影响取决于审查开销以及哪些补丁进入验证。**

索引词—**Rust，软件重构，大语言模型，多智能体系统，自动化代码审查**

I. 引言

随着 Rust 库不断演进，原本只承担单一职责的例程可能逐渐承担更多职责，相关逻辑也可能分散到多个调用点。例如，一个最初用于遍历集合的方法，后来可能还负责输入验证和结果构造。维护者若要修改其中一项职责，就必须追踪周围的逻辑，以确定此次修改会影响哪些代码。重构可以明确这些职责，让后续修改更容易定位 [1]。方法级重构会在保持可观察行为不变的同时重新组织代码。提取代码片段会将其移入新的辅助函数，并用对该函数的调用替换原片段。在 Rust 中，如果辅助函数返回借用引用，其签名可能需要指定该引用借用了哪个输入的内容。内联则会用函数或方法调用所执行的计算替换该调用。改写必须保留参数的求值方式，以及临时值的销毁时机。将方法移至其他模块或 crate，可能会改变它能够访问的条目，以及调用方访问它的方式。即使开发者只指定了预期的结构性变更，工具也必须考虑这些要求。

先前研究已针对方法提取解决了其中若干挑战。REM 利用编译器信息修复提取所引入的控制流、所有权和生命周期问题 [2]。REM-V 和 REM2.0 探讨了如何为受支持的提取操作建立经机器验证的等价性 [3], [4]。多智能体系统也已用于规划、执行和验证 Java 中的重构 [5]。当不同配置生成不同补丁时，结果上的差异可能既源于提案，也源于对提案的审查。本研究探讨：在候选补丁和下游验证程序固定的情况下，专家审查能带来什么贡献。

本文实现了一种基于大语言模型的工作流，用于在 Rust 中进行方法提取、内联和移动。该工作流会将生成的或给定的候选项提交给基于容器的验证，并可选择在此之前进行专家审查。我们比较了直接验证、单个指定专家、自适应专家选择以及固定专家组。我们的主要贡献是一项配对评估：向所有四种配置提供完全相同的候选补丁，并衡量错误补丁何时被拒绝、参考补丁是否在审查期间被拒绝，以及每种配置执行了多少工作。

我们还在来自三个开源 Rust 项目的六项任务上进行了端到端实验，以补充配对评估。全部 18 次生成运行均在首次尝试时通过任务检查，但不同配置的模型调用成本各不相同。在 48 次运行的配对评估中，指定专家在容器验证前拒绝了全部六个改变行为的补丁，并让全部六个参考补丁继续通过。固定专家组也提前拒绝了全部六个错误补丁，并拒绝了两个参考补丁，而不是基于未经测量的性能问题。这些结果量化了提前拒绝、审查开销与丢弃满足任务检查的补丁所带来风险之间的权衡。

II. 背景与相关工作

A. *Rust* 中的方法级重构

我们研究了 Fowler 重构目录 [1] 中的三种操作。在我们的 Rust 任务中，我们将相应操作称为**提取方法**、**内联方法**和**移动方法**。

a) **提取方法**：提取方法会将选定的代码片段替换为对新辅助函数的调用。代码片段读取的值可能会成为参数，而之后所需的值

可能会成为返回值。如果辅助函数返回借用集合中的一个引用，其签名必须保留该引用与集合之间的关系。提取方法还需要保留跨越代码片段边界的控制流，例如从外层函数提前返回 [2]。

b) 内联方法： 内联会在调用点用被调用方的计算过程替换调用。如果某个参数具有副作用，而被调用方两次使用对应的形参，那么直接替换会导致该参数被计算两次。局部绑定可以保留原本只计算一次的行为。析构作用域的变化也可能改变值被销毁的时机，例如锁守卫释放其锁的时机 [6]。rust-analyzer 语言服务器为函数提取和调用内联提供代码操作 [7]。

c) 移动方法： 移动方法会重新定位其实现，并更新调用方访问该方法的方式。跨模块移动可能改变对私有字段和辅助函数的访问。跨 crate 移动还可能要求更改目标方法的可见性。所得实现必须保留相关副作用、返回值以及与调用方之间的契约。

B. Rust 中的自动化与经验证重构

REM 修复方法提取所引入的控制流、所有权和生命周期问题 [2]。REM-V 在此基础上，使用 Coq 对受支持的方法提取进行观察等价推理 [3]。REM2.0 在这一系列工作的基础上，进一步引入基于 rust-analyzer 的提取工具链，以及针对受支持 Rust 子集的可选等价性证明流程 [4]。在我们的研究中，特定任务的行为测试和结构检查为这三种重构操作提供验收标准。

C. 大语言模型辅助重构和补丁审查

EM-Assist 将模型建议、静态分析和 IntelliJ 的 Java 方法提取重构引擎相结合。模型提出候选片段，静态分析会筛选并完善这些建议，而集成开发环境则应用选定的转换 [8]。CodeReviewer 通过对代码变更和审查进行预训练，支持变更质量评估和评审评论生成 [9]。

RefAgent 协调专用智能体，负责规划、应用、测试和完善重构。其评估在开源 Java 项目上比较了多智能体和单智能体方法 [5]。我们的配对实验聚焦于审查 Rust 重构工作流的阶段。为每种配置提供相同的补丁，便可衡量哪些故障在执行前被捕获、哪些参考补丁在审查期间被拒绝，以及审查如何改变验证器执行的工作。

III. 方法

我们的系统以 Cargo 项目、自然语言目标和一组可编辑的源文件为输入，对 Rust 代码进行重构。它支持方法提取、内联和移动操作，

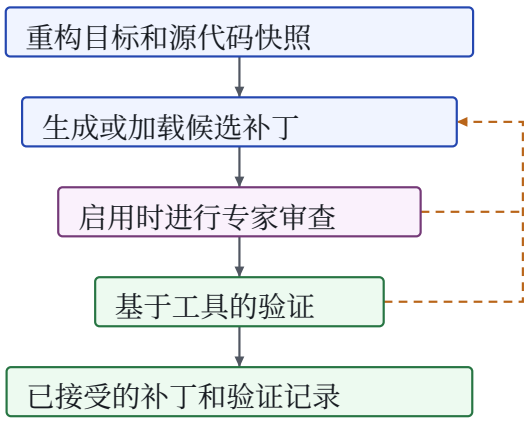


图 1. Rust 重构工作流。实线箭头表示通向接受的路径。虚线箭头表示在修复预算允许时，审查拒绝或验证失败如何反馈到修订后的候选项。

并将这些操作纳入统一的工作流程。图1展示了编排器如何获取候选项、选择性地将其提交给专员审查，并根据原始程序进行验证。失败的候选项可以利用拒绝它的阶段提供的证据进行修订。被接受的候选项将以统一差异的形式导出，并附带其验证记录。

A. 任务背景与补丁表示

编排器会捕获可编辑文件的原始内容，并向大语言模型提供相关源代码和任务目标。对于已配置的目标符号，它还可以包含 Tree-sitter 抽象语法树以及 Rust 编译器的中级表示（MIR）。这些表示为目标代码的结构和控制流提供了额外上下文。

候选项由一系列编辑组成。每项编辑都指定一个相对于项目的路径、一个原始片段和一个替换内容。在暂存候选项之前，编排器会检查每条路径是否属于可编辑集合、每个原始片段在应用时是否恰好匹配一个位置，以及源文件是否仍与捕获的快照一致。随后，它会将这些编辑应用到项目的临时副本中，并生成统一差异。给定候选项采用相同的表示方式，因此可以根据不同的审查策略评估同一补丁。

B. 智能体配置

这些配置使用相同的底层模型。它们在分配给每次模型调用的职责以及这些调用在工作流中的执行位置上有所不同。专员角色涵盖所有权和生命周期、并发以及性能。

- **单智能体**。 提议者生成初始候选项，并直接进入基于工具的验证。如果提供了候选项，编排器会将其直接发送进行验证。
- **指定专家**。指定的审查者会检查生成的或给定的候选项，并给出接受或拒绝的决定及其理由。

拒绝会阻止候选项进入执行阶段。

- **自适应协作**。 路由规则会根据任务目标、已更改的代码和验证设置选择专家。例如，涉及同步的更改可能会触发并发审查。选定的专家依次审查同一个候选项。第一次拒绝会终止审查。如果没有规则选择专家，候选项将直接进入验证。

- **固定团队**。 在端到端生成过程中，规划器会将子任务分配给专家，专家基于同一个源代码快照开展工作。编排器会合并互不重叠的编辑，并在拟议编辑发生重叠时请求协调。生成的候选项随后会接受审查和验证。在配对审查实验期间，固定团队会作为审查小组审查所提供的补丁，不进行规划或补丁生成。

C. 审查证据与调度

编排器会区分三个依赖不同证据的决策。第一个决策是确定候选项能否基于源代码快照具体生成。编辑会标识一个文件和一个源代码片段，且只有当该片段在应用点唯一匹配时，替换才会被定义。这条规则对于小型编辑十分重要，例如替换辅助函数名称：同一个名称可能会出现在文件中的多个调用点。第二个决策是确定审查者是否识别出值得阻止流程的问题。第三个决策是确定暂存候选项是否满足已配置检查。启用审查后，审查者的批准会将候选项推进至第三个决策。审查者不会决定下游测试的结果。

自适应调度将目标、变更片段和验证设置作为路由输入。其规则可以识别与借用、可见性、同步或性能敏感路径上的工作相关的信号。在支持解析的情况下，路由器可以避免仅仅因为注释或字面量中出现某个关键词就选择专员。分配结果会说明选择审查者的原因以及需要检查的约束。审查在首次拒绝时停止。这种设计可以节省后续审查调用和执行工作量。这也意味着，如果补丁中没有路由规则涵盖的信号，本地语义错误可能会到达验证器。

本文中固定团队的两种用法出现在工作流的不同阶段。对于端到端生成，规划器会分配子任务，专家可以贡献编辑内容。各方提案会在验证前合并；若编辑内容发生重叠，则会调用协调机制。对于提供的补丁，团队则是一个按固定顺序审查未修改候选项的评审小组。配对审计会对该评审小组进行测试，并衡量其审查决策。为拆分或协调工作而进行的模型调用，属于端到端生成实验。

D. 基于工具的验证

对于进入执行阶段的候选项，编排器会在禁用网络的 Docker 容器中，分别对原始版本和候选版本运行配置的 Cargo 命令。原始版本的运行用于确认配置的检查在基线上能够通过。在两个主要实验中，执行检查是针对特定任务的 Cargo 测试目标。评估框架还会检查差异，确认其中包含任务所要求的结构性变更。

启用后，系统支持额外检查。差分检查会比较从两个版本收集的观测结果；性能门槛则可以拒绝超过指定阈值的测量性能回退。可选的 Z3 检查会针对受支持的源代码模式验证预定义性质。例如，它可以检查原始版本和候选版本对于建模长度和阈值是否选择相同的分支。在智能体辅助模式下，模型会从预定义目录中选择一项性质。验证器会提取相关表达式并调用求解器。求解器结果仅适用于所选性质及其建模输入。主要实验使用第 IV 节中描述的任务专用测试和结构检查。

E. 修复与可追溯性

当审查或验证拒绝某个候选项时，编排器可以将先前的提案和失败证据传递给模型调用进行修复。在自适应模式下，编排器可以将修复任务交给提出问题的专员，或与验证失败相关的专员。如果无法分配专员，则由提议者负责修复。

每个修订后的候选项都会根据同一份源代码快照进行评估。当候选项通过已配置检查、修复预算耗尽或基线失败时，循环结束。运行记录会将每次修订与上一次尝试关联起来，并保存审查决策和验证结果。这些记录展示了补丁在哪些情况下被拒绝，以及请求修订时有哪些证据可用。

F. 示例演示：失败后的修复 *Check*

petgraph 反向迭代任务展示了系统如何利用验证反馈修订候选项。该任务是在单个调用点内联 edge_index，位置为 EdgeIndices::next_back。此转换必须保持每次调用所消耗的元素数量不变；当调用方交替从前端和后端推进时，还必须保持剩余元素的顺序不变。

我们使用一个给定候选项重放了该任务，该候选项将所请求的内联与对迭代器推进的修改结合在一起。如图 2 所示，该候选项正确替换了传递给 map 的辅助函数，同时还将 next_back() 替换为 nth_back(1)。后者会额外跳过一个元素。在四边测试夹具上，第一次反向调用返回了索引 2，而测试预期的索引为 3。该候选项成功编译，并执行到了行为断言。

修复调用收到先前的提案和验证失败的输出。它恢复了 next_back()，并保留了

（a）原始调用点

```
自身.r.next_back().map(edge_索引)
```

（b）给定候选项

```
自身.r.nth_back(1).map(边索引::new)
```

（c）修订后的候选项

```
自身.r.下一个_返回().map(边索引::new)
```

图 2. petgraph 内联任务中由反馈驱动的修复。提供的候选项改变了迭代器的推进方式。修订后的版本恢复了 next_back()，同时保留了内联构造函数。

调用点处的 EdgeIndex::new。修订后的候选项通过了配置的行为测试，包括对第一个反向结果、剩余长度、交错的正向和反向迭代以及最终耗尽的检查。它也通过了针对所请求内联操作的结构化检查。

这次重放从一个已知有缺陷的补丁开始，并在一次修订后完成。它展示了从观察到故障，到生成新补丁并进行新一轮验证运行的过程。端到端生成实验和配对审查实验分别报告各自的结果。

IV. 实验设计

两项主要实验使用相同的六个 Rust 重构任务。端到端实验比较了从候选生成到验收的三种完整工作流。配对审查实验将相同的补丁交给四种策略，并对进入执行阶段的候选项采用相同的下游验证。该实验衡量审查如何改变拒绝阶段，以及处理所提供补丁的成本。

第 III-F 节中的修复示例追踪了由一次未通过的行为检查所促成的修订。第 IV-I 节中的补充评估考察了在另外四项任务上进行的后续工作流修订。该修订同时更改了生成指令和基于语法的预检。因此，补充结果描述的是这项综合修订。

A. 研究问题

- **RQ1（端到端重构）**。在六项任务中，三种生成配置在观察到的任务验收、模型调用次数和经过时间方面有何差异?

- **RQ2（专家审查）**。在候选补丁完全相同的情况下，每种审查策略有多大比例会在执行前拒绝改变行为的补丁，或停止会通过任务检查的参考补丁？这些策略在模型调用次数、Docker 调用次数和经过时间方面有何差异?

B. 任务和验证标准

我们从三个开源 Rust 项目的固定快照中选取了六项任务，每种重构操作各两项（表 I）。上游修订版本分别为 petgraph 的 d6489f248fb0、rustc-hash 的 f16da569fcb8，以及 crossbeam 的 a28f40fdddb7。petgraph 的提取任务改编自 REM 工件 [2]。rustc-hash 中的批量循环内联任务，以及 crossbeam 中的两项跨 crate 移动任务，使用了引入待重构辅助函数或包装器的测试夹具。每项任务都明确说明了所需的转换、可编辑文件以及必须保留的行为。

表 I 评估中的重构任务和检查的行为。

操作	项目	已检查行为
提取	petgraph	迭代器推进和访问借用的边权重
提取	rustc-hash	短输入边界处的哈希输出值
内联	petgraph	反向迭代与正向迭代
内联	rustc-hash	字节解码和读取顺序批量循环中
Move	crossbeam	退避状态转换和返回值返回值
Move	crossbeam	保留被拒绝的负载荷和队列内容（当队列已满

对于每项任务，我们都会准备一个参考补丁，并通过修改该补丁来生成一个会改变行为的变体。在评估之前，我们会验证原始程序和参考补丁都能通过该任务对应的 Cargo 测试目标。我们还会验证会改变行为的变体能够编译，并且会在该测试目标中导致一项行为断言失败。因此，配对实验中的每个错误变体都有一个已知的下游测试能够检测到它。

评估框架会针对差异执行任务专属检查，以确认被接受的补丁实现了所要求的结构性变更。例如，提取操作必须引入所要求的辅助函数，并从原始方法中调用它。在这些实验中，是否接受由任务的行为检查和结构检查共同决定。

C. 成对补丁的构造

我们通过修改同一任务的参考补丁来构造每个会改变行为的候选项。因此，两个候选项都包含所要求类型的重构，并且针对相同的可编辑文件。额外的修改会改变该重构中的某项操作，例如传递给迭代器的索引、边界比较，或队列已满时采取的操作。这种配对使审查者可能遇到结构上看似合理、但其错误取决于变更语义的补丁。

表 II 将每项修改与用于对其进行资格认定的证据关联起来。测试会检查返回值和调用方可见的状态，包括迭代器的消耗情况和队列内容。对于借用边提取，测试还会检查返回的权重是否指向原始图条目。对于哈希任务，测试会将选定输入产生的哈希与独立计算的预期值进行比较。

表 II 六个植入故障的构造与验证。每个变异体均由同一任务的参考补丁派生。

Task	植入变更	行为测试的证据
借用边提取	将 $n+1$ 传递给用于该测试索引。	调用返回边 (3, 4)，而不是预期的 (2, 3)。该测试还会检查引用标识和后续的迭代器操作。
短输入哈希提取	更改 8 字节分支条件，从长度 >= 8到长度 > 8。	对于每个输入，都会将哈希与预期结果进行比较长度范围为 0 到 16。
反向边内联	使用 nth_back(1) 进行反向推进。进。	第一次调用返回索引 2，而预期应为索引 3。测试还会检查长度以及交错推进的顺序。
批量哈希内联	计算第二个字的值时，再次解码第一个字计算第二个单词的值。	批量输入哈希会与计算出的预期值进行比对按源代码中的单词顺序读取。
退避方法移动	获取待处理的退避步骤后返回 false。	该测试将包装器的返回值和完成状态与底层的退避行为进行比较。
队列方法移动	将满队列时的拒绝替换为覆盖，并报告成功。	测试期望获得原始的被拒绝负载，并检查现有项目是否保持队列顺序。

该资格认定程序为每个成对补丁确立了已知结果。原始版本和参考版本均通过测试，说明该测试在转换前后都能接受预期行为。要求变异体能够编译且未通过行为断言，可排除仅由语法或类型错误导致的失败。由此得到的集合支持分析已知故障何时会被检测到。对自然生成补丁中故障的覆盖情况仍是一个独立的实证问题。

D. 端到端重构（*RQ1*）

我们在六项任务中的每一项上分别运行一次单智能体、自适应和固定团队配置，共计 18 次运行。对于给定任务，三种配置接收相同的任务目标和可编辑源代码快照。初始提案后，每种配置都有一次修复尝试的额度。在本实验中，固定团队参与候选生成。

我们记录首次尝试接受情况、任何修复后的最终接受情况、模型调用总次数以及端到端墙钟时间。本实验比较完整的工作流，包括候选生成和审查方面的任何差异。

E. 配对补丁审查（*RQ2*）

我们为每种配置提供相同的六个参考补丁和六个行为变更变体，共计 48 次运行。四种配置分别采用直接验证、单个指定专家、自适应专家选择或固定专家组。固定评审组审查所提供的补丁，但不运行规划或生成阶段。测试框架使用补丁哈希值确认候选输入相同，将参考标签置于审查者输入之外，并在不同补丁间轮换配置顺序。禁用修复，以确保每个结果都反映对所提供补丁的处置结果。

当审查在 Docker 验证前停止一个会改变行为的补丁时，我们将其计为提前拒绝。当审查停止一个满足任务检查的参考补丁时，我们将其计为行为参考标签下的误拒绝。我们还会记录最终处置结果、模型调用次数、Docker 调用次数以及端到端墙钟时间。基线的每次执行或候选命令在 Docker 中执行均计为一次调用。经过时间包括候选项所经历各阶段的审查、暂存和执行时间。

由于行为变更变体使用相同的下游测试进行资格验证，配对比较衡量的是审查如何改变拒绝发生的阶段及其成本。六个参考补丁和六个错误变体构成了一个有意均衡的工作负载，用于比较这些策略。

F. 执行环境和制品

两项实验均使用带有模型标识符 deepseek-flash 的 DeepSeek API。配置的 Cargo 测试目标在使用 rust:1.85 镜像的本地 Linux 容器中离线运行。主要实验使用任务专用测试和结构检查。可选的 Z3 和性能门槛不在这些实验的范围内。

evaluation/dataset-pilot/ 目录包含任务定义、测试夹具和行为测试。reports/ 中的独立 JSON 记录记载了任务资格验证、端到端重构以及配对补丁审查。配对审查记录包含输入补丁哈希值，以及每次运行的审查和验证结果。

G. 配对和结果解释

专家审查的比较单位是某项审查策略下提供的补丁。对于每项任务，参考补丁实现请求的重构。对其实现进行一项范围狭窄的修改即可生成错误变体，同时保留重构的可见结构。例如，在插入迭代器错误后，提取操作仍会引入请求的辅助函数。因此，审查者不能仅通过检查预期的辅助函数或调用点是否存在来推断行为正确性。资格验证结果表明，下游测试在原始程序和参考程序上通过，而在错误变体上失败。该选择程序使审查阶段比较具有可解释性，并将其限定在任务测试已经能够检测到的故障上。

审查测试框架保留用于评分的补丁标签，并针对给定输入向每项策略提供相同的补丁内容。每个进入执行阶段的候选项都必须接受相同的

表 III 不同重构类型的验收证据示例。结构检查会检查候选差异。

Type	结构性证据	行为证据
提取	引入辅助函数，并从原位置调用它	保留返回的是哪条边 这一信息 并保留 its 从 图中借用的权重。
内联	移除 the 所选通过旧的映射 辅助函数，并构造该位置的边索引。	返回最后一条剩余的边，并保留后续的迭代 顺序。
Move	移除旧的包装 方法，并委托给 被迁移的实现 。	在满队列时退回被拒绝的负 满队列上的负载 并保留队列中的元素。

任务测试。审计记录将一份在审查阶段被拦截的参考补丁标记为行为标签下的误拒绝。该标签表明补丁通过了可用的任务检查，但无法解决这些检查范围之外的行为问题，包括性能。模型调用次数衡量的是请求数量，而不是令牌消耗。一次 Docker 调用对应一条记录在案的基线或候选命令，而墙钟时间则记录候选项所经历的完整路径。

H. 结构约束与行为检查

结构检查与测试目标的作用不同。若补丁保留原始方法，并添加一个未使用的辅助函数，就可能保留所有经过测试的观测结果。但它仍无法完成提取任务。反之，补丁可以引入所需的辅助函数并替换原始调用，同时改变返回给调用方的值。成对故障变体就是后一种情况的示例。因此，评估既会检查所要求的转换，也会检查该转换必须保留的观测结果。

表 III 展示了这两类证据如何相辅相成。结构验收因任务而异。在反向迭代器示例中，重构仅应用于一个调用点，其他调用方可以继续使用现有的辅助函数。对于移动操作，仅搜索旧函数名并不足够，因为目标位置可能保留相同的名称。检查可以分别确认旧位置已移除、新位置已引入，以及通过未变更的公共入口点进行委托。

行为输入旨在区分注入错误会改变的行为。在借用边任务中，不同的端点会暴露额外的推进操作。指针比较会检查返回边的权重是否为图中存储的值。在满队列任务中，不同的负载使测试能够区分被拒绝的插入和覆盖。这些断言支持我们对所选输入和故障机制得出的结论，但并未刻画原始实现或候选项能够执行的每一种操作。

表 IV 各配置下六项任务的端到端结果。模型调用次数为各项任务的总和。

配置	第一遍	调用次数	中位数（秒）
单智能体	6/6	6	39.9
自适应	6/6	13	58.5
固定团队	6/6	31	71.0

I. 补充条件编译评估

我们还考察了来自相同 rustc-hash 和 crossbeam 修订版本的另外四项任务。这些任务涉及两项内联操作和两项移动操作。一项内联操作在 FxHasher::write_u128 中替换了四次辅助函数调用，同时保留哈希更新的顺序以及彼此独立的指针宽度保护条件。另一项则在 ArrayQueue 的消费型迭代器中的单个位置内联了队列容量查询。两项移动操作分别重新安置了公开的 FxHasher::with_seed 构造函数和 SegQueue 的 Default 实现。前者保留了公开的 const 签名，后者则保留了其分配及原子指针配置条件。

我们在这四项任务上比较了两个连续的工作流版本。每个版本针对每项任务和每种生成配置运行两次，因此每个版本共运行 24 次。这些后续运行使用 deepseek-flash、一次修复的预算、offlineCargo 验证，以及通用的特定任务结构验收规则。修订后的版本增加了一条关于保留语句级配置属性的生成指令，并加入了基于语法的预检；当违反该约束时，预检会提供针对性诊断信息。比较将这两项变更视为一次工作流修订。实验还分别对原始程序和修复后的 write_u128 示例进行了 32 位检查。该检查旨在执行一个普通主机目标运行不会执行的目标特定分支。

V. 结果

A. RQ1：端到端重构

三种配置均在首次尝试时通过了全部六项任务（表 IV）。每种配置都通过了两项提取任务、两项内联任务和两项移动任务。18 次运行中没有一次使用可用的修复尝试。

单智能体配置在六项任务中共进行了六次模型调用，而自适应协作进行了 13 次，固定团队则进行了 31 次。单智能体配置的端到端时间中位数也最低，为 39.9 秒。自适应协作和固定团队的中位数分别为 58.5 秒和 71.0 秒。因此，这三种工作流的观测成本各不相同，但在任务检查中的通过数量相同。

B. RQ2：配对补丁审查

指定专家和固定评审组在 Docker 验证前均拒绝了六个改变行为的补丁中的全部六个（表 V）。自适应协作在审查阶段拒绝了其中五个。

剩余的补丁改变了 petgraph 中的反向迭代，而任务专用测试发现了该故障。在直接验证配置中，测试发现了全部六个故障，这与任务资格认定流程的预期一致。

指定专家与自适应协作使全部六个参考补丁得以继续。固定评审组拒绝了两个已通过任务检查的参考补丁。在这两种情况下，性能专家都对新引入或重新定位的辅助函数的编译器内联提出了疑虑。一个补丁提取了 rustc-hash 中的短输入哈希逻辑。另一个补丁将 crossbeam 中的退避方法移到了其他 crate。该实验未对这些潜在的性能退化进行基准测试，因此，所报告的误拒绝是相对于行为参考标签而言的。

在包含 12 个补丁的平衡数据集上，指定专家将 Docker 调用次数从 24 次减少到 12 次，并记录了最低的总墙钟时间。相较于直接验证，对于一个参考补丁，它平均增加了 5.6 秒；对于一个行为变更补丁，它平均节省了 8.8 秒。固定评审组发起的 Docker 调用次数最少，但总体耗时最长。其调用次数既反映了六次早期故障拒绝，也反映了它在执行前停止的两个参考补丁。

图 3 展示了 petgraph 提取任务。参考补丁将选定的代码移动到私有辅助函数中，并保留了返回边权重的生命周期。种子变体更改了辅助函数中的迭代器推进方式。指定专家识别出这一更改，并在 2.5 s 内拒绝了该补丁，且未启动 Docker 验证。行为断言失败时，直接验证在 13.5s 后拒绝了同一补丁。这两个时间都是这两次运行的端到端测量结果。

C. 按任务划分的审查结果

表VI 显示了每个行为变更变体被拒绝的阶段。指定专家和固定评审组在审查阶段发现了全部六个变体。自适应协作发现了其中五个。其路由规则未为反向迭代补丁选择任何专家，因此候选项直接进入任务测试。

漏检的补丁将 next_back() 替换为 nth_back(1)，这会从迭代器末尾额外消耗一个元素。第一个断言预期边索引为 3，实际得到边索引 2。指定专家和固定评审组在执行前识别出了相同的错误。在此案例中，差异出现在路由阶段：该补丁未经专家评估便进入验证。

关于参考补丁的两项分歧发生在固定评审组中。对于短输入哈希提取，其性能审查者提出了一种可能性：新的辅助函数在下游构建中仍会保持为非内联。对于退避移动，审查者也提出了关于新跨 crate 调用成本的类似担忧。在审计之前，这两个参考补丁都已通过行为检查。其他策略允许它们进入验证阶段。因此，评审组决策体现了用于阻止候选项的证据存在差异。具体测试可以证明迭代器在某个输入上消耗了错误的项目。关于内联的论证则指出需要进行一项测量，同时无法确定运行时效应是否存在及其幅度。

均衡工作负载也解释了执行开销与经过时间之间的差异。指定专家通过提前停止六个有故障的补丁，减少了十二次 Docker 调用。固定评审组减少了十六次调用：其中十二次来自有故障的补丁，另四次来自两个被停止的参考补丁。后一项减少的代价是对参考补丁的接受率较低。此外，在记录的审查阶段中，指定专家、自适应协作和固定评审组在各自的 12 次运行中分别耗时 62.0s、82.6 s 和 176.7 s。这些总时间包含了各次模型调用的差异。因此，固定评审组虽然减少了容器操作，却仍比直接验证耗费更多墙钟时间。

D. 按补丁对计时

汇总结果涵盖了不同的执行成本和审查路径。表 VII 报告了每对任务的两个成员，以便呈现工作负载的影响。例如，对于种子化短输入哈希故障，直接路径耗时 6.0 s，专家路径耗时 4.5 s。相应的参考补丁直接验证耗时 5.9 s，由专家审查则耗时 19.0 s。对于队列故障，直接路径耗时 20.1 s，专家路径耗时 3.7 s。因此，同一审查策略在不同任务中的计时影响各不相同，即便其决策在所有情况下都与任务标签一致。

审查说明也有助于理解捕获到了哪些问题。对于短输入哈希辅助函数，指定专家指出，变更边界从 `>= 8` 到 `> 8`。长度为八时，有缺陷的辅助函数走的是四字节路径，而不是八字节路径。在批量哈希情形中，游标前进了两次，但第二次解码使用的切片边界却由第一次读取决定。尽管读取次数和游标移动仍与原始结构相似，审查者还是指出了错误操作数。这两个故障需要不同的检查：第一种情形需要边界覆盖，第二种情形则需要可区分输入字。

跨 crate 移动任务表明，仅检查调用方是否委托给新位置，只是验收标准的一部分。有缺陷的退避方法委托给了迁移后的代码，却在推进状态后返回了错误的布尔值。有缺陷的队列方法委托给一项操作，该操作可能会在队列已满时覆盖最旧元素，并报告成功。指定专家指出了这两处契约违规。测试通过比较包装器的返回值和完成状态来检验相应路径，并检查满队列是否返回提交的负载，同时保留其中已有的元素。这些检查在移动后方法的调用方处定位影响，因为编辑带来的行为后果可在那里观察到。

E. 条件编译的补充结果

表VIII报告了第 IV-I 节中的两个工作流版本。在原始工作流下，24 次运行中有 18 次在首次尝试时通过，另外 6 次在一次修复后通过。所有需要修复的运行都发生在 write_u128 上，其中

表 V 配对补丁审查结果。每种配置均收到六个参考补丁和六个行为变更变体。误拒绝计入行为参考标签。模型调用、Docker 调用和墙钟时间按每种配置的 12 次运行累计。

配置	提前拒绝	误拒绝	模型调用	Docker 调用	总时间（秒）
单智能体（直接验证）	0/6	0/6	0	24	148.9
指定专家	6/6	0/6	12	12	129.5
自适应	5/6	0/6	12	14	163.1
固定评审组	6/6	2/6	24	8	225.0

（a）提取前

```
fn nth(&mutself, n: usize)      ->Option<Self::Item>{ // 选定的片段
self.inner.nth(n).map(          |(&(n1, n2),weight)|(n1,n2,weight)      ) }
```

（b）参考提取

```
1 fn nth(&mut self, n: usize) 2 ->Option<Self::Item>{ 3 self.hard_nth_边(n) 4 } 5 6 fn hard_nth_边(&mut self, n: usize) 7 ->Option<(N, N, &'a E)>{ 8 self.inner.nth(n).map(                |(&(a, b), 权重) |(a, b, 权重) 10 ) 11 }
```

（b）中的植入故障。替换。 nth(n) 替换为 .nth(n.saturating_add(1))。对于 nth(1)，测试期望边 (2, 3)，而 ile 修改后的辅助函数返回 (3, 4)。

图3. 来自 petgraph 的 AllEdges::nth 的提取方法示例。代码清单展示了相关方法，并省略了其所属的 impl 块。

错误变体按任务划分的拒绝阶段。表VI

Task	直接	专家	自适应	面板
借用边	T	R	R	R
短输入哈希	T	R	R	R
反向迭代器	T	R	T	R
批量哈希内联	T	R	R	R
退避方法移动	T	R	R	R
队列方法移动	T	R	R	R

R：审查时被拒绝。T：任务测试未通过。

每种配置下各有两次此类运行。在修订后的工作流中，全部 24 次运行均在首次尝试时通过。其他三项任务在每个版本的全部 18 次运行中也都在首次尝试时通过。因此，观察到的差异集中在执行语句级指针宽度条件的任务上。

write_u128 任务说明了额外检查为何有用。一条记录在案的错误候选方案直接将 cfg 属性附加到赋值语句上。稳定版编译器在编辑后的上下文中拒绝这种形式。反馈诊断信息后，模型得以将条件赋值包装在守卫代码块中，并保留原始状态更新的顺序。另有一项针对 i686-unknown-linux-gnu 目标的检查，运行了原始版本和修复后的版本。两者均通过了包测试，以及一项使用不同 32 位字来检查更新顺序的针对性断言。这项检查为该修复在一个 32 位目标上的有效性增加了执行证据。常规运行则在主机目标上比较工作流，并通过结构检查确认守卫代码块是否存在。

本次后续研究中的首轮计数仅用于描述。生成指令和预检均已更改，并且实验分批依次进行。因此，该比较描述的是修订后的工作流整体的观察行为。此前的配对审计回答的是另一个问题，其原始结果和分母均予以保留。

VI. 讨论与有效性威胁

A. 专家审查的成本权衡

三个端到端工作流均在首次尝试时通过了六项任务。在这些运行中，直接验证所需的模型调用次数最少，耗时中位数也最低。配对实验揭示了专家审查可能发挥作用的场景：对于存在可检测故障的候选项，审查者可以在补丁暂存和容器执行之前将其拦截。对于进入验证流程的参考补丁，审查会增加一次模型调用，而下游检查仍会继续运行。

这种权衡取决于收到的补丁。与直接验证相比，指定专家处理每个参考补丁平均增加 5.6 秒，而处理每个行为变更补丁平均节省 8.8 秒。设 p 为收到的补丁中存在与植入故障相当的故障的比例。如果实测平均成本保持不变，则每个补丁的预计时间差（以秒为单位）为

$$\Delta T(p) = 5.6(1-p) - 8.8p.$$

负值意味着从耗时角度看，优先采用专员审查。由此得出的盈亏平衡比例约为 39%。这是针对观测到的工作负载所做的示例性估算。模型延迟、测试运行时间或故障构成的变化都可能改变盈亏平衡点。

表 VII 给定候选项的端到端墙钟时间。每个单元格分别给出参考版本和故障变体的时间，单位为秒。加粗的参考时间表示在审查中遭到拒绝。

Task	直接验证	指定专家	自适应	固定评审组
借用边提取	18.0 / 13.5	18.2 / 2.5	20.4 / 3.3	33.7 / 8.6
短输入哈希提取	5.9 / 6.0	19.0 / 4.5	24.7 / 5.2	**38.4**/ 7.0
反向迭代器内联	12.3 / 12.0	15.4 / 2.8	12.2 / 13.3	28.6 / 3.5
批量哈希内联	5.7 / 5.8	14.1 / 3.6	24.4 / 4.1	31.5 / 2.7
退避方法移动	20.8 / 14.8	22.0 / 2.3	25.7 / 3.7	**22.9**/ 2.7
队列方法移动	14.0 / 20.1	21.5 / 3.7	21.8 / 4.3	39.0 / 6.5

表VIII 工作流修订前后的补充运行。每个配置在每个版本中运行八次。

配置	第一遍	调用次数	中位数（秒）
单智能体	$6 \rightarrow 8$	$10 \rightarrow 8$	$25.9 \rightarrow 24.3$
自适应	$6 \rightarrow 8$	$10 \rightarrow 8$	$29.3 \rightarrow 24.4$
固定团队	$6 \rightarrow 8$	$44 \rightarrow 40$	$59.4 \rightarrow 52.1$

B. 审查决策的证据要求

配对审计说明，拒绝依据会如何影响对拒绝的解读。在反向迭代案例中，审查者发现迭代器多推进了一次，而任务测试则体现了这一问题对返回边的影响。在两项针对参考补丁的固定面板拒绝中，审查者预见到编译器内联变更可能带来运行时成本。这些补丁已经通过行为检查，而审计并未测量所怀疑的性能回归。

对于审查策略而言，这些情况意味着不同的后续步骤。与任务行为要求存在具体差异时，可以作为提前拒绝的依据。性能问题则可以进一步确定代码路径和构建配置，以便进行针对性基准测试。随后，基准测试结果可以为接受决策提供依据。这一策略能否减少不必要的拒绝，仍有待未来评估。

C. 验证设计的影响

评估展现了反馈的两种不同用途。反向迭代示例在格式正确的 Rust 表达式通过编译后，使用了任务级断言。条件编译示例则使用编译器诊断信息，或针对编辑形式的结构化预检结果。两种情况下，修订补丁所需的证据各不相同。对于前者，断言指出了多消耗一个元素这一可观察效果。对于后者，检查指出了如何在保留目标特定分支的同时表达条件代码。有效的修复阶段可以在发送给模型的反馈中保留这一差异，然后依据相同的任务标准重新评估生成的补丁。

这些案例还说明了如何在结构化检查与专家决策之间进行选择。对于反复出现的转换约束，例如保留语句级配置守卫，可以将其编码并根据补丁进行验证。对于新引入调用可能带来的性能问题，则取决于构建和运行时环境，可以通过单独的测试进行验证，并采用有针对性的基准测试。对于反向迭代顺序这类语义要求，特定于任务的断言仍然是直接的证据来源。模型的审查可作为针对这些要求的更早决策点。

D. 泛化所需证据

本研究区分了后续评估需要回答的三个问题。第一个问题涉及审查决策的可重复性。可以通过重复相同的补丁–策略对，并记录拒绝阶段、模型调用和经过时间的变化来进行考察。由于多个策略共用同一个候选项，重复实验设计在比较二元决策和成本时都能保留配对关系。第二个问题涉及故障覆盖率。针对自然产生的重构补丁开展研究，需要独立的行为标签，包括明确标记测试和审查者证据都无法确定结果的情况。这样一组数据能够揭示审查在比本研究所用的经过刻意筛选的数据更广泛的故障范围内如何发挥作用。第三个问题涉及接受或拒绝模糊案例的成本。固定评审组中未测量的性能问题，为这项研究提供了具体候选案例。

上述成本计算将经过时间视为结果，并使用抽样补丁的平均值。在实践中，提前拒绝可能会将工作转移到后续修复或人工维护者身上。因此，部署评估应记录从初始审查决策到候选项达到可供采用的状态之间所需的工作量。本研究将修复预算设为零，从而在配对审计中单独考察首次决策。生成实验和修复示例分别提供了有关修复行为的观测结果。这些设定共同说明，为何首轮审查延迟和完整修复成本需要分别测量。

E. 效度威胁

a) 任务与故障选择：这六项任务旨在检验三个 Rust 项目中的特定要求，其中部分任务使用了添加到上游快照中的测试夹具。配对故障经过了刻意植入，并通过下游测试进行验证。这种设计有助于受控比较已知故障在何种情况下会被拒绝。自然出现的补丁可能引入不同的故障，其中一些故障可能无法被现有测试发现。平衡补丁集也无法估算生产环境中有故障补丁的占比。

b) 验收标准：任务验收依据行为测试和结构检查。通过的补丁满足该任务所采用的检查要求。因此，报告的误拒绝是相对于行为参考标签而言的。主要实验并未衡量全程序语义等价或运行时性能。

c) 可重复性与成本：在主要实验中，我们针对每个任务-配置对和补丁-配置对，使用一个模型标识符各运行了一次。观测到的数量和耗时无法量化重复模型调用或不同模型之间的差异。经过时间也取决于 API 延迟、构建缓存和本地执行环境。配对比较在不同补丁间轮换配置顺序，而计时结果仍仅适用于记录的运行。补充比较在四项任务上重复生成，并同时改变两个工作流组件。该比较未对固定补丁审查决策进行重复观测。

VII. 结论

我们提出了一种基于大语言模型的工作流，用于在 Rust 中提取、内联和移动方法。为衡量专家审查的贡献，我们向四种配置提供了相同的候选补丁，并对进入执行阶段的补丁采用相同的下游验证程序。

在端到端实验中，三种生成工作流均在首次尝试时通过了六项任务。单智能体工作流使用的模型调用次数最少。在配对审查实验中，指定专家在 Docker 验证前拒绝了全部六个会改变行为的补丁，允许全部六个参考补丁进入后续流程；与直接验证相比，Docker 调用次数从 24 次减少到 12 次。固定评审组进一步将调用次数降至 8 次，但其总经过时间最高，并以未测量的性能问题为由拒绝了两个参考补丁。

在所研究的任务中，专家审查改变了已知故障被发现的时机，以及进入基于执行的验证阶段的工作量。其实际价值取决于候选补丁中包含故障的频率、下游检查的成本，以及审查拦下满足这些检查要求的补丁的频率。未来工作应使用自然产生的补丁重复这项比较，并通过针对性基准测试评估审查期间提出的性能问题。

参考文献

[1] M. Fowler，《重构：改善既有代码的设计》，第 2 版。Addison-Wesley Professional，2018。[2] S. Thy、A. Costea、K. Gopinathan 和 I. Sergey，“《一生的冒险：Rust 的提取方法重构》”，《ACM 编程语言学报论文集》，第 7 卷，OOPSLA2 期，2023 年，第 245 号文章。[3] M. Britton、A. Potanin 和 S. Pak，“在 Rust 中验证提取方法重构”，载于《2025 年 ACM SIGPLAN 人类软件系统、编程、语言与应用国际会议（SPLASH Companion）附属论文集》，2025 年，第 58–59 页。[4] M. Britton、S. Pak 和 A. Potanin，“《Rust 中的重构与等价性》：以新颖方法扩展 REM 工具链，实现自动化等价性证明”，arXiv 预印本 arXiv:2601.19207，2026 年。 [在线]。链接：https://arxiv.org/abs/2601.19207

[5] K. Oueslati、M. Lamothe 和 F. Khomh，“RefAgent：一种用于自动软件重构的多智能体大语言模型框架”，载于《第 *48* 届 *IEEE/ACM* 国际软件工程会议（*ICSE*）论文集，2026 年。[6] “析构函数”，《Rust 参考》。 [在线]。链接：https://doc. rust-lang.org/reference/destructors.html [7] “辅助功能（代码操作）”，《rust-analyzer 用户手册》。 [在线]。链接：https://rust-analyzer.github.io/book/assists.html [8] D. Pomian、A. Bellur、M. Dilhara、Z. Kurbatova、E. Bogomolov、T. Bryksin 和 D. Dig，“新一代重构：结合大语言模型的洞见与集成开发环境的能力来提取方法”，载于《*2024* 年 *IEEE* 国际软件维护与演化会议（*ICSME*），2024 年，第 275–287 页。[9] Z. Li、S. Lu、D. Guo、N. Duan、S. Jannu、G. Jenks、D. Majumder、J. Green、A. Svyatkovskiy、S. Fu 和 N. Sundaresan，“通过大规模预训练自动化代码审查活动”，载于《第 *30* 届 *ACM* 欧洲软件工程联合会议暨软件工程基础研讨会（*ESEC/FSE*）论文集，2022 年，第 1035–1047 页。