Atomizer：基于大语言模型的协作式多智能体框架，用于意图驱动的提交解缠

Kangchen Zhu*
国防科技大学计算机科学与技术学院，中国长沙
zhukangchen18@nudt.edu.cn

Zhilian g Tian†
国防科技大学计算机科学与技术学院，中国长沙
tianzhiliang@nudt.edu.cn

Shangwen Wan g*†
国防科技大学计算机科学与技术学院，中国长沙
wangshang
wen13@nudt.edu.cn

Mingyue Leng
国防科技大学计算机科学与技术学院，中国长沙
lengmengyue23@nudt.edu.cn

Xiaoguang Mao*
国防科技大学计算机科学与技术学院，中国长沙
xgmao@nudt.edu.cn

arXiv:2601.01233v1 [cs.SE] 3 Jan 2026

摘要

包含多个无关关注点的复合提交在软件开发中十分常见，会严重阻碍程序理解与维护。现有的自动解缠方法，尤其是最先进的基于图聚类的方法，存在两项根本局限：（1）过度依赖结构信息，无法把握变更背后的关键**语义意图**；（2）采用“单次处理”的算法，缺乏**人工审查流程中不可或缺的反思与优化机制**。为应对这些挑战，我们提出了 Atomizer，这是一种用于复合提交解缠的新型协作式多智能体框架。为弥补语义缺失，Atomizer 采用**意图导向思维链（IO-CoT）**策略，促使大语言模型（LLM）结合代码的结构信息和语义信息，推断每项代码变更的意图。为克服“单次处理”分组的局限，我们使用两个智能体构建**分组器与审查者协作优化闭环**，通过反复优化分组，直至某个簇中的所有变更都具有相同的底层语义意图，以此模拟人工审查实践。在两个 C# 和 Java 基准数据集上进行的大量实验表明，Atomizer 的表现显著优于多个具有代表性的基线方法。平均而言，其表现比最先进的基于图的方法在 C# 数据集上高出 6.0% 以上，在 Java 数据集上高出 5.5%。在复杂提交上，这一优势尤为明显，Atomizer 的性能领先幅度扩大至 16% 以上。

CCS 概念

• **软件及其工程 → 软件维护工具**。

关键词

复合提交解缠、大语言模型、多智能体框架、思维链、软件维护

ACM 参考格式：
Kangchen Zhu、Zhiliang Tian、Shangwen Wang、Mingyue Leng 和 Xi-aoguang Mao。2026 年。Atomizer：一种基于大语言模型的协作式多智能体框架，用于意图驱动的提交解缠。载于 *2026 IEEE/ACM 第48届国际软件工程大会（ICSE '26），2026年4月12日至18日，巴西里约热内卢。* ACM，纽约，纽约州，美国，13页。https://doi.org/10.1145/xxxxxxx.xxxxxx

1 引言

在协作式软件开发中，开发人员通常会修改代码，并将变更提交到代码仓库。理想情况下，版本控制系统（VCS）中的提交应当是“原子的”，即每个提交只处理一个关注点（例如功能实现或缺陷修复），从而提升历史记录清晰度、代码理解能力和评审流程 [34, 40, 44]。然而，开发人员常常会创建“复合提交”，将多个互不相关的关注点打包到一个提交中。这类复合提交在实际代码仓库中的占比高达 11% 至 40% [9, 13, 14, 31, 41]。

复合提交会对软件维护和漏洞管理造成重大风险。例如，来自 Pulumi 项目的一个真实复合提交（提交 ID：3988）[1] 涉及 24 个文件和 131 个差异区域，包含缺陷修复、新功能和支持更新。这种复杂性会导致两个主要问题：**（1）自动化安全工具失效**：漏洞检测模型可能会错误标注该提交，从而错误地将互不相关的变更联系起来，污染其训练数据。**（2）人工维护难题**：回退功能 A 可能会意外地重新引入漏洞 B，使维护工作变得困难且容易出错。这些问题严重阻碍代码审查、增加代码仓库分析的复杂性，并降低自动化缺陷预测和

*Kangchen Zhu、Shangwen Wang 和 Xiaoguang Mao 也隶属于复杂关键软件环境国家重点实验室。
†Zhiliang Tian 和 Shangwen Wang 是通讯作者。

特此授权免费制作本作品的全部或部分数字版或纸质版副本，用于个人或课堂使用，但前提是不得为营利或商业利益而制作或分发副本，且副本必须附有本声明以及首页上的完整引文。本作品中第三方组件的版权必须得到尊重。如需其他用途，请联系所有者/作者。*ICSE '26*，巴西里约热内卢© 2026 版权所有归所有者/作者所有。ACM ISBN
xxx-x-xxxx-2025-3/26/04https://doi.org/10.1145/xxxxxxx.xxxxxx

[1] https://github.com/pulumi/pulumi/commit/398878de31e42f7ec4485eab1c665f9aeea4c98a

漏洞检测 [3, 20, 24]。因此，研究界正积极开发自动化方法，将复合提交分解为原子单元。

现有的自动解缠方法经历了三种主要范式的发展。早期的基于启发式规则的方法依赖人工指定的启发式规则来推断变更之间的关系 [3, 14, 20, 44]。随后，基于特征的方法通过定义可机器学习特征对此进行了改进 [7, 47]。近年来，基于图聚类的方法日益受到重视：它们将代码变更表示为图，并应用聚类算法，根据结构依赖对其进行分组 [23, 32, 39]。尽管这些基于图聚类的方法前景可期，但仍存在两个根本性局限。

首先，这些方法过度依赖结构信息，却未能充分利用语义。它们依据代码依赖对变更进行分组，却往往忽略开发者的**意图**，即变更背后的具体目标和动机。这些方法可能会错误地将结构上相关、语义上无关的变更归为一组。反过来，对于意图相同但缺少显式结构依赖的变更，它们也可能无法将其归为一组 [9]。尽管开发者意图可能会隐含地反映在提交信息中，但仅依赖这些信息不足以可靠地捕捉语义。先前研究 [1, 4, 37]表明，提交信息往往存在噪声、不完整，甚至完全缺失 [31, 44]，，因此无法作为语义理解的可靠来源。

其次，这些方法通常采用"单次处理"流程，缺乏优化或反馈机制。尽管现有方法（如基于GNN的方法）可能会进行内部迭代，但最终仍缺少对其输出进行全局审查和纠正的机制。这与人类开发人员的工作方式截然不同 [3, 14]：开发人员通常会反思初次尝试，发现错误，并通过迭代加以改进。缺少这种细化机制，这些系统就无法检测或修正初始输出中的错误，往往会导致分组不理想，甚至存在逻辑矛盾。

为克服这些挑战，我们提出了 Atomizer，这是一种用于自动解缠复合提交的新型协作式多智能体框架。具体而言，为捕捉结构信息，我们采用一个*Purifier* 智能体，将代码变更表示为最小变更子图（MCSs），以对代码依赖关系建模。为更好地捕捉语义信息并减轻对结构信号的过度依赖，我们引入了一种**意图导向思维链（IO-CoT）**提示策略。IO-CoT 引导大语言模型（LLMs）逐步推理，模拟人类推断代码变更背后**意图**的过程。通过明确推理每项变更的目的，IO-CoT 帮助 LLMs 揭示其更深层的语义含义。为解决"单次处理"分组的局限性，该框架引入了由分组器 智能体和审查者 智能体组成的**分组器—审查者协作式细化循环** 。它遵循类似人类的审查流程，反复进行自我纠正，直至获得合理的划分。

为评估 Atomizer 的有效性，我们在两个广泛使用的 C# 和 Java 基准数据集上进行了大量实验。结果表明，Atomizer 的表现显著优于当前最先进的（SOTA）基线方法，在变更节点预测任务上取得了平均准确率 57% 的显著成绩。该任务旨在将代码变更正确地分组到相应的关注点中。此外，在评估包含大型图的复杂提交时，这种优势更加明显，Atomizer 相较于传统图聚类方法进一步拉大了性能差距。本文的贡献如下：

- 我们识别并解决了现有提交解缠方法的两项关键局限：过度依赖结构信号，却未能充分利用语义意图；以及缺少用于迭代纠正的优化循环。
- 我们提出了 Atomizer，这是一种用于复合提交解缠的多智能体框架，整合了两个组件：(1) 一种能够准确推断开发者意图的**意图导向思维链（IO-CoT）**提示策略；(2) 一个分组器-审查器协作细化循环，通过优化分组质量生成高质量的原子提交。
- 我们在广泛使用的 C# 和 Java 数据集上对 Atomizer 进行了严格、全面的评估。结果表明，Atomizer 达到了最先进的性能，显著优于现有方法，尤其是在处理具有大型图的复杂提交时。[2]

2 背景与相关工作

2.1 自动化复合提交解缠

自动化提交解缠的研究经历了三种主要范式：基于启发式规则、基于特征以及基于图聚类的方法。

基于启发式规则的方法。 该领域的早期工作依赖于人工定义的启发式规则来推断代码变更之间的关系。例如，*Herzig* 等人 [14] 将变更分解为单个操作，并使用置信度投票器预测这些操作之间的关系。类似地，*Kirinuki* 等人 [20] 提出基于历史变更数据库建议拆分提交，而*CoRA* [44] 则采用依赖关系分析和 PageRank 来识别对审查者而言重要的变更。另一种方法*Clus-terChanges* [3], 主要依据定义-使用和使用-使用代码关系来组织修改。这些方法虽然奠定了基础，但通常设计起来耗费大量人力，且缺乏通用性。

基于特征的方法。 为减少对手动创建规则的依赖，后续方法着重于从特征中学习。*EpiceaUntangler* [7] 收集细粒度特征，例如变更是否发生在同一个包或方法中，并使用聚类算法对其进行分组。在此基础上，*Yamashita* 等人 [47] 引入了*ChangeBeadsThreader*（CBT），这是一种交互式环境，首先使用该基于特征的聚类方法的简化版本，然后开发人员可以通过拆分或合并簇来手动优化结果。然而，基于特征的方法也受到预定义特征的限制，无法为开发人员取得令人满意的解缠结果。

基于图聚类的方法。 近年来，这种方法逐渐转向将提交表示为图，并应用聚类算法。*Flexeme* [32]构建包含名称流的多版本依赖图，并应用凝聚式

[2]我们的包已公开发布于：https://zenodo.org/records/17592234。

聚类。 *SmartCommit* [39] 通过边收缩对差异代码块图进行划分，以便交互式优化。 *UTango* [23] 对图神经网络（GNN）生成的代码变更嵌入进行聚类。*HD-GNN* [9] 使用层次化图神经网络检测提交之间的隐藏依赖关系。尽管近期取得了进展，基于图的方法仍面临两个核心局限。首先，它们依赖结构信号，却无法捕捉代码变更背后的语义意图。这往往会导致分组结果无法反映开发者意图。其次，它们通常缺少细化机制。初始分组完成后，系统便无法重新审视或修正结果。这些局限限制了现有方法的有效性，也表明我们需要更具意图感知能力且能够自我纠正的解决方案。

2.2 软件工程中的开发者意图分析

理解开发者意图对于提升自动化软件工程工具的准确性、相关性和可用性至关重要，因为这有助于弥合低层代码变更与高层开发目标之间的差距。意图建模在需求追踪、 [15], 代码生成 [21], 注释生成 [11, 29, 52], 和二进制摘要 [53]等任务中日益重要。一种常见策略是从提交信息中提取意图。例如，一些方法将提交信息中的句子分类为“决策”和“理由”等类别，以构建知识图谱 [6], ；另一些方法则使用预训练模型识别包含理由的句子 [4]。然而，这些方法严重依赖人工编写的消息是否存在及其质量，而在实际应用中，这类消息往往稀少或含糊不清 [31, 44]。尽管已有这些努力，现有的解缠技术仍未充分整合开发者意图，因此，在使自动化解缠结果与开发者的实际目标保持一致方面，仍有很大的改进空间。

2.3 用于代码推理的大语言模型

用于代码的大语言模型。 大语言模型（LLM）的应用标志着众多软件工程任务发生了显著的范式转变。这一演进始于CodeBERT*CodeBERT* [10]和*CodeT5* [46], 等基础模型，它们证明，在海量代码语料库上进行预训练，能够让大语言模型捕捉程序深层的上下文语义。随后，该领域涌现出一批强大的大型专有模型，例如 OpenAI 的*Codex* [46], ，它为第一代 GitHub Copilot 提供支持；以及 Google 的*AlphaCode* [22],，它在编程挑战中展现出媲美竞品的性能。近来，专门针对代码进行调优的高性能开源模型不断涌现，包括 Meta 的 *CodeLlama* [35],、BigCode 的*Star-Coder* [27],以及强大的 *DeepSeekCoder* [12],，进一步推动了这项能力的普及。这些现代模型展现出卓越的程序复杂语义理解能力，使其能够理解代码变更的高层意图，而这正是代码解缠任务的一项关键前提。

用于代码推理的思维链。 为应对复杂推理，一个关键技术是思维链（CoT）提示。事实证明，将思维链应用于面向代码的特定推理任务时，其效果尤为显著，因为它能引导模型阐述其中间逻辑步骤。例如，研究人员已成功应用思维链来推断一次变更的目的，以生成提交信息 [48, 51]。类似地，思维链也被用于增强复杂任务，例如代码生成 [30, 43], 代码补全 [18, 25], 和漏洞检测 [8, 28], ，所有这些任务都要求深入理解代码逻辑，而不仅仅是表层语法。这些成果凸显了将基于思维链的推理应用于我们所研究问题的潜力：推断纠缠的代码变更背后的实际意图。

基于大语言模型的多智能体系统在代码任务中的应用。 多智能体系统（MAS）是一种由多个相互协作的智能体组成的框架，它们能够协同解决单个智能体无法胜任的复杂问题。多智能体系统在软件工程领域有着广泛的应用，包括需求工程 [2, 19, 38], 代码生成 [26, 42, 49, 50], 和软件维护 [17, 33]。多智能体系统的架构通常遵循若干模式。角色扮演框架（如 *MARE* [19] 和*MetaGPT* [16],）通过模拟成熟的软件开发流程，提供结构化的工作流。任务分解框架（如*CodeS* [49]和 *MegaAgent* [45],）通过将大型问题拆解为更小、更易管理的任务来应对复杂性。此外，反馈驱动框架（包括*INTERVENOR* [42] 和*SpecRover* [36], ）通过迭代反馈实现自我纠正，从而提升输出质量和稳健性。这些架构模式凸显了多智能体系统在协调复杂推理与自我纠正方面的潜力，使其成为应对提交解缠等挑战的一种有前景的解决方案。

3 动机

为了说明现有解缠方法的根本局限性，我们以 *shadow-maint* 项目[3] 中的一个真实提交 作为示例。如图 1（左）所示，该提交将两个功能上截然不同的变更捆绑在一起：

- 变更 1（类型兼容性修复）：通过从 struct passwd *pw 变量声明中移除 const 限定符，消除一条编译警告。
- 变更 2（拼写错误修正）：通过将函数调用从 getspnam 更正为 getpwnam，修正一处拼写错误。

挑战 1：关键语义缺失。现有方法通常过度依赖结构信息，却未能有效融入语义理解。它们通常根据句法依赖关系对代码变更进行分组，往往忽略开发者的潜在意图——即驱动每项变更的目的或理由。如图 1（a）所示，尽管这两处编辑通过共享变量 pw在结构上相互关联，但它们实际上服务于两个无关的目的：一处是类型修复，另一处则是修正排版错误。由于结构联系紧密，基于图的方法误判了二者之间的关系，并错误地将它们归为同一个关注点。此例凸显了传统方法缺乏语义感知会导致严重的解缠错误。

挑战 2：缺乏细化机制。 现有基于图的方法的另一个局限在于它们是“单次处理”流程，缺乏迭代优化。它们会直接生成结果，却没有反思或自我纠正的机制。即使系统能够推断出基本意图，初始的

[3] https://github.com/shadow-maint/shadow/提交/2ba18ea4a90fc1502c56032a63729b08cd13cc80

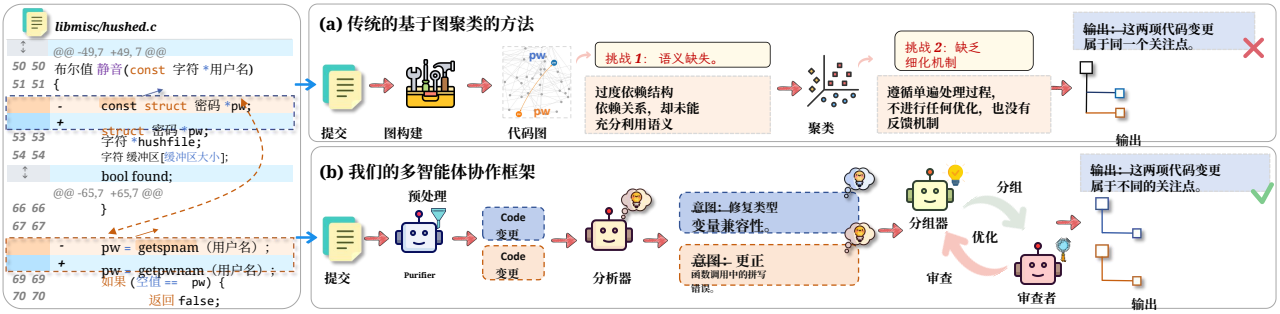

图 1：一个用于说明的示例 例：使用来自 *shadow-maint* 项目的一个真实复合提交（ID：2ba18ea）。该提交 it 包含两个具有不同 不同的理由：类型修复（移除 const）和拼写错误修正（getspnam → getpwnam）。

贪心分组仍可能存在缺陷。例如，自动化智能体可能会提议将这些变更分在一起，因为它们都被归类为“修复”。如果没有严格审查流程，这种看似合理但实际错误的分组就会成为最终输出。这与人类最佳实践形成鲜明对比：审查和迭代优化对于保证质量至关重要。

为克服上述挑战，我们的方法Atomizer采用了如图 1（b）所示的多智能体协作框架。**（1）为应对第一个挑战，**该分析器智能体会分析每项代码变更并推断其语义意图，不仅捕捉变更了什么，还捕捉为什么变更。例如，它可以识别出“修复：纠正变量的类型兼容性”或“修复：纠正函数调用中的拼写错误”等意图。这种意图感知分析使Atomizer 能够区分结构上相互关联但语义上无关的变更，而传统的基于依赖关系的方法通常无法做到这一点。**（2）为应对第二个挑战，**Atomizer 并不只是做出初始猜测。不同于在首次尝试后便终止处理的以往方法，Atomizer 采用了优化循环：由分组器智能体生成的初始分组会经过审查者智能体的审查，该智能体扮演专家审查者的角色；凭借全局且全面的视角，审查者智能体会评估分组后的变更是否确实属于同一组。在此示例中，它正确识别出“类型修复”和“拼写错误修复”解决的是不同问题，因此会拒绝错误的分组。随后，这一反馈会触发优化周期，最终得到正确且逻辑一致的结果。该示例展示了Atomizer的优势。它不再局限于脆弱的结构分析，而是首先推断开发者意图以理解这些变更；更重要的是，它引入了类似人类的协作审查与优化循环，以确保最终结果在逻辑上是合理的。这标志着从单遍分析到稳健且能够自我纠正的推理的范式转变。

4 方法

4.1 概述

图1 展示了Atomizer的概述。它采用协作式多智能体框架，系统地拆解复合提交。为克服传统方法的两大局限（语义缺失和缺乏优化循环），Atomizer首先解读代码变更背后的开发者意图，然后对分组进行严格审查和优化。该框架通过三阶段流程运行，由一个基础大语言模型管理四个专门的智能体：*Purifier* 智能体、分析器智能体、分组器智能体和审查者智能体。三个阶段如下：

1. **表征阶段：** *Purifier* 智能体执行预处理步骤，将代码变更信息表示为最小变更子图 （MCS）。它解析原始代码文件，获取包含结构信息的 MCS，为后续阶段提供简洁清晰的表示。
2. **分析阶段：** 为解决**语义缺失**问题，分析器智能体会对每个 MCS 进行深入的语义分析：它采用一种新颖的意图导向思维链（*IO-CoT*）提示策略，根据代码变更的语义信息推断开发者的深层意图；随后，为每项变更生成结构化的**意图档案**。
3. **优化阶段：** 为克服“单轮”分组的局限，本阶段模拟真实的代码审查流程，由分组器智能体和审查者智能体共同参与。分组器智能体首先运用意图驱动的贪心分组算法，生成初始分组结果。接下来，审查者智能体作为严格的审查者，评估所提分组的整体逻辑连贯性。根据评估结果，审查者智能体向分组器智能体提供反馈，后者据此修订分组。由此形成一个迭代式**分组器-审查器协作细化循环**，使框架能够逐步改进并收敛到准确的分组结果。

4.2 预处理阶段：最小变更子图构建

为了提高代码变更分析的效率，并消除无关代码语句的干扰，*Purifier* 智能体会构建经过优化的代码子图，称为最小变更子图（MCSs）。这些子图旨在仅捕获代码变更的关键信息，同时过滤掉无关的结构噪声。每个 MCS 都是原始抽象语法树（AST）的一个子图，其中节点表示单个语句级代码元素（例如赋值语句、方法调用、声明），边则表示由 AST 结构导出的语法关系（例如父子关系、块作用域）。具体而言，每个 MCS 包含两个关键组成部分：

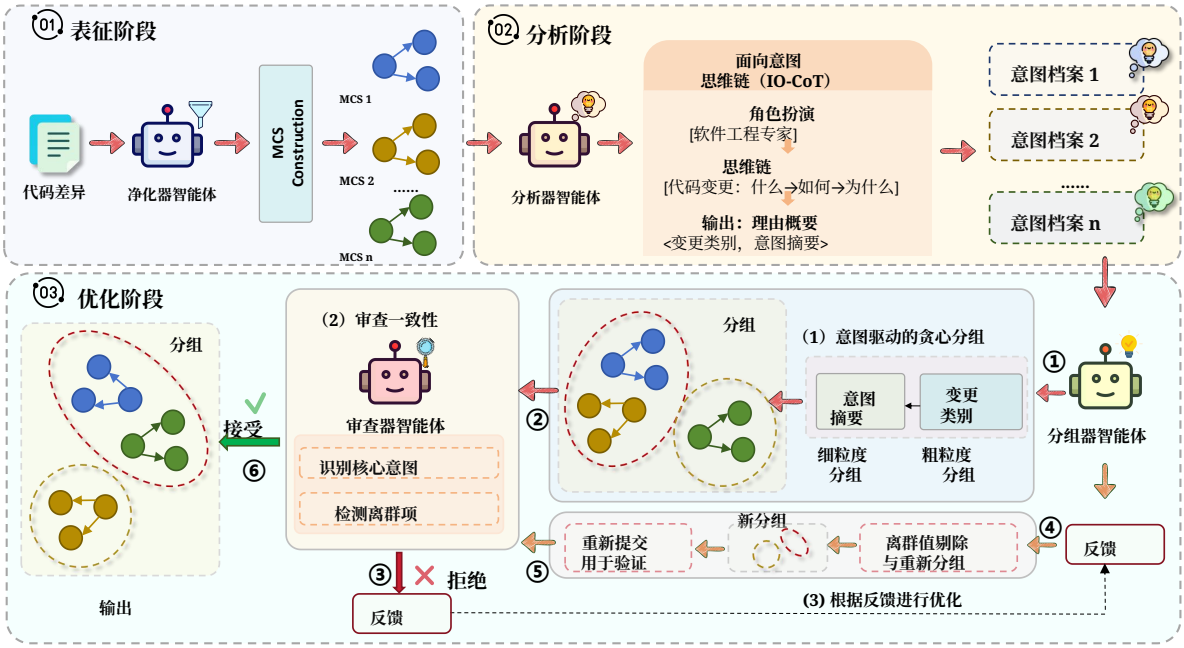

图2：Atomizer 概述。Atomizer 采用多智能体框架进行复合提交解缠。*Purifier*智能体首先对代码差异进行预处理，将其转换为聚焦的最小变更子图（MCS）。随后，*Profiler* 智能体采用意图导向思维链（IO-CoT）策略，推断每个 MCS 对应的开发者意图，以弥补语义缺失。最后，*Grouper* 智能体和*Reviewer* 智能体通过协作式的"分组—审查—优化"循环，使框架能够进行反思、自我纠正，并生成逻辑连贯的原子提交。

• **核心变更集：**一组在语法和逻辑上相互关联的语句级变更（添加或删除）。
• **必要语义上下文：** 对核心变更集进行语义解释所必需的最小未变更语句集合（例如，变量声明、方法签名）。

通过聚焦于这些经过精简且具备上下文感知能力的子图，最小变更子图有助于隔离有意义的变更模式，同时抑制来自周围无关代码的噪声。*Purifier* 智能体通过四个步骤构建最小变更子图：

(1) **种子节点识别**：首先，使用 *Tree-sitter*[4] 解析输入差异，将每个修改行映射到抽象语法树中对应的语句级节点。这些已识别的节点将作为后续分析的初始 种子节点。

(2) **通过变更内依赖分析识别核心变更集**：此步骤用于识别**核心变更集**，即种子节点中紧密耦合的一组代码修改。它仅分析种子节点之间的依赖关系，重点关注两种依赖：**数据依赖**，即某个变更语句使用了另一个变更语句中定义的变量或方法；以及**控制依赖**，即某个变更语句的执行取决于另一个变更语句，例如被嵌套在条件语句或循环中。通过这些依赖关系连接种子节点，该方法将其聚类为逻辑上连贯的**核心变更集**。每个集合对应一个由内部数据依赖边和控制依赖边连接的变更抽象语法树节点连通子图。

(3) **通过有界反向程序切片提取必要语义上下文**：此步骤提取理解每个**核心变更集**所需的**必要语义上下文**。通过对抽象语法树执行有界反向程序切片，从变更节点向后追踪，以收集最小集合的未更改语句，例如变量声明和方法签名；这些内容对于从语义上理解**核心变更集**至关重要。最终得到一个经过扩展的子图，其中包含变更节点及这些关键的未更改上下文节点，边则表示抽象语法树的父子关系以及控制/数据依赖关系。

(4) **最小变更子图定稿**：将每个**核心变更集** 与其**必要语义上下文**相结合，最终生成最小变更子图（MCS）。

4.3 分析阶段：通过意图导向思维链进行语义分析

接下来，为了弥补先前工作的 **语义缺失**，分析器智能体会分析每个 MCS，以推断变更意图。这一过程形式化为一个**意图档案**，即包含以下内容的结构化数据对象：
• **变更类别：**对变更目的进行高层分类（例如：缺陷修复、功能新增、重构）。
• **意图摘要：** 适合作为标准提交信息的简洁摘要。

[4]https://tree-sitter.github.io/tree-sitter/

提示 1：意图导向思维链（IO-CoT）

你是一名资深软件工程师 . 你的目标是仔细 - 分析以下代码变更，并遵循结构化推理过程来推断其意图。

输入代码变更（最小变更子图）： – a/[文件_路径]+++ b/[文件_路径]代码差异内容]

请严格遵循**what -> how->why**推理层级进行分析。**逐步思考**:

- **步骤 1：代码变更字面描述（What）：** 描述语法层面具体进行了哪些修改。添加、删除或更改了什么?
- **步骤 2：功能影响分析（How）：** 此变更如何实现其效果? 它对程序行为产生了什么直接的运行影响?
- **步骤 3：变更类别推断（Why）：** 为什么有必要进行这项功能变更? 开发者的最终目标是什么？请选择一个类别，并简要说明理由：[缺陷修复、功能、重构、性能、文档、测试、其他]。
- **步骤 4：意图摘要综合：** 根据完整分析（What、How 和 Why），用祈使语气撰写一条简洁的摘要（例如："修复....." "添加....."）。请以结构化 JSON 格式返回结果。

为准确生成意图画像，分析器智能体采用了一种新颖的**意图导向思维链（IO-CoT）**提示策略，在大语言模型上引导其复现专家开发者的认知工作流程（"what -> how ->why"），并要求生成结构化输出，以确保意图推断的质量和一致性（详见提示1 ）：

- **角色扮演：** 提示首先将大语言模型设定为"资深软件工程师"。这会激活模型关于软件工程的内部知识，使其生成的输出在术语、推理和领域相关性方面更加专业。

- **思维链：** IO-CoT旨在引导大语言模型复现专家开发者的认知工作流程，使其系统地从观察变更逐步深入到理解其意图。该工作流程按照清晰的"是什么-> 如何-> 为什么"推理层级展开，最终汇总为摘要。每个阶段都回答一个不同的问题：

1. **代码变更字面描述（是什么）：** 这一阶段通过回答"具体变更了什么？"来确立客观基础。模型以修改的原始语法为分析依据，避免偏离事实证据。

2. **功能影响分析（如何）：** 这一阶段超越语法层面，回答"这项变更如何实现其效果？"模型推理变更的运行后果，分析其对程序行为的直接影响。这一步是从观察代码变更到理解其功能意义的关键桥梁。

3. **变更类别推断（原因）：** 这一阶段旨在探究变更背后的动机，回答"为什么要进行这项变更？"模型不再关注语法模式或功能影响，而是转而推断开发者的高层意图，例如错误修复、功能实现或重构。它整合先前的推理，并引导大语言模型选择最能解释变更背后意图的预定义意图类别。我们使用预定义列表，因为这些离散类别对于类别筛选至关重要。开放式分类并不切实际，因为它可能引入含义模糊的同义词（例如，"缺陷修复"与"错误修复"），从而破坏这一关键筛选步骤。

4. **意图摘要综合：** 此阶段整合"什么-> 如何-> 为何"各阶段的输出，将整个分析链提炼为连贯、简洁的叙述，适合作为正式的提交信息。

- **结构化输出：** 提示要求采用结构化格式，包含**意图档案**： <变更类别、意图摘要>。

通过为每个 MCS 补充**意图档案**，此阶段为最终优化阶段提供了核心语义单元。

4.4 优化阶段：分组器-审查器协作式细化循环

推断出每项变更的意图后，最终的挑战是将它们分组成连贯的原子提交。传统聚类算法通过"单次遍历"完成这一任务，**却缺乏验证或优化其输出逻辑正确性的机制**。为解决这一问题，我们设计了一个**模拟真实代码审查流程**的协作阶段，由分组器智能体和审查者 智能体参与，形成迭代式的**分组器-审查者协作式优化**循环。

4.4.1 阶段 *1*：意图驱动的贪心分组。该分组器智能体负责初步起草，任务是生成一个合理的 MCS 分组，记为 $\mathcal{M} = \{m_1, \ldots, m_n\}$。它采用**局部、增量式的视角**，根据当前分组赋值状态，为每个 MCS 决定其归属。目标是使用意图驱动的贪心分组算法（如算法 1所示），生成一组连贯的最终分组$\mathcal{G}_{\text{set}}$ 。

1. **初始化**：该流程首先将第一个 MCSm_1 分配到一个新分组 $G_1 = m_1$，并将其加入输出集：$\text{G}_{\text{set}} = G_1$。 m_1 的意图档案用于初始化该分组的代表性意图，作为未来比较的语义锚点。2. **通过两阶段分组进行分层放置**：对于之后的每个 MCS $m_i \in$ M，分组器 代理会使用分层两阶段策略确定其放置位置：

- **粗粒度类别筛选：** 首先，它通过筛选代表性意图与 m_i具有相同变更类别的候选分组，对 Gset 进行过滤。由此得到的子集构成候选集 $C_i \subseteq \text{Gset}$。

- **细粒度意图匹配：** 在C_i中的候选项之间，模型会对 m_i 的意图摘要与各分组的代表性意图进行语义匹配。这样便能依据语义接近程度，而非仅凭标签匹配，实现更连贯的分组。.

3 **动态意图更新**：如果 m_i 被分配到现有分组$G_j \in C_i$，分组器 智能体会立即更新该分组的

算法 1：意图驱动的贪心分组

输入：一组具有意图档案的最小变更子图 M = {m_1、……、m_n}。

结果：一组分组 G_{set}。

函数贪心分组(M)：
  $G_{set} \leftarrow \varnothing$;
  如果$\mathcal{M}$ 不为空 **则**
    $G_1 \leftarrow \{m_1\}$;
    设置代表意图(G_1，获取意图配置(m_1));
    $G_{set} \leftarrow \{G_1\}$;
    对于每个 *MCS* m_i 从 m_2 到$m_{|\mathcal{M}|}$**执行**
      $C_i \leftarrow \varnothing$;
      对于每个分组 G_j 在 $\mathcal{G}_{set}$ **执行**
        如果获取类别(获取意图配置(m_i)) == 获取类别(获取代表意图(G_j)) **then**
          $C_i \leftarrow C_i \cup \{G_j\}$;
      $G_{best} \leftarrow$ 比较判断(m_i, C_i);
      如果 G_{best} 是一个新分组 **那么**
        $G_{new} \leftarrow \{m_i\}$;
        设置代表意图(G_{new}, 获取意图配置(m_i));
        $G_{set} \leftarrow G_{set} \cup \{G_{new}\}$;
      else
        $G_{best} \leftarrow G_{best} \cup \{m_i\}$;
        $new_intent \leftarrow$ 合成意图(G_{best});
        设置代表意图(G_{best}, new_intent);
  return $\mathcal{G}_{set}$;

代表性意图，将 m_i 的语义意图与 G_j 的语义意图进行综合。此动态调整可确保分组不断演进，以反映其汇总后的目的。如果找不到合适的分组，则会为 m_i 创建一个新分组，并将其添加到G_{set}。

此过程会生成一份完整的分组提案。然而，由于分组器代理的决策是贪婪且局部的，因此最终组成的分组可能缺乏全局一致性。这时，审查者 代理便会发挥必要的审查作用。

4.4.2 阶段 *2*：审查一致性。 审查者 代理充当专家审查者，其机制旨在系统地模拟人类专家的认知过程。为其判断建立可靠的基准，并避免潜在离群项造成偏差，其审查过程分两个步骤执行：

(1) **识别核心意图**：人工审查者通常会先理解一组相关代码变更背后的总体目的。与此行为相呼应，审查者代理会首先识别提议分组中的**最大连贯子集**，其定义为可由某个具体开发目的共同解释的最大 MCS 子集。推断出的目的被视为该分组动态确立的参考级**核心意图**，并作为下一步评估分组连贯性的依据。

(2) **检测离群项**：确立**核心意图**后，审查者 代理会通过大语言模型的定性推理，逐一检查分组中的每个 MCS，评估其个别意图是否与共同的核心意图逻辑一致。我们将**离群项** 定义为：其个别目的与已确立的核心意图不一致，因而无法逻辑地纳入**最大连贯子集**的任何 MCS。如果未发现离群项，则将该分组标记为“接受”。否则，将该分组标记为“拒绝”，并提供诊断反馈，其中包括已识别的离群项和该分组的核心意图。

4.4.3 阶段 *3*：根据反馈进行优化。此阶段根据阶段 2 的反馈优化分组。具体而言，分组器代理会从审查者代理处收到对每个组的判定：(1) 接受，即确认该组有效；或 (2) 拒绝，即表示组 G_i 中包含一个或多个 m_{outlier} 偏离该组核心意图的离群项。此判定会作为明确的反馈，指导后续优化。对此，系统会启动**智能纠正与重新分组协议**。

(1) **离群项剔除与重新分组**：首先从被拒绝的组 G_i中移除所有已识别的离群项。对于每个离群项，分组器代理都会寻找更合适的语义上下文。

- 识别候选组：筛选当前组集，找出与离群项具有相同**变更类别**的组。
- 语义判断：大语言模型会评估离群项在语义上是否适合归入任何候选组。如果找不到合适的候选组，则将该离群项放入一个新的单元素组。

(2) **重新提交验证**：相应地更新优化后组 G_i（现已移除离群项）的代表性意图。随后，将参与此次重组的所有组（包括修订后的 G_i 组，以及任何新建或修改的组）再次提交给审查者代理，以获取新一轮反馈。

这种由反馈驱动的“分组-审查-优化”循环会不断迭代，直至所有分组都获得接受决定，从而确保最终得到一组语义连贯且可验证正确的原子提交。

5 实验设置

本节介绍了多项实验，以评估 Atomizer 的有效性，并旨在回答以下三个研究问题（RQ）。

- **RQ1：Atomizer 在常用的 C#数据集和 Java 数据集上的效果如何?**
- **研究问题2：Atomizer 对复杂提交的效果如何?**
- **RQ3：Atomizer 的各个组件分别如何影响其整体性能?**

5.1 数据集

在本研究中，我们选用了以往研究中广泛使用的两个合成数据集：（1）**C#数据集** [32]：该数据集包含 1,612 个合成复合提交，由 9 个开源 C# 项目构建而成。每个合成提交均由 2 到 3 个原子提交合并而成 [5, 23, 32]。（2）**Java 数据集** [23]：该数据集

表 1：C#数据集上的性能。所有准确率均以百分比（%）表示。每个单元格按 Avg / OA 的格式显示结果。评估的 C# 项目包括 CL（Commandline）、CM（CommonMark）、HF（Hangfire）、HU（Humanizer）、LE（Lean）、NA（Nancy）、NJ（Newtonsoft.Json）、NI（Ninject）和 RS（RestSharp）。在 Acc^c%、Acc^a% 以及两个聚合指标（OA 和 Avg）上，Atomizer 的表现始终优于所有最先进基线，且所有提升均具有统计显著性（$p < 0.01$）。

指标	基线	项目（Avg% / OA%）									
		CL	CM	HF	HU	LE	NA	NJ	NI	RS	平均值
准确率^c%	*Barnett* 等人 [3]	14.2/14.8	13.1/13.5	12.6/11.3	13.9/13.2	7.4/8.1	8.6/8.3	7.9/7.4	10.2/10.8	10.1/9.5	10.9/10.8
	Herzig 等人 [14]	28.1/28.7	27.5/27.2	28.8/28.1	27.3/27.9	27.4/29.2	28.8/29.5	28.3/28.6	26.9/26.1	31.5/31.7	28.3/28.6
	δ-PDG+CV [32]	34.6/34.2	35.8/35.1	36.4/35.7	30.3/30.9	35.2/35.8	34.6/34.1	34.9/34.5	37.2/37.6	33.8/31.3	34.8/34.4
	Flexeme [32]	34.1/34.7	33.5/33.2	33.6/31.8	33.1/33.7	35.4/33.2	32.8/32.5	27.3/27.9	32.1/32.6	34.8/33.4	33.0/32.6
	UTango [23]	46.5/46.1	45.7/45.3	46.9/45.6	44.2/44.8	46.4/45.1	44.7/44.3	41.9/41.5	46.2/46.8	46.4/43.1	45.4/44.7
	HD-GNN [9]	52.3/52.9	51.5/51.1	52.7/48.4	50.2/50.8	50.6/48.3	49.9/49.5	45.2/45.8	54.4/54.1	47.7/48.3	50.5/49.9
	Atomizer（我们的方法）	**58.4/58.7**	**57.2/57.5**	**58.6/58.1**	**56.8/56.3**	**56.4/56.9**	**56.5/56.1**	**53.7/53.2**	**60.4/60.8**	**54.6/55.1**	**57.0/57.0**
	改进↑	+6.1/+5.8	+5.7/+6.4	+5.9/+9.7	+5.4/+5.5	+5.9/+8.6	+6.6/+6.6	+7.5/+7.4	+6.0/+6.7	+6.9/+6.7	+6.5/+7.1
准确率^a%	*Barnett* 等人 [3]	20.9/19.2	20.1/20.6	15.5/15.9	25.3/18.8	16.7/18.1	9.4/9.8	13.2/15.6	14.8/14.3	13.5/12.9	16.6/16.1
	Herzig 等人 [14]	58.4/64.1	65.7/65.3	62.2/64.8	53.6/62.2	66.8/69.4	63.1/67.7	64.5/71.9	57.3/57.8	70.2/70.6	62.4/66.0
	δ-PDG+CV [32]	81.3/80.8	90.5/90.1	86.7/87.3	69.9/69.4	78.2/84.8	83.6/86.2	78.8/82.4	94.1/94.7	64.6/70.1	80.9/82.9
	Flexeme [32]	87.5/82.1	70.8/70.3	77.2/79.8	70.6/81.2	80.8/80.4	87.1/84.7	62.6/71.2	80.9/80.5	86.3/82.9	78.2/79.2
	UTango [23]	90.7/90.2	89.4/89.8	88.3/86.6	89.9/89.4	89.2/88.8	88.5/88.1	83.7/83.3	91.9/91.5	88.2/87.8	88.9/88.4
	HD-GNN [9]	91.6/91.1	90.8/90.4	90.3/89.9	90.7/90.2	91.5/90.1	89.8/89.4	86.3/86.9	93.7/93.2	90.5/89.1	90.6/90.0
	Atomizer（我们的方法）	**97.6/97.1**	**96.8/96.3**	**96.5/96.9**	**97.2/97.8**	**95.4/95.1**	**95.7/95.3**	**97.5/97.9**	**97.2/97.8**	**96.4/96.1**	**96.7/96.7**
	改进↑	+6.0/+6.0	+6.0/+5.9	+6.2/+7.0	+6.5/+7.6	+3.9/+5.0	+5.9/+5.9	+11.2/+11.0	+5.5/+4.6	+5.9/+7.0	+6.1/+6.7

规模更大，包含来自 10 个不同开源 Java 项目的 14,000 多个合成复合提交 [23]。每个合成提交由 2 到 32 个原始原子提交合并而成，涵盖了广泛的纠缠复杂度。

数据集划分：我们遵循 *UTango* [23] 中的时间顺序划分方法。对于每个项目，按提交创建日期排序。对于可训练的基线方法，我们使用最早的 80% 提交进行训练，接下来的 10% 用于验证，最近的 10% 用于测试。不可训练的基线方法和我们的方法则直接在 10% 的测试集上进行评估。

5.2 基线方法

对于 C#数据集，我们将 Atomizer 与六种支持 C# 的代表性基线方法进行了比较。(1)*Barnett* 等人 [3]：一种基于启发式规则的方法，考虑代码变更之间的定义-使用、使用-使用和同一封闭方法关系。(2) *Herzig* 等人 [13]：一种基于启发式规则的方法，结合多种置信度投票器，并构建三角分区矩阵以解缠提交。(3)*Flexeme* [32]：一种基于图聚类的方法，构建提交的 *δ*-NFG，然后应用凝聚聚类来解缠提交。(4) *δ-PDG + CV* [32]：Flexeme 的一个变体，将*Herzig* 等人的置信度投票器直接应用于 *δ*-PDG。(5) *UTango* [23]：一种基于图聚类的方法，从提交构建 *δ*-PDG，然后应用 GNN 和凝聚聚类来解缠提交。(6) *HD-GNN* [9]：一种基于图聚类的方法，使用分层图检测隐藏依赖关系，然后应用 GNN 来解缠提交。

对于 Java 数据集，我们将 Atomizer 与多种支持 Java 的现有方法进行了比较，包括基于规则的方法：*Barnett* 等人 [3] 和*Herzig* 等人 [13], , 以及三种基于图聚类的方法：*SmartCommit* [39],*UTANGO* [23], 以及*HD-GNN* [9]。为确保比较更加全面，我们还纳入了文献 [39]中介绍的另外三种策略：(1)*SingleConcern-All* 是一种简单的基于规则的方法，会将所有变更归入一个关注点。(2) *SingleConcern-File* 是一种基于规则的方法，会将每个文件中的变更归入一个关注点。

5.3 评估指标

为评估解缠准确率，我们采用先前工作中广泛使用的两个指标 [5, 9, 23, 32]：

• **Acc$_c$（变更准确率）：** 由 *UTango* [23], 提出，该指标仅关注已更改的语句。它直接衡量模型对需要解缠的实际代码变更进行正确分组的准确程度。

$$\text{Acc}_c = \frac{\#\text{Correctly predicted changed statements}}{\#\text{All changed statements}}$$

• **Acc$_a$（绝对准确率）：** 由*Flexeme* [32], 提出，该指标计算所有语句（包括已更改和未更改语句）中正确分类的语句所占的百分比。它衡量模型的总体正确性，以及模型在不错误干扰大量未更改代码的情况下区分变更代码的能力。较高的 Acc$_a$ 表明，模型不仅能够很好地对变更进行分组，还能正确保留文件中稳定的部分。

$$\text{Acc}^a = \frac{\#\text{正确预测的语句 \#图中的所有语句}}{}$$

为说明这两个指标关注点的不同，假设一个提交包含100条语句，其中95条未更改，5条已更改。如果模型正确预测了所有未更改语句，以及60中的3条，那么其核心性能为Acc / %，而总体正确性为Acc$_a$ = ( )/( ) %。

$$_c = 3\ \ 5 = \text{条已更改语句}$$

$$95 + 3 \quad 95 + 5 \quad = 98$$

$$5$$

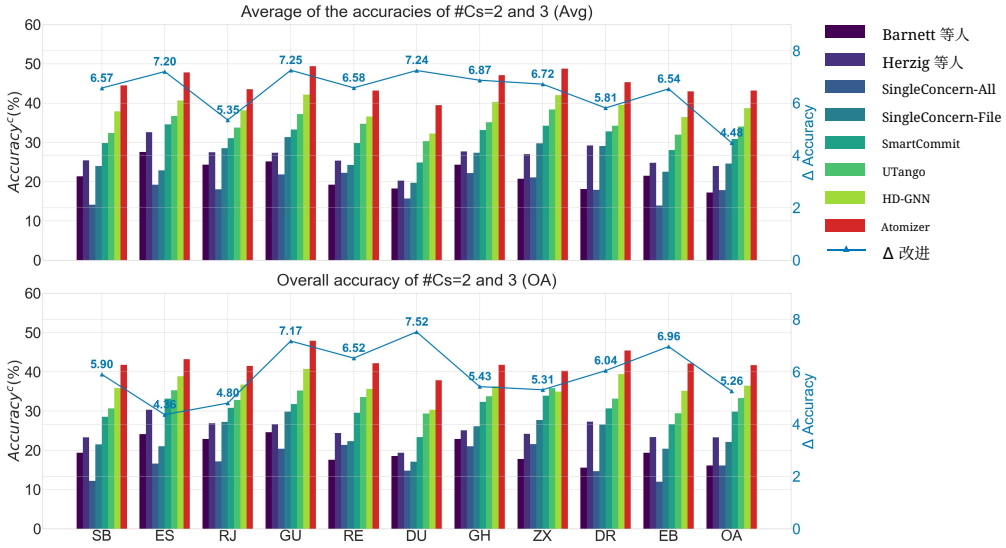

图3：在 Java 数据集上的有效性。评估的 Java 项目包括 SB（Spring Boot）、ES（Elasticsearch）、RJ（RxJava）、GU（Guava）、RE（Retrofit）、DU（Dubbo）、GH（Ghidra）、ZX（Zxing）、DR（Druid）和 EB（EventBus）。

遵循先前研究的做法 [5, 9, 23, 32], ，我们报告C#数据集的 Acc_a 和 Acc_c ，而对Java数据集仅报告 Acc_c 。这一决定旨在保持一致性，并确保与各数据集原始基准论文中确立的方法进行公平、直接的比较。遵循先前研究的做法 [5, 9, 23, 32]，我们还报告两个汇总准确率指标：**OA**（总体准确率），通过对测试集中所有多关注点提交整体评估 Acc_c 来计算。它衡量模型在整个数据集上的原始性能，并将每条已更改语句视为同等重要。（2）**平均值**（平均准确率），分别计算恰好包含两个和三个关注点的提交的 Acc_c 的平均值。它衡量模型在最常见解缠场景下按提交计算的典型性能，避免单个大型提交对评估结果产生偏差。

5.4 实现细节

所有实验均在一台 Linux 服务器上运行，服务器配备 Intel i9-12900K CPU 和两块 RTX 4090 GPU（每块 24GB）。Atomizer 默认使用 GPT-4o 作为推理代理，解码参数设置为 temperature=0、top_p=1 和 n=1，以确保确定性。迭代的 分组-审查-优化 循环最多运行 3 轮，以避免无限循环。我们还在相同设置下测试了 Gemini-2.5-Pro 和 DeepSeek-V3，以评估模型的泛化能力。

6 实验结果

6.1 RQ1：在 C#数据集和 Java 数据集上的有效性。

C#数据集上的有效性。 表 1展示了在 C#数据集上的比较结果。对于变更节点预测准确率（Acc^c%），Atomizer 显著优于所有最先进（SOTA）方法。平均而言，Atomizer 在*Barnett* 等人、*Herzig* 等人、*Flexeme*、 *δ-PDG + CV*、*UTango*、*HD-GNN* 方面分别提升了 46.1%、28.7%、22.2%、24.0%、11.6% 和 6.5%，指标为**平均值** 准确率。A

表 2：不同复杂度 C# 提交的性能比较（Acc^c%）。

节点范围	数量	*HD-GNN*	Atomizer（我们的方法）	提升幅度
0-1000	605	53.7	59.6	+5.9%
1000-2000	385	49.3	58.2	+8.9%
2000-7000	513	46.5	58.5	+12.0%
>7000	109	41.4	57.6	+16.2%

在**OA** 指标上也观察到类似趋势，相应提升分别为 46.2%、28.4%、22.6%、24.4%、12.3% 和 7.1%。Atomizer 在全节点准确率（Acc^a%）上同样取得了最高的整体性能。值得注意的是，所有模型在 Acc^a% 上的结果都高于 Acc^c%，因为它们默认能够正确分类未变更节点。特别是，与*HD-GNN* [9],这一最强的基于图聚类的基线相比，Atomizer 在三个指标上的平均增益均超过 6.0%。为验证改进的显著性，我们进行了配对 t 检验，结果显示 Atomizer 相较于所有基线方法的性能提升均具有统计显著性，所有 p-值均低于 0.01。

在 Java 数据集上的有效性。 图 3展示了 Java 数据集上的结果。在 **平均值** 和 **OA** 指标中，Atomizer 始终表现最佳。与最强的基线方法 *HD-GNN*相比，Atomizer 平均提升超过 5.5%。配对 t 检验证实，相较于基线方法的所有提升在统计学上均显著，p值低于 0.01。

> **回答RQ1：**实验结果表明，在 C#数据集和 Java 数据集上，Atomizer 的表现始终优于所有最先进基线。平均而言，它分别超过最强基线 *HD-GNN* 6.0%以上↑ 和5.5%↑。对于两个数据集，这些提升均具有统计显著性（$p < 0.01$）。

表3：C#数据集上的消融研究结果（Acc^c%）。

变体	平均准确率（%）	OA准确率（%）
Atomizer	57.0	57.0
不使用净化	48.1	49.2
不使用 RA-CoT	49.8	50.5
不使用审查	51.2	51.9

6.2 研究问题2：对复杂提交的有效性。

为进一步研究 Atomizer在不同复杂度水平下的性能，我们根据图规模，具体而言是图中包含的节点数量，进行了分层分析。我们按照图规模对 C#数据集中的所有提交进行排序，并将其划分为四个复杂度组：0–1000、1000–2000、2000–7000，以及 >7000 个节点。值得注意的是，在现实世界的版本历史中，超过 7000 个节点的图规模极大 [5, 9, 23]；很少有提交超过这一规模，提交的平均规模通常小于 1500 个节点。因此，该组代表了解缠中最具挑战性的情况。如表 2 所示，**随着提交复杂度增加，其性能保持稳定，甚至略有提升，而最先进基线 *HD-GNN* 的准确率则急剧下降**。具体而言，Atomizer 即使对于规模最大的提交，也能保持 58–59% 的稳定准确率，而 *HD-GNN* 则从 53.7% 降至仅 41.4%。因此，两种方法之间的准确率差距从最简单组的 +5.9% 扩大到最复杂组的显著 +16.2%。这一差异凸显了我们方法的根本优势。Atomizer通过预处理获取聚焦的最小变更子图，并推理开发者意图，因此能够稳健地处理大型图。相比之下，*HD-GNN*依赖图聚类，在大型提交中越来越容易受到无关依赖的影响，导致其性能显著下降。

> **回答研究问题2：** 对于具有大型图的复杂提交，Atomizer 尤为有效。随着提交规模增大，现有最强方法 *HD-GNN* 的性能下降了 12.3% 以上↓，而 Atomizer 的准确率保持稳定，甚至略有提升。

6.3 RQ3：Atomizer 的各个组件如何为整体性能做出贡献?

为评估Atomizer 框架中不同组件的性能，我们进行了一系列消融研究，如表 3所示。我们设计了三个Atomizer 消融变体，每次禁用一个关键组件，并将这些变体与完整实现进行比较：

- **无净化**：跳过净化阶段，将原始且含噪的文件差异直接提供给分析器 代理，而不是提供经过清理的最小变更子图。
- **无 IO-CoT**：在分析阶段用零样本提示替换 IO-CoT 策略，要求大语言模型直接生成意图摘要。
- **不含审查**：禁用审查者 智能体和审阅—优化循环。输出是分组器 智能体未经进一步验证的初始贪心分组算法分组结果。

表 4：Atomizer 在 C#数据集上使用不同基础大语言模型时的性能（Acc^c%）。

基础大语言模型	平均准确率（%）	总体准确率（%）
GPT-4o（主模型）	57.0	57.0
Gemini-2.5-Pro	56.9	57.1
DeepSeek-V3	56.4	56.8

表 3中的结果清楚地表明，各组件都对框架的整体性能作出了显著且积极的贡献。Atomizer 不含净化和 Atomizer 不含 IO-CoT 的变体性能大幅下降。移除初始净化阶段会导致准确率骤降，这证实了我们的假设：如果不将原始差异净化为干净的最小变更子图，后续智能体就会受到结构噪声的干扰，进而导致意图推断出现偏差。同样，移除 IO-CoT 策略也会造成显著的性能损失，这验证了明确引导大语言模型按照结构化的“是什么 -> 如何 -> 为何”的推理流程，对于准确辨别开发者意图至关重要。最后，Atomizer 不含审查的变体准确率也出现了明显下降。这一结果凸显了协作分组与审查阶段的价值，证明审查者智能体的全局视角能够有效识别并纠正分组器 智能体在初始贪心分组阶段产生的逻辑不一致，从而将输出优化得更加连贯、准确。

> **回答 RQ3：** Atomizer 框架的每个组件都发挥着至关重要且积极的作用。消融研究表明，净化阶段对于减少结构噪声至关重要，IO-CoT 对于准确进行语义理解不可或缺，而协同审查阶段则提供了必要的验证机制，从而提高了最终的分组准确率。结果证实，这些专门阶段之间的协同作用是 Atomizer 卓越性能的关键。

7 讨论

7.1 Atomizer 对不同基础大语言模型的影响

为验证Atomizer的泛化能力，我们进行了一项实验，替换了其核心推理引擎。本文的主要结果采用 GPT-4o 得出；在此实验中，我们将其替换为另外两种强大的大语言模型：Gemini-2.5-Pro 和 DeepSeek-V3。其余Atomizer架构保持不变。为了确保公平比较，三个模型均采用相同的确定性参数（温度=0、top_p=1），并在 C#数据集上进行评估。如表4所示，尽管 GPT-4o 的性能最佳，Gemini-2.5-Pro 和 DeepSeek-V3 也取得了有竞争力的结果。这证实了Atomizer 的多智能体设计具有稳健性，不依赖特定模型，并且适用于不同的大语言模型。这表明，更强大的大语言模型虽然能够提升最终准确率，但Atomizer 架构本身才是其成功的主要驱动力。

表5：在专家标注的MVD数据集上的性能比较 [9]。

方法	C# MVD		Java MVD
	准确率^c (%)	准确率^a (%)	准确率^c (%)
Barnett 等人 [3]	9.2	51.4	35.7
UTango [23]	38.6	74.1	37.9
HD-GNN [9]	43.4	81.3	48.9
Atomizer（我们的方法）	**61.6**	**94.5**	**64.7**

7.2 真实世界数据集上的评估

为评估Atomizer在真实世界数据上的性能，我们使用来自 [9],的小型自定义数据集开展了实验。该数据集由专家开发人员标注，用于检测提交中的隐藏依赖关系。如表5所示，结果表明，Atomizer 在这一具有挑战性的数据集上显著优于强大的基线方法。具体而言，Atomizer 在准确率方面显著优于其他方法，在 C# MVD任务上的准确率（Acc^c）达到 61.6%，在 Java MVD任务上达到 64.7%，在真实世界数据集上的准确率（Acc^a）则高达 94.5%。这些结果表明，Atom-izer 应用于真实世界数据集时，能够提供更高的准确率和更强的依赖关系检测能力。

7.3 成本分析

我们分析了 Atomizer 的计算成本，该成本与任务复杂度相关（如表 6 所示）。在处理提交时，GPT-4o 的平均性能因数据集而存在显著差异。对于 C#数据集（平均包含 2–3 个关注点），处理一次提交约需 48 秒，并消耗 16,500 个令牌。然而，对于更复杂的 Java 数据集（关注点范围更广，为 2 至 32 个），这一数值增至平均 63 秒和 20,100 个令牌。Gemini-2.5-Pro 和 DeepSeek-V3 也呈现出相同的处理时间和令牌使用量增长趋势。根据主流 API定价，使用 GPT-4o 时，每次提交的成本约为 0.053 美元（C#）和 0.064 美元（Java）。使用 Gemini-2.5-Pro 时，成本较低，分别为 0.047 美元和 0.059 美元；使用 DeepSeek-V3 时，分别为 0.026 美元和 0.032 美元。 分析器 代理是主要的成本来源，占令牌总量的 72%，因为它需要对每个 MCS 执行详细的语义分析。在涉及超过 7,000 个节点的复杂提交图场景中，Atomizer 相较于传统的基于GNN的方法平均提升约 16.2%；同时，使用 GPT-4o 时每次提交的成本仅为 0.069 美元，展现出出色的成本效益。为进一步优化成本，一种可行的方法是将低成本的传统方法与强大的基于大语言模型的方法相结合。具体而言，对于简单且无歧义的提交，组合方法首先使用低成本的基于图的方法进行初步筛选；对于复杂或语义模糊的提交，则采用基于大语言模型的多智能体框架。一个关键挑战是设计有效的方法来分类提交难度。这种组合策略应能在实际部署中实现切实可行的平衡。

表6：Atomizer成本分析。

模型	指标	C#	Java
GPT-4o	时间（秒）	48	63
	令牌	16,500	20,100
	成本（美元）	0.053	0.064
Gemini-2.5-Pro	时间（秒）	61	78
	令牌	15,300	18,470
	成本（美元）	0.047	0.059
DeepSeek-V3	时间（秒）	55	69
	令牌	15,720	17,690
	成本（美元）	0.026	0.032

7.4 审查器智能体重新评估频率分析

对 Atomizer 重新评估频率的统计分析表明，85% 的 C# 提交在一轮内完成，9% 需要两轮，6% 需要三轮。对于更复杂的 Java 数据集，这一分布变为 71%、17% 和 12%。值得注意的是，67% 的包含超过 16 个关注点的提交需要进行三轮评估。此外，80% 的重新评估会为“离群项”创建新的单元素分组，而 20% 则涉及将变更合并到现有分组中。

7.5 有效性威胁

内部有效性。 尽管我们使用开发者标注的原子提交作为真实值，但其中可能存在固有噪声或轻微纠缠，从而影响绝对准确率。此外，虽然我们将大语言模型的温度固定为 0 以尽量减少随机性，但无法保证完全确定性，而且 审查者 智能体仍受限于底层模型的推理能力。

外部有效性。 我们的研究结果基于开源 C# 和 Java 数据集，可能无法完全推广到专有系统或其他编程语言。此外，尽管合成数据集（合并的原子提交）已成为标准，但它们可能缺乏真实世界变更中未结构化的复杂性。我们通过真实世界数据集对此进行了补充，但其规模有限且项目领域特定，可能无法涵盖所有开发场景。

8 结论

在本研究中，我们介绍了 Atomizer，这是一种旨在通过捕捉开发者意图并提供自我优化机制来拆分复合提交的多智能体框架。通过采用意图导向思维链（IO-CoT）策略和迭代式“审查并优化”流程，Atomizer 实现了类似人类的自我纠正。在 C# 和 Java 数据集上进行的实验表明，Atomizer 达到了最先进的性能，显著优于基线方法，尤其是在处理大型复杂提交时。

致谢

作者谨向匿名审稿人致以诚挚谢意，感谢他们提出的建设性意见。本研究得到国家自然科学基金（项目编号：62402506、62474196）和国防科技大学科研基金（项目编号：ZK24-05）的资助。

参考文献

[1] Khadijah Al Safwan、Mohammed Elarnaoty 和 Francisco Servant。2022。开发人员对代码提交理由的需求：一项广度与深度研究。《系统与软件杂志》189（2022），111320。[2] Mohammadmehdi Ataei、Hyunmin Cheong、Daniele Grandi、Ye Wang、Nigel Morris 和 Alexander Tessier。2024。Elicitron：一种用于设计需求引出的基于大语言模型智能体的仿真框架。arXiv 预印本 arXiv:2404.16045 None, None（2024）。[3] Mike Barnett、Christian Bird、João Brunet 和 Shuvendu K Lahiri。2015。帮助开发人员自助：自动分解代码审查变更集。载于《2015年IEEE/ACM第37届IEEE国际软件工程会议》，第1卷。IEEE，None，None，134–144。[4] Francesco Casillo、Antonio Mastropaolo、Gabriele Bavota、Vincenzo Deufemia 和 Carmine Gravino。2024。迈向代码变更理由的生成。（2024）。[5] Siyu Chen、Shengbin Xu、Yuan Yao 和 Feng Xu。2022。通过属性图聚类解缠复合提交。载于《第13届亚太互联网软件研讨会论文集》。117–126。[6] Mouna Dhaouadi、Bentley James Oakes 和 Michalis Famelis。2023。利用知识图谱方法理解和分析提交消息中的理由。载于《2023年ACM/IEEE模型驱动工程语言与系统国际会议伴会（MODELS-C）》。IEEE，622–630。[7] Martín Dias、Alberto Bacchelli、Georgios Gousios、Damien Cassou 和 Stéphane Ducasse。2015。解缠细粒度代码变更。载于《2015年IEEE第22届软件分析、演化与再工程国际会议（SANER）》。IEEE，341–350。[8] Xueying Du、Geng Zheng、Kaixin Wang、Jiayi Feng、Wentai Deng、Mingwei Liu、Bihuan Chen、Xin Peng、Tao Ma 和 Yiling Lou。2024。Vul-rag：通过知识层级检索增强生成提升基于大语言模型的漏洞检测。arXiv 预印本 arXiv:2406.11147（2024）。[9] Mengdan Fan、Wei Zhang、Haiyan Zhao、Guangtai Liang 和 Zhi Jin。2024。检测隐藏依赖以解缠提交。载于《第39届IEEE/ACM国际自动化软件工程会议论文集》。179–190。[10] Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xiaocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、Daxin Jiang 等。2020。Codebert：一种面向编程语言和自然语言的预训练模型。arXiv 预印本 arXiv:2002.08155（2020）。[11] Mingyang Geng、Shangwen Wang、Dezun Dong、Haotian Wang、Ge Li、Zhi Jin、Xiaoguang Mao 和 Xiangke Liao。2024。大语言模型是少样本摘要生成器：通过上下文学习生成多意图评论。载于《第46届IEEE/ACM国际软件工程会议论文集》。1–13。[12] Daya Guo、Qihao Zhu、Dejian Yang、Zhenda Xie、Kai Dong、Wentao Zhang、Guanting Chen、Xiao Bi、Yu Wu、YK Li 等。2024。DeepSeek-Coder：当大语言模型遇上编程——代码智能的崛起。arXiv 预印本 arXiv: *2401 14196* *2024* 10 48550 2401 14196。（）。doi：。/ [13] 2016 arXiv.。Kim Herzig、Sascha Just 和 Andreas Zeller。。纠缠的代码变更对缺陷预测模型的影响。《实证软件工程》21（2016），303–336。[14] Kim Herzig 和 Andreas Zeller。2013 2013 10 121 130。纠缠的代码变更的影响。载于第届软件仓库挖掘工作会议（MSR）。IEEE，–。[15] Tobias Hey。2019。INDIRECT：意图驱动的需求到代码可追溯性。载于《2019年IEEE/ACM第41届国际软件工程会议：伴会论文集（ICSE-Companion）》。IEEE，190–191。[16] Sirui Hong、Xiawu Zheng、Jonathan Chen、Yuheng Cheng、Jinlin Wang、Ceyao Zhang、Zili Wang、Steven Ka Shing Yau、Zijuan Lin、Liyang Zhou 等。2023。MetaGPT：面向多智能体协作框架的元编程。arXiv 预印本 arXiv:2308.00352 3, 4（2023），6。[17] Sihao Hu、Tiansheng Huang、Fatih Ilhan、Selim Furkan Tekin 和 Ling Liu。2023。大语言模型驱动的智能合约漏洞检测：新视角。载于《2023年第5届IEEE智能系统与应用中的信任、隐私与安全国际会议（TPS-ISA）》。IEEE，297–306。[18] Rasha Ahmad Husein、Hala Aburajouh 和 Cagatay Catal。2024。用于代码补全的大语言模型：系统性文献综述。《计算机标准与接口》（2024），103917。[19] Dongming Jin、Zhi Jin、Xiaohong Chen 和 Chunhui Wang。2024。MARE：面向需求工程的多智能体协作框架。arXiv 预印本 arXiv:2405.03256（2024）。[20] Hiroyuki Kirinuki、Yoshiki Higo、Keisuke Hotta 和 Shinji Kusumoto。2016。通过历史代码变更拆分提交。载于《2016年第23届亚太软件工程会议（APSEC）》。IEEE，129–136。[21] Shuvendu K Lahiri、Sarah Fakhoury、Aaditya Naik、Georgios Sakkas、Saikat Chakraborty、Madanlal Musuvathi、Piali Choudhury、Curtis von Veh、Jeevana Priya Inala、Chenglong Wang 等。2022。通过测试驱动的用户意图形式化实现交互式代码生成。arXiv 预印本 arXiv:2208.05950（2022）。[22] Yujia Li、David Choi、Junyoung Chung、Nate Kushman、Julian Schrittwieser、Rémi Leblond、Tom Eccles、James Keeling、Felix Gimeno、Agustin Dal Lago 等。2022。利用AlphaCode实现竞赛级代码生成。《科学》378, 6624（2022），1092–1097.[23] Yi Li、Shaohua Wang 和 Tien N Nguyen。2022。UTANGO：一种利用上下文感知、基于图的代码变更聚类学习模型解缠提交的方法。载于《第30届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集》。221–232。[24] Yikun Li、Ting Zhang、Ratnadira Widyasari、Yan Naing Tun、Huu Hung Nguyen、Tan Bui、Ivana Clairine Irsan、Yiran Cheng、Xiang Lan、Han Wei Ang 等。2024。CleanVul：使用大语言模型启发式方法自动检测代码提交中的函数级漏洞。arXiv 预印本 arXiv:2411.17274（2024）。[25] Zongjie Li、Chaozheng Wang、Zhibo Liu、Haoxuan Wang、Dong Chen、Shuai Wang 和 Cuiyun Gao。2023。Cctest：测试和修复代码补全系统。载于《2023 IEEE/ACM第45届国际软件工程会议（ICSE）》。IEEE，1238–1250。[26] Zihan Liu、Ruinan Zeng、Dongxia Wang、Gengyun Peng、Jingyi Wang、Qiang Liu、Peiyu Liu 和 Wenhai Wang。2024。Agents4PLC：使用基于大语言模型的智能体自动化工业控制系统中的闭环PLC代码生成与验证。arXiv 预印本 arXiv:2410.14209（2024）。[27] Anton Lozhkov、Raymond Li、Loubna Ben Allal、Federico Cassano、Joel Lamy- Poirier、Nouamane Tazi、Ao Tang、Dmytro Pykhtar、Jiawei Liu、Yuxiang Wei 等。2024。Starcoder 2 和 the stack v2：新一代。arXiv 预印本 arXiv:2402.19173（2024）。[28] Guilong Lu、Xiaolin Ju、Xiang Chen、Wenlong Pei 和 Zhilong Cai。2024。GRACE：利用图结构和上下文学习增强基于大语言模型的软件漏洞检测。《系统与软件杂志》212（2024），112031。[29] Fangwen Mu、Xiao Chen、Lin Shi、Song Wang 和 Qing Wang。2023。由开发者意图驱动的代码注释生成。载于《2023 IEEE/ACM第45届国际软件工程会议（ICSE）》。IEEE，768–780。[30] Fangwen Mu、Lin Shi、Song Wang、Zhuohao Yu、Binquan Zhang、ChenXue Wang、Shichao Liu 和 Qing Wang。2024。Clarifygpt：通过需求澄清增强基于大语言模型的代码生成的框架。《ACM软件工程论文集》1，FSE（2024），2332–2354。[31] Hoan Anh Nguyen、Anh Tuan Nguyen 和 Tien N Nguyen。2013。过滤混合用途修复提交中的噪声，以改进缺陷预测与定位。载于《2013 IEEE第24届国际软件可靠性工程研讨会（ISSRE）》。IEEE，138–147。[32] Profir-Petru Pârțachi、Santanu Kumar Dash、Miltiadis Allamanis 和 Earl T Barr。2020。Flexeme：利用词汇流解缠提交。载于《第28届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集》。63–74。[33] Yihao Qin、Shangwen Wang、Yiling Lou、Jinhao Dong、Kaixin Wang、Xiaoling Li 和 Xiaoguang Mao。2025。Soap FL：一种用于大语言模型方法级故障定位的标准操作规程。《IEEE软件工程汇刊》（2025）。[34] Peter Rigby、Brendan Cleary、Frederic Painchaud、Margaret-Anne Storey 和 Daniel German。2012。实践中的当代同行评审：来自开源开发的经验。《IEEE软件》29，6（2012），56–61。[35] Baptiste Roziere、Jonas Gehring、Fabian Gloeckle、Sten Sootla、Itai Gat、Xiao- qing Ellen Tan、Yossi Adi、Jingyu Liu、Romain Sauvestre、Tal Remez 等。2023。Code llama：面向代码的开放基础模型。arXiv 预印本 arXiv:2308.12950（2023）。[36] Haifeng Ruan、Yuntong Zhang 和 Abhik Roychoudhury。2024。SpecRover：通过大语言模型进行代码意图提取。arXiv 预印本 arXiv:2408.02232（2024）。[37] Khadijah Al Safwan 和 Francisco Servant。2019。分解代码提交的理由：软件开发者的视角。载于《2019年第27届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集》。397–408。[38] Malik Abdul Sami、Muhammad Waseem、Zheying Zhang、Zeeshan Rasheed、Kari Systä 和 Pekka Abrahamsson。2024。用于需求引出与分析的基于人工智能的多智能体方法。arXiv 预印本 arXiv:2409.00038（2024）。[39] Bo Shen、Wei Zhang、Christian Kästner、Haiyan Zhao、Zhao Wei、Guangtai Liang 和 Zhi Jin。2021。SmartCommit：一种面向活动导向型提交的基于图的交互式助手。载于《第29届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集》。379–390。[40] Yida Tao、Yingnong Dang、Tao Xie、Dongmei Zhang 和 Sunghun Kim。2012。软件工程师如何理解代码变更？一项工业界探索性研究。载于《ACM SIGSOFT第20届软件工程基础国际研讨会论文集》。1–11。[41] Yida Tao 和 Sunghun Kim。2015。对复合代码变更进行分区以便于代码审查。载于《2015 IEEE/ACM第12届软件仓库挖掘工作会议》。IEEE，180–190。[42] Hanbin Wang、Zhenghao Liu、Shuo Wang、Ganqu Cui、Ning Ding、Zhiyuan Liu 和 Ge Yu。2023。Intervenor：通过交互式修复链激发大语言模型的编码能力。arXiv 预印本 arXiv:2311.09868（2023）。[43] Jianxun Wang 和 Yixiang Chen。2023。大语言模型代码生成综述：应用与评估。载于《2023 IEEE国际医学人工智能会议（MedAI）》。IEEE，284–289。

[44] Min Wang、Zeqi Lin、Yanzhen Zou 和 Bing Xie。2019。CoRA：为审查者分解并描述纠缠的代码变更。载于*2019*年*IEEE/ACM*第*34*届国际自动化软件工程会议（*ASE*）。IEEE，1050–1061。[45] Qian Wang、Tianyu Wang、Qinbin Li、Jingsheng Liang 和 Bingsheng He。2024。MegaAgent：面向大规模大语言模型智能体系统自主协作的实用框架。*arXiv* 预印本 *arXiv:2408.09955*（2024）。[46] Yue Wang、Weishi Wang、Shafiq Joty 和 Steven CH Hoi。2021。CodeT5：用于代码理解与生成的标识符感知统一预训练编码器—解码器模型。*arXiv* 预印本 *arXiv:2109.00859*（2021）。[47] Satoshi Yamashita、Shinpei Hayashi 和 Motoshi Saeki。2020。Changebead- sthreader：用于定制自动解缠变更的交互式环境。载于*2020*年*IEEE*第*27*届软件分析、演进与再工程国际会议（*SANER*）。IEEE，657–661。[48] Guang Yang、Yu Zhou、Xiang Chen、Xiangyu Zhang、Terry Yue Zhuo 和 Taolue Chen。2024。神经代码生成中的思维链：面向轻量级语言模型及由其驱动。*IEEE*软件工程汇刊（2024）。[49] Daoguang Zan、Ailun Yu、Wei Liu、Dong Chen、Bo Shen、Wei Li、Yafen Yao、Yongshun Gong、Xiaolin Chen、Bei Guan 等。2024。CodeS：自然语言到

通过多层草图构建代码仓库。*arXiv* 预印本 *arXiv:2403.16443* (2024)。[50] Huan Zhang、Wei Cheng、Yuhan Wu 和 Wei Hu。2024。通过多方案探索和反馈驱动的优化实现代码生成的结对编程框架。载于第*39*届*IEEE/ACM*国际自动化软件工程会议论文集。1319–1331。[51] Linghao Zhang、Jingshu Zhao、Chong Wang 和 Peng Liang。2024。使用大语言模型生成提交信息：一项初步研究。载于*2024*年*IEEE*软件分析、演化与再工程国际会议（*SANER*）。IEEE，126–130。[52] Xiaowei Zhang、Zhifei Chen、Yulu Cao、Lin Chen 和 Yuming Zhou。2024。基于大语言模型的多意图行内代码注释生成。《style id='10'>《国际软件工程与知识工程杂志》34, 06 (2024), 845–868。[53] Kangchen Zhu、Zhiliang Tian、Shangwen Wang、Weiguo Chen、Zixuan Dong、Mingyue Leng 和 Xiaoguang Mao。2025。MiSum：用于多意图二进制代码摘要的多模态异构代码图学习。*ACM*软件工程论文集2, FSE (2025), 1339–1362。