RepoRepair：利用代码文档实现代码仓库级自动程序修复

ZHONGQIANG PAN, CHUANYI LI*, WENKANG ZHONG, YI FENG, and BIN LUO,新软件技术国家重点实验室，南京大学，中国VINCENT NG, 人类语言技术研究所，得克萨斯大学达拉斯分校，美国

arXiv:2603.01048v1 [cs.SE] 1 Mar 2026

自动程序修复（APR）难以从孤立函数扩展到完整代码仓库，因为它需要具备全局且了解任务的理解能力，才能定位必要的更改。受上下文限制，并依赖浅层检索或成本高昂的智能体迭代，现有方法难以处理复杂的跨文件问题。为此，我们提出 RepoRepair，这是一种用于代码仓库级故障定位和程序修复的新型文档增强方法。我们的核心思路是利用大语言模型为代码仓库生成层次化代码文档（从函数到文件），创建结构化的语义抽象，使大语言模型能够理解代码仓库级上下文和依赖关系。具体而言，RepoRepair 首先使用基于文本的大语言模型（例如 DeepSeek-V3）为代码仓库生成文件级/函数级代码文档，作为辅助知识来指导故障定位。随后，强大的大语言模型（例如 Claude-4）会根据故障定位结果和问题描述，尝试修复已识别出的可疑代码片段。在 SWE-bench Lite 上评估时，RepoRepair 以每次修复 0.44 美元的低成本实现了 45.7% 的修复率。在 SWE-bench Multimodal 上，尽管每次修复的成本较高，为 0.56 美元，它仍以 37.1% 的修复率取得最先进性能，展现出在不同问题领域中稳健且经济高效的表现。

附加关键词和短语：仓库级程序修复、代码文档、大型语言模型

ACM参考格式：
Zhongqiang Pan、Chuanyi Li、Wenkang Zhong、Yi Feng、Bin Luo和Vincent Ng。2018年。RepoRepair：利用代码文档进行仓库级自动程序修复。*ACM*软件工程与方法学汇刊 37, 4, 第111篇文章（2018年8月），27页。https://doi.org/XXXXXXX.XXXXXXX

1 引言

GPT-4[22] 和 DeepSeek-V3[9]等大语言模型（LLM）的迅速发展，深刻影响了软件工程，尤其是代码合成[6, 18, 24],、翻译[23, 26, 27],和修复[4, 40, 42]等核心开发活动的自动化。尽管这些进展表明模型在受限场景中表现出色，但在扩展到仓库级软件维护时，仍存在一个关键缺口。此时，目标不再是生成语法正确的代码片段，而是在庞大且相互依赖的代码库中执行语义正确的更改。这不仅需要解析数千行代码，还要辨别架构意图、隐式契约，以及问题报告中往往以非文本形式呈现的线索。因此，模型必须

*通信作者。

作者地址：Zhongqiang Pan，602023720002@smail.nju.edu.cn；Chuanyi Li，lcy@nju.edu.cn；Wenkang Zhong，zhongwenkang97@foxmail.com；Yi Feng，fy@nju.edu.cn；Bin Luo，luobin@nju.edu.cn，新软件技术国家重点实验室，南京大学，中国南京；Vincent Ng，vince@hlt.utdallas.edu，人类语言技术研究所，得克萨斯大学达拉斯分校，得克萨斯州理查森，美国。

特此授权免费复制本作品的全部或部分内容，用于个人或课堂使用，但前提是不得出于营利或商业利益而复制或分发副本，并且副本必须保留本声明及首页上的完整引文。对于本作品中由作者以外的其他方拥有版权的组件，必须遵守其版权规定。允许在注明出处的情况下进行摘要。若要以其他方式复制、重新出版、发布到服务器或重新分发到列表，则必须事先获得具体许可和/或支付费用。请通过 permissions@acm.org 申请许可。© 2018年版权所有归所有者/作者所有。出版权已授权给 ACM。1049-331X/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

克服两个相互交织的障碍：原始代码仓库数据会消耗过多上下文窗口，以及与故障相关的语义信息分散在多个文件和模态中。当前范式仍未能充分应对这些挑战。

为了系统地评估大语言模型解决现实世界软件工程任务的能力，研究人员提出了许多基准测试。在程序修复领域，Jimenez 和 Yang 于2023年和2024年构建了两个基准测试数据集：**SWE-bench**[13] 以及多模态版本 **SWE-benchMultimodal**[47]。这两个数据集都包含真实的 GitHub 问题描述及其对应的代码仓库。自 SWE-bench 推出以来，在其应用最广泛的子集 SWE-bench Lite 上，最高解决率已从最初的 3% 提升至 60%（不包括专有工具）[1]。然而，这些工具在 SWE-bench Multimodal 上的表现明显较差。例如，目前最先进的修复工具 Agentless Lite[2],在 SWE-bench Lite 上的解决率达到 32.33%，但在 SWE-bench Multimodal 上却显著下降至仅 25.34%。这一表现差距主要源于两个因素。首先，许多工具在设计时未充分考虑编程语言泛化，导致其核心模块与 Python（SWE-bench 的后端语言）紧密耦合。例如，AutoCodeRover[48] 提供的大多数工具都高度依赖 Python 特有的程序分析特性，甚至需要预先了解 SWE-bench 中的特定代码仓库。其次，尽管一些工具（如 SWE-Agent[46] 和 Agentless[39]）已摆脱对特定编程语言的依赖，可以跨语言运行，但它们对仓库信息的利用仍停留在表面层面，仅限于目录结构树以及通过搜索检索到的零散代码片段。这个局限在 SWE-bench Multimodal 中尤为突出，因为其中的问题通常更加复杂，且可能涉及多个文件，仅凭表层仓库信息，这些工具难以准确定位所有可疑代码片段。这些局限共同指向一个根本挑战：缺乏一种结构化的语义表示来描述代码仓库，从而有效弥合高层次问题描述与底层代码变更之间的差距。

为应对自动程序修复中实现对代码仓库进行深入语义理解这一根本挑战，我们提出了 **RepoRepair**，一种新颖的文档增强方法。我们的工作源于一个关键洞见：在软件开发中，代码文档会系统地抽象实现细节，因此是帮助理解和维护程序的重要认知辅助工具[8, 43, 49]。我们认为，利用大语言模型生成此类结构化文档，可以构建有效的中间表示，弥合高层问题描述与修复所需底层代码变更之间的差距。尽管 Luo 和 Ye 展示了大语言模型生成的文档在 Python 仓库中的效用[20], ，我们在此基础上进行了大幅拓展与创新。具体而言，我们提出了一种由两层文档驱动的新型分层故障定位机制：文件级文档用于筛选可疑文件，随后借助函数级文档精确定位可疑类或函数。这种结构化方法使我们的方法能够克服以往工具对代码仓库利用流于表面的问题。**RepoRepair** 分两个阶段运行：**(1)** 在定位阶段，仅使用分层文档实现精确的故障定位；**(2)** 在修复阶段，我们提出了一种新颖的上下文感知流水线。该流水线首先精简已定位代码，构建依赖关系保持型修复上下文；随后生成最小化、采用 diff 格式的补丁；最后通过迭代式组合测试策略对其进行验证。这种连贯的两阶段设计使定位阶段获得的深度理解能够

[1]https://www.swebench.com/index.html（访问于2025年9月）[2]
https://github.com/sorendunn/Agentless-Lite

有效转化为正确可靠的代码变更，在严格控制成本的同时最大限度地提高修复准确率。

为评估**RepoRepair**的有效性，我们在两个基准套件上开展了全面实验：广泛采用的SWE-bench Lite，以及更具挑战性的SWE-bench Multimodal。我们的实验流程包括：（i）使用开源tree-sitter库[3]解析源文件；（ii）通过大语言模型生成分层文档；（iii）在定位与修复过程中处理问题描述（包括多模态内容）。在SWE-bench Lite上，**RepoRepair**取得了具有竞争力的45.7%修复率，平均每次修复成本仅为0.44美元，同时文件级故障定位准确率高达77.6%。我们的**RepoRepair** 在复杂的SWE-bench 多模态基准上优势更为显著。**RepoRepair** 以出色的成本效率（每次修复0.56美元）取得了37.1%的最先进修复率。至关重要的是，其文件级故障定位准确率达到59.8%，相较此前最佳工具Agentless Lite绝对提升近30%。定位能力的这一显著提升直接支撑了更优的修复表现，表明我们的方法恰恰在先前方法难以应对之处表现突出：理解并解决复杂的跨文件问题。这些结果共同验证了我们的文档增强框架在多样编程环境与问题复杂度下的泛化能力、有效性与实际效率。

本研究的主要贡献有三点：

- **文档增强修复框架**：我们提出了 RepoRepair，这是一种新颖的无代理框架，它（1）利用层次化代码文档，显著提升基于大语言模型、根据问题描述进行故障定位的效果；（2）实现了高效的修复流程，通过组合式补丁验证和基于温度的迭代生成，确保稳健的补丁生成。
- **全面的性能基准评测**：在 SWE-bench 和 SWE-bench Multimodal 基准测试上与既有基线进行系统比较，不仅评估最终解决率，还量化了定位阶段文件检索和文件定位准确率的差异。
- **可用的研究材料**：为促进研究的可复现性和后续工作，我们在 GitHub 代码仓库 https://github.com/ZhongQiangDev/RepoRepair 中公开发布了所有实验材料（生成的代码文档、源代码和结果）。

本文其余部分安排如下。第 2 节回顾相关工作。第 3 节详细介绍 RepoRepair 的设计。第 4 节介绍实验设置，第 5 节展示实验结果。第 6 节讨论有效性威胁，第 7 节概述局限性。最后，第 8 节总结全文并提出未来工作的方向。

2 相关工作

2.1 仓库级自动化程序修复

在自动程序修复（APR）领域，Defects4J[15] 长期以来一直被视为标准基准。然而，随着 Jimenez 和 Yang[13] 于2024年推出 SWE-bench 基准，研究重点逐渐从传统的函数级缺陷修复任务转向更具挑战性的仓库级程序修复。值得注意的是，许多在 Defects4J 上表现出色的自动程序修复技术，例如 AlphaRepair[41]和 RepairLLaMA[29]，都专门用于修复已知存在缺陷的文件中已识别出的函数。这些架构并不适用于仓库级程序修复，因为后者不仅需要生成补丁，还需要在整个代码仓库中准确定位文件，任务复杂度因此大幅提高。

[3]https://github.com/tree-sitter/tree-sitter

早期的仓库级自动程序修复探索通常将定位与修复这两项挑战分开处理。在定位方面，FuseFL[37] 等工作将传统故障定位方法（如基于频谱的故障定位）与大语言模型相结合，以生成推理过程；而 LLMAO[44]则通过微调代码模型，直接进行行级定位。相反，在修复方面，RLCE（仓库级上下文提取）[7] 等方法着重检索相关代码片段，以增强基于大语言模型的补丁生成能力，并在其自行构建的 RepoBugs 基准上展现出更好的性能。然而，这些方法通常假定相关文件已预先确定，或者采用了端到端约束较宽松的设置。SWE-bench 基准的出现，将真实世界的 GitHub问题与完整代码仓库配对，正式确立了对真正集成系统的需求：这类系统须从头开始同时完成定位和修复。Jimenez 和 Yang[13]的开创性工作本身提出了一种检索与修复基线，但初期结果较低（在 SWE-bench Lite 上为 3%），凸显了这一任务的难度。这一挑战推动了更复杂方法的发展，这些方法大致可分为基于代理和无代理两种范式。

随着代理框架的兴起，Jimenez 和 Yang 提出了 SWE-Agent[46], ，这是首个基于代理的仓库级自动程序修复方法。SWE-Agent 使用代理-计算机接口（ACI）来浏览代码仓库、编辑文件并运行测试。与此同时，Zhang 等人提出了 AutoCodeRover[48], ，通过提取结构规范来指导代理。在此之后，多代理框架相继涌现：CodeR[5] 使用专业代理通过任务图协同工作，而 SWE-Search[3] 则利用蒙特卡洛树搜索（MCTS）[30] 进行动态规划。近期，OpenHands[33], 、OpenHands-Versa[31], 和 ExpeRepair[21] 等性能更强的基于代理的方法展现出了卓越的能力。然而，基于代理的方法存在固有局限：它们可能不加批判地接受误导性反馈，并因交互轮次过多而产生高昂的计算成本。

为缓解这些问题，Xia 和 Deng 提出了 Agentless[39], ，这是一种无代理自动程序修复方法，采用精简的两阶段定位-修复流程，无需大语言模型决定操作或使用复杂工具。令人意外的是，Agentless 的性能可与许多基于代理的工具媲美，甚至更胜一筹，同时大幅降低了成本，证明了无代理方法的可行性。最近，GUIRepair[11] 专为 SWE-bench 多模态基准提出。它引入了专门的 Image2Code 和 Code2Image 模块来处理视觉信息（例如屏幕截图），并提高前端问题的修复准确率，进一步推动了无代理范式的发展。Agentless 和 GUIRepair 等方法依赖原始代码检索或视觉处理，而我们提出的 **RepoRepair** 利用分层生成的代码文档来捕获文件间的语义关系，从而实现更准确的定位与修复，尤其适用于需要全局仓库理解的问题。它通过结构化文档强化语义表示，为无代理范式带来了独特进展。

2.2 自动代码文档生成

在软件工程领域，早期的自动化文档生成方法[2, 10, 12, 16, 32, 34]旨在为源代码生成描述性摘要，从而减轻开发人员的维护负担。然而，这些方法存在一个关键局限：它们只关注总结孤立的代码片段，却忽略了更广泛的代码仓库级上下文中的依赖关系，因此往往导致摘要质量较差。

为解决这一局限，Luo 和 Ye 提出了 RepoAgent[20], ，这是一种基于智能体的自动化代码文档框架，旨在主动生成、维护和更新 Python 仓库中的代码文档。RepoAgent 集成了三个核心模块：全局结构分析（用于捕获整个代码仓库的依赖关系）、文档生成和文档更新。盲测偏好评估表明，评估者更青睐 RepoAgent 生成的文档，而非人工撰写的文档。类似地，Yang 等人

引入了 DocAgent[45], ，这是一种多智能体系统，通过按拓扑顺序处理代码并模拟人类工作流来应对同一局限。DocAgent 使用五个专门的智能体——Reader、Searcher、Writer、Verifier 和 Orchestrator——协同为 Python 仓库生成高质量的代码文档。

尽管这些工作展示了自动生成仓库感知文档的可行性，但它们主要是作为通用文档工具设计的。相比之下，我们将这一范式重新用于特定的下游任务——自动程序修复，并进行了大幅调整。在大语言模型生成的文档这一基础理念之上，我们设计了一种基于提示的生成方法，可生成专为故障定位优化的分层文档（函数级和文件级）。这种任务驱动的设计确保文档能够捕捉修复指导所需的精确语义和依赖信息，从而直接提升整个流程的有效性，同时改善其可扩展性和跨语言适应性。

3 REPOREPAIR

图 1 展示了我们提出的方法 RepoRepair 的整体概览，该方法包含三个关键阶段：**文档生成**、**定位**和**修复**。在**文档生成**阶段，我们使用大语言模型为目标代码仓库中的所有代码文件生成函数级和文件级代码文档。在**定位**阶段，我们首先执行基于文本的检索，找出文件级代码文档与问题描述语义相似度较高的文件。接下来，我们根据问题描述和生成的文件级代码文档，让大语言模型定位并排列最有可能需要修改以解决问题的可疑文件。由于这些文件中并非所有内容都需要编辑，我们还会根据问题描述和函数级代码文档，让大语言模型定位每个文件中需要修改的函数/类。在**修复**阶段，根据每个文件中定位到的函数/类，我们首先删除其余无需修改的函数/类中的内容（代码和注释），并精简文件内容，以构建最小化的修复上下文。然后，我们让大语言模型修改每个文件的内容，并生成差异格式补丁，以尝试解决问题描述中所述的问题。最后，我们验证生成的补丁，并从单个或多个文件的补丁中选出最合理的补丁作为最终提交。RepoRepair 的完整工作流程以算法1形式化表示，各阶段将在后续小节中详细介绍。

3.1 文档生成

如前所述，代码文档系统地描述实现逻辑，有助于开发人员理解功能。这对于需要深入理解代码的程序修复任务至关重要。然而，为整个代码仓库手动创建高质量的代码文档需要耗费大量时间、人力和财力资源[49]。更重要的是，随着代码仓库不断演进，代码文档也需要同步更新，但大多数项目缺乏足够的资源或动力来确保文档实时更新。

为克服这一障碍并实现可扩展的自动化代码理解，我们采用了一种策略，为整个代码仓库生成全面、层次化的代码文档。尽管这一基础步骤计算成本高昂，但它有两个关键作用。首先，它为后续定位阶段提供完整且一致的知识库，确保跨文件分析时不会遗漏任何可能相关的代码——这也是大型代码仓库面临的常见挑战。其次，通过在代码仓库不断演进的过程中增量生成文档（本节稍后将详细介绍），我们能够将初始生成成本分摊到更长时间内，这符合现实开发工作流中按顺序处理问题的情况。这种方法受到 RepoAgent 方法的启发[20], ，从一般的代码仓库理解转向

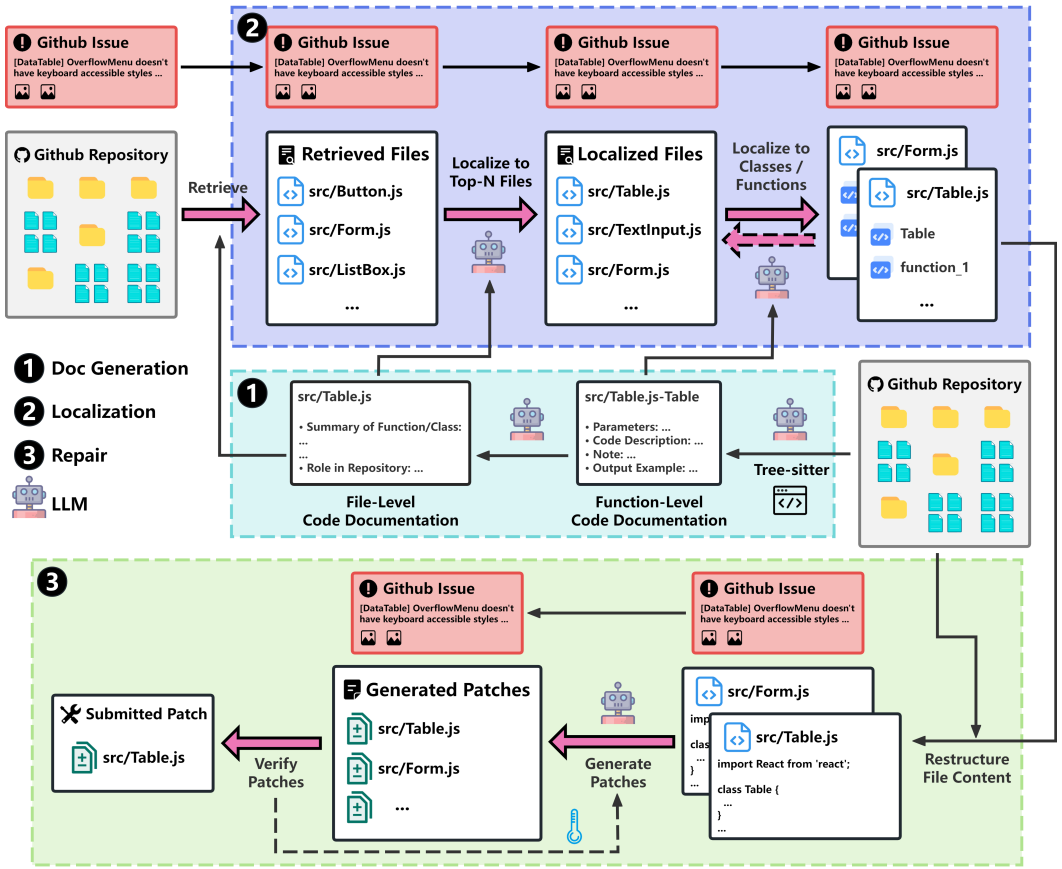


图 1. RepoRepair 概览。

一种有针对性的层次化文档系统（函数级和文件级），专门用于指导精确的故障定位和修复。

生成函数级文档。在生成文档之前，我们首先将所有源文件转换为统一的中间表示。借助 Tree-sitter 及其多语言语法，我们将代码解析为抽象语法树，以系统地提取关键程序结构——函数、类、导入语句和调用，同时剔除具体的语法细节。这种规范化表示将文档生成与特定语言的语法解耦，使我们能够在 Python、JavaScript、TypeScript 及其他受支持的语言中一致地提取语义信息。随后，我们从解析后的抽象语法树中提取所有函数/类的关键信息，包括名称、签名、上下文、依赖关系等。对于 JavaScript/TypeScript 中的匿名函数（如箭头函数或函数表达式），我们会按 function_X 的格式为其指定标准化名称，其中 X 是唯一编号。此过程确保生成的文档包含完整的结构元数据，为后续故障定位提供可靠基础。

随后，为了生成高质量的文档，我们构造了一个提示词（图 2(a)），其中包含目标函数或类的完整源代码及其提取的元数据。我们提供完整实现，而非摘要或代码片段，以确保大语言模型能够获取所有必要的上下文细节（例如内部逻辑、变量用法和控制流）

算法1 RepoRepair

要求： 代码仓库 R，问题描述 I
确保： 修复补丁 P^* 或 $\emptyset$

阶段1：文档生成
D ←生成文档(R)
阶段 2：定位
Q ←将文本化(I) ▷ 将多模态问题转换为文本
F_s ← 定位文件(Q，D)
N_s ← 定位代码单元(F_s，Q)
如果 $\mathcal{F}_s = \emptyset$ **或** $\mathcal{N}_s = \emptyset$ **，则**
    返回 $\emptyset$ ▷ 无法定位问题
结束 if
阶段 3：修复
for $T \in [0.0, 0.9]$ **步长** 0.1 **执行**
    P_{all} ← 生成补丁(F_s，N_s，Q T) ▷ 生成 $C(n,\ k)$ 组合补丁
    P_{valid} ← 验证补丁(P_{all}，R)
    如果 $\mathcal{P}_{valid} \neq \emptyset$**则**
        $P^* \leftarrow \arg\min_{P \in \mathcal{P}_{valid}}$修改文件数($P$)
        返回 $\mathcal{P}^*$ ▷ 返回用于最终提交的补丁
    结束 if
结束 for
返回 $\emptyset$ ▷ 修复失败

以生成准确且信息丰富的描述。随后，大语言模型会处理此输入，以生成标准化的代码文档，其中包含：参数介绍、代码描述、使用说明和输出示例，格式如图 1 所示。

生成文件级文档。在为所有文件生成函数级代码文档后，我们继续生成文件级代码文档。这一更高层次的抽象将重点从实现细节转向总结每个文件在代码仓库中的整体用途和架构角色——这些信息对于定位阶段最初的文件检索和排序步骤至关重要。

该过程需要两个关键输入：(1) 文件的所有函数级代码文档，以及 (2) 项目的结构树，该结构树提供了有关文件在代码库层级结构中的位置及其关系的重要上下文。我们通过提示词（图 2(b)）将这些元素输入大语言模型，指示其生成全面的文档，其中包括：(1) 对每个函数/类功能的简洁总结，以及 (2) 对文件在项目中的架构角色和依赖关系的高层次描述。生成的文档格式如图 1 所示，同时捕捉每个文件的实现细节和架构意义，从而为后续的定位与修复任务提供更有效的代码库级理解。

3.2 定位

要实现缺陷修复或新功能，对源代码进行准确定位从根本上至关重要——没有正确定位，就无法做出正确的修改。然而，挑战在于代码仓库可能包含数百甚至数千个代码文件，每个文件都可能包含数千行代码，而实际的编辑位置可能只是某个特定函数/类中的几行。为应对这一挑战，我们采用分层三步定位流水线：（i）检索与问题高度相关的文件，（ii）识别

Prompt Text for Generating Code Documentation

Your task is to generate documentation for a given code object. The goal is to help developers understand the functionality and proper usage of the code.
You are required to create documentation for a **{code_type}** named "**{code_name}**".

The code content is as follows:
{code_content}

{reference_letter}

Please produce a detailed explanatory document for this object based solely on the provided code.

Output Requirements:
Begin with a bolded one-sentence description of the function of **{code_name}** in plain text.
Follow with a detailed analysis in plain text (include all relevant details), written primarily in **{language}**.
You may include English terms where appropriate to improve readability—especially for function or variable names.

The standard format is as follows:
{code_name}: The function of **{code_name}** is to XXX. (One sentence only)

{parameters_or_attribute}: The **{parameters_or_attribute}** of this **{code_type}**.
· parameter1: XXX
· parameter2: XXX
· ...

Code Description: A detailed explanation of the **{code_type}**. (Provide a thorough and certain analysis of the code. Please also detail its functional dependencies and interactions with downstream components.)

Note: Important considerations when using this code

Output Example: A sample representation of the value this code may output.

Important:
Do not use any Markdown formatting (including headings, dividers, or bold/italic syntax other than the initial bolded function name).
Write primarily in the target language, but retain English technical terms (e.g., function names, parameters) as needed.

(a) Prompt for Generating Function-Level Documentation

Prompt Text for Generating Code Documentation

Your task is to summarize and generate documentation based on the given code description of one or more functions or classes in the same file.

The structure of the project is as follows:
{project_structure_tree}

The path of this **{file_type}** file is **{file_path}**.

The content of the code descriptions of this **{file_type}** file (sorted by number) are as follows:
{code_description_list}

Please generate a detailed summary document for this **{file_type}** file based on the code description of function or class in it.

The standard format is as follows:
Summary of Function/Class: Summarize each code description by a few sentences.

Role in Repository: State the file's overall purpose in the project, referencing its location and structural context to explain its architectural role.

(b) Prompt for Generating File-Level Documentation

图 2。用于生成代码文档的提示词。

其中的可疑文件，以及（iii）在每个可疑文件中定位与问题相关的特定函数/类。

检索最相关的文件。鉴于代码仓库通常包含数百个代码文件，而可能需要修改的文件只有少数几个，直接输入所有对应的源代码会导致上下文过长，难以处理。为此，我们使用文件级代码文档作为检索代理，而不是使用原始代码。这一设计选择基于两个关键考虑因素：(i) 效率：文档提供了对每个文件用途和关键功能的精炼且语义丰富的表示，与完整源代码相比，可大幅减少词元数量。(ii) 与自然语言查询的对齐：由于问题描述以自然语言编写，将其与结构相似的文档进行匹配（而非与原始语法匹配）有助于实现更准确的语义检索。具体而言，我们首先使用嵌入模型将所有文档向量化并构建索引，然后将问题描述作为查询，以检索最相关的文件。

受以往多模态检索工作的启发[19]，我们采用一种策略，在与纯文本代码文档数据库进行匹配之前，将所有查询端信息（包括屏幕截图等视觉细节）统一转换为文本表示。这一设计有两大优势：（i）无需依赖通用多模态嵌入（例如 CLIP[25]），因为这类嵌入可能并未针对捕捉前端特定的代码与视觉关系进行优化；（ii）

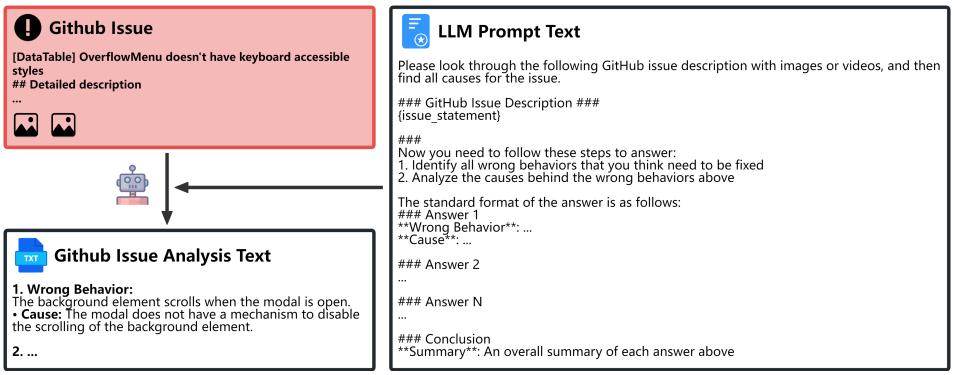


图3. 用于检索的多模态问题描述处理。

在查询处理过程中，将复杂的跨模态对齐任务交由强大的大语言模型完成，无需在检索时实时计算多模态相似度，从而提高效率。具体而言，对于多模态问题描述，我们使用多模态大语言模型生成对问题的全面文本分析，描述所有可能的错误行为和根本原因（如图 3 所示）。生成的文本随后用作检索查询。

定位可疑文件。为从检索结果中识别可疑文件，我们继续向大语言模型提供文件级代码文档，而非完整源代码。这是对我们以效率为重点的设计思路的有意延续。虽然提供完整代码或许能带来略多一些上下文，但随之激增的令牌数量会严重限制我们在预算约束下实际能提供给大语言模型分析的文件数量。文档经过提炼，涵盖了架构角色和关键功能，为大语言模型提供了充分的高层次上下文，使其能够推断哪些文件最有可能包含故障；正如我们的示例中运用思维链（COT）[36] 推理所展示的那样。在分析问题“carbon-design-system/carbon-3918”（“DataTable 内的 OverflowMenu 键盘导航失效”）时，思维链过程首先识别核心错误行为（“嵌入 DataTable 的 OverflowMenu 在键盘导航时没有视觉指示”），随后分析根本原因，例如“DataTable.js”中缺少键盘事件处理程序，或“OverflowMenu.scss”中的焦点状态样式存在冲突，最后将这些原因与相关文件关联起来。这种方法确保定位决策基于对组件交互的深入理解，而非肤浅的关键词匹配。

定位到相关函数/类。在识别出可疑文件后，我们进一步定位需要修改的具体函数/类。此处，我们再次使用函数级代码文档作为输入上下文。我们有意避免像某些先前工作[39]，那样使用更简洁的“文件骨架”（例如仅包含函数签名），因为此类骨架无法捕获匿名函数，且缺乏实现上下文，而这些对于在 JavaScript/TypeScript 代码库中进行准确定位至关重要。相反，提供所有函数的完整源代码在计算上代价过高。函数级文档达到了最佳平衡：它比完整代码紧凑得多，但在语义细节上又足够丰富（描述逻辑、参数和行为），从而能够精确指出相关代码单元。

值得注意的是，函数级定位阶段还可作为对前序文件级定位进行验证的重要机制。如果该过程未能在可疑文件中识别出任何相关函数/类，则表明先前的文件定位可能存在偏差。在

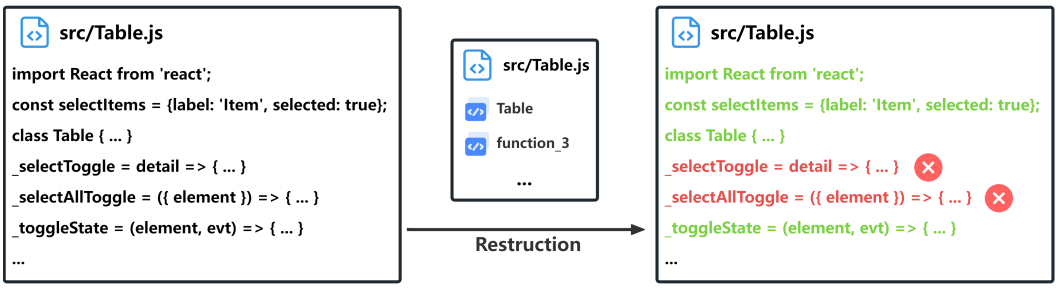


图 4. 代码文件剪枝。

这种情况下，我们的方法会自动舍弃当前这组可疑文件，并重新执行文件定位流程，以获得更可靠的结果。

3.3 修复

在修复阶段，我们的最终目标是生成能够有效解决问题的正确补丁。构建有效的修复上下文面临双重挑战：既要包含足够的语义信息，以指导正确的代码修改；又要保持简洁，避免无关内容使大语言模型不堪重负。传统方法往往采用任意截断或保留整个文件，这两种做法都可能掩盖关键依赖关系或引入噪声。

为解决这一问题，我们实现了一种依赖感知代码剪枝策略（如图 4 所示），不再简单地进行上下文窗口截断，而是根据语义相关性和静态引用有选择地保留代码。这一设计体现了审慎的权衡，在前端和多文件修复场景中尤为有效。具体而言，我们保留三类内容：（i）所有被识别为主要编辑目标的已定位函数或类；（ii）提供结构脚手架的必要文件级构造，例如 import/require语句和全局变量声明；（iii）任何被保留的修复目标通过静态方式引用的未定位函数或类，从而确保文件内依赖完整性。

至关重要的是，我们的策略采用依赖感知保留规则来处理数据流与控制流问题：除了保留已定位代码外，我们还会保留同一文件中被该代码静态引用的所有未定位函数或类。这样可以确保直接依赖的文件内依赖得以保留，从而直接降低遗漏相关上下文的风险。与之相对，我们会激进地删除所有其他代码，具体而言，就是删除未定位且未被引用单元的实现。我们明确选择这种权衡，以提高效率并聚焦重点：虽然保留完整的传递依赖在理论上可能实现完备性，但这会导致上下文爆炸（尤其是在模块化前端代码库中），使大语言模型的注意力被间接相关代码分散，并抵消定向定位带来的效率提升。我们的评估证实，对于典型的基于UI的问题，这三部分——已定位代码、其直接依赖以及文件级脚手架——能够提供足够丰富且聚焦的上下文，在严格控制计算成本的同时，最大限度地提高修复准确率。

生成补丁。我们依次将经过裁剪的代码文件输入大语言模型，由其根据问题描述为每个文件生成 diff 格式补丁，而不是生成完整的替换代码片段。具体而言，这些差异补丁由搜索/替换编辑对[4]组成，其中包含两个关键部分：(i) 搜索片段（要修改的原始代码）和 (ii) 替换片段（新代码）。要将此类补丁应用于原始代码文件，只需

[4]https://aider.chat/

找到搜索片段，并将其替换为相应的替换片段。与完整代码生成相比，这种差异补丁有三大显著优势：首先，它能将大语言模型的注意力限制在最小范围的针对性修改上，而非完整重写；其次，通过减少输出的大小和复杂度，大幅提升成本效益；第三，它能显著降低从头生成大段代码时常出现的幻觉，从而提高可靠性和准确性。

验证补丁。为确保生成的补丁正确且可靠，我们采用系统化的迭代验证流程，并整合传统的软件工程测试实践。该工作流程详见算法1，步骤如下：

1. 每次迭代的补丁收集：对于给定的问题，大语言模型生成候选补丁后（每个定位文件对应一个补丁），我们不会逐一验证，而是根据同时修改的文件数量，将补丁汇总为逻辑分组。具体而言，在每轮验证中，我们收集跨越 n 个定位文件的所有可能补丁组合：先收集单文件补丁（$C(n,\ 1)$组），再收集双文件组合（$C(n,\ 2)$组），直至将全部 n 个补丁一并应用（$C(n,\ n)$组）。这种全面的组合方法使我们不仅能够评估单独的修改，还能评估多个文件之间可能产生的协同效应或冲突。

2. 批量验证：在单次迭代中收集的所有补丁组合，都会针对代码仓库现有的测试套件并行验证，以执行回归测试 [38]。在通过全部测试的补丁组合中，我们采用最小改动准则：选择修改文件数量最少的补丁组合。形式上，我们选择 $P^* = \arg\min_{P \in \mathcal{P}_{\text{pass}}} |\text{Files}(P)|$，其中 $\mathcal{P}_{\text{pass}}$ 表示所有通过验证的补丁组合的集合。偏好较小改动符合软件维护最佳实践，即优先采用局部修复，以降低产生意外副作用的风险。如果当前批次中没有任何补丁组合通过测试，则丢弃整个批次。

3. 基于温度的迭代：如果当前批次中的补丁组合均未通过所有测试，我们就启动新一轮修复迭代。在之后的每轮迭代中，我们将大语言模型的采样温度递增 0.1（从 0 开始），以便在生成的补丁中引入可控变化，鼓励产生多样化的解决方案。该循环将持续进行，直到找到有效的补丁组合或温度达到 0.9。迭代式温度退火策略兼顾了对修复空间的探索与确定性收敛的需求。

4 实验设置

4.1 数据集

为严格评估 RepoRepair 的有效性和泛化能力，我们在两个代表现实世界修复不同方面的基准测试上开展实验：SWE-bench Lite 和 SWE-bench Multimodal。这项双重评估检验了我们的文档增强方法能否同时应对传统的、以代码为中心的错误，以及现代 Web 开发中不断涌现的挑战——后者需要理解视觉用户界面行为。SWE-bench Lite 包含 323 个问题（“dev”子集中有 23 个，“test” 子集中有 300 个），这些问题来自 Python 仓库，问题描述纯为文本，且修改通常仅涉及单个文件。相比之下，SWE-bench Multimodal 包含 619 个问题（“dev”子集中有 102 个，“test”子集中有 517 个），来自 JavaScript/TypeScript 仓库；其问题描述包含多模态信息（文本和视觉元素），且通常涉及更复杂的跨文件更改。

这两个基准测试的主要区别在于编程语言多样性、问题描述模态和修复复杂度。SWE-bench Lite 专注于仅提供文本描述的 Python 问题，而 SWE-bench Multimodal 则将评估扩展到了 JavaScript/TypeScript

环境，并提供丰富的多模态上下文。实证观察证实，在 SWE-bench Lite 上表现出色的方法，在 SWE-bench Multimodal 上的表现始终显著较低，这表明后者本身更具挑战性。

此外，鉴于 SWE-bench Lite 的推出已有一年多时间，而大语言模型的快速迭代不可避免地带来了数据泄漏问题，仅限于 SWE-bench Lite 的评估正变得越来越不充分。因此，同时在两个基准测试上开展全面评估——尤其是在更新且更具挑战性的 SWE-bench Multimodal 上进行评估——能够更严格、更切合实际地衡量程序修复能力。

4.2 基线

为确保评估公平且全面，我们建立了一套系统的标准，用于选择基线工具。我们的选择旨在从三个关键维度反映最先进水平：泛化能力、架构相关性和可复现性。

通用基线。为确保我们的比较能够反映跨基准测试的泛化能力，我们纳入了所有正式在 SWE-bench Lite 和 SWE-bench Multimodal 上接受评估的三种工具：RAG[13], SWE-Agent[46], 和 Agentless[39]。这确保了我们的主要基线能够处理传统的纯代码问题和新兴的多模态场景，为跨数据集性能分析提供共同基础。

架构相关基线。在架构相关性方面，我们选择 Agentless Lite（Agentless 表现最佳的变体）作为主要架构基线。这一选择基于其精简的两阶段流程：首先通过嵌入模型检索相关文件，然后根据排名靠前的检索文件迭代生成修复方案。该流程与我们的方法（检索增强、上下文感知修复）共享关键设计原则，同时保持了计算效率。通过这一有针对性的比较，我们能够将自身具体贡献的影响与一个理念相似但实现方式不同的基线进行区分和分析，例如分层文档和代码剪枝。

基准测试专用基线。为确保在每个基准测试生态系统中进行全面评估，我们纳入了各数据集特别报告的所有主要且可复现的开源工具。对于 SWE-bench Lite，这包括六种额外工具：AutoCodeRover[48], OpenHands[33],PatchPilot[17], DARS[1], ExpeRepair[21], SWE-Search[3],和 SpecRover[28]。对于 SWE-bench Multimodal，我们纳入了两种额外工具：OpenHands-Multimodal[31] 和 GUIRepair[11]。这确保我们的评估涵盖各领域已发表解决方案的全貌，避免选择偏差，并能够全面评估这些工具在语言特定功能、多模态适配和修复复杂度方面的能力。

4.3 RepoRepair 的详细设置

在 RepoRepair 的文档生成阶段，我们使用 DeepSeek-V3[9] (DeepSeek-V3-0324) 生成函数级和文件级代码文档。为确保输出具有确定性且内容准确，我们将采样温度设为 0，其余参数均保持默认值。在定位和修复阶段，我们使用多模态大语言模型 Claude 4 (claude-4-sonnet-20250514)，同样将采样温度设为 0，并固定随机种子，以确保输出稳定且可复现。对于输入多模态大语言模型的动态视觉输入（例如 GIF 和视频），我们通过计算连续帧之间的结构相似性指数（SSIM）[35] 来提取关键帧。这一预处理步骤在保留关键信息的同时，能够高效地表示时序视觉信息。

在文档生成阶段，我们将代码仓库中的所有议题视为其在不同时期演化状态的体现，而非彼此独立的项目。我们的方法会严格按照 GitHub 分配的 ID 升序处理议题（该顺序与时间顺序一致）

并实施增量代码文档生成。如果存在先前议题的代码文档，我们只为与先前议题状态不同的文件生成文档。这一策略有两大优势：通过避免为未更改的文件重复生成文档来优化令牌用量，并准确反映现实中的仓库维护工作流。尽管这一机制在初期可能产生较高的令牌成本，但随着已解决议题数量增加，平均每个议题的成本会逐渐下降，从而显著提升长期效率。

在定位阶段，我们使用 sentence-transformers/all-mpnet-base-v2[5]model 作为嵌入模型，通过检索流程将向量化表示存储在 FAISS[14] 数据库中。对于每个问题描述，我们根据语义相似度检索排名前 50 的相关文件。两个实际因素会影响最终检索到的文件数量：（i）如果排名前 50 的结果中出现重复文件，去重会减少最终数量；（ii）当问题描述通过多个查询处理时，各查询结果的并集可能超过 50 个文件。检索完成后，我们会从检索结果中定位最可疑的前 5 个文件，然后定位其中所有可能相关的函数/类，数量不设限制。

在修复阶段，我们采用第 3.3 节所述的迭代验证方法。为兼顾修复探索与计算效率，我们将大语言模型的温度设为每次迭代递增 0.1，直至达到 0.9。每批中的所有补丁组合都会针对代码仓库现有的测试套件并行验证，每个进程的超时阈值为一分钟。如果某个补丁组合的验证时间超过此限制，则将该组合视为失败；该批次中其他组合的验证则独立进行。

4.4 评估指标

在 SWE-bench Lite 和 SWE-bench Multimodal 上进行评估时，我们采用了以往研究中确立的指标 [39, 47, 48]：**(1) 解决率** - 基准测试中成功修复的问题所占的百分比。值得注意的是，对于代码仓库级评估，所有补丁都通过自动化测试套件进行验证；任何通过测试的补丁都被视为正确，不再使用“合理补丁”这一概念。**(2) 平均成本** - 每次工具运行的平均推理成本。此外，我们还引入了**正确定位百分比** - 工具定位的文件与开发者补丁中修改的文件完全匹配的案例所占的百分比。我们在定位阶段衡量这一指标；若 RepoRepair 定位出的文件集合完整包含所有开发者补丁涉及的文件，则视为定位正确。这是一种以召回为导向的指标，先前研究[39] 曾用它来评估工具能否成功识别所有相关故障位置，即使这可能导致纳入一些假阳性。对于基线工具，我们采用其代码仓库中官方报告的测试结果。

5 实验结果

5.1 整体性能

表1展示了所有方法在 SWE-bench Lite 基准测试上的整体性能指标（解决率和平均成本），表2则展示了 SWE-bench 多模态基准上的相应结果。表3和表4针对两个数据集，对 RepoRepair 与 Agentless Lite 和 GUIRepair 进行了细致比较，具体考察了不同代码仓库上的修复成功率（解决率）和定位准确率（正确定位百分比）。

与基线工具的性能比较。表 1 和表 2 分别全面比较了各方法在 SWE-bench Lite 和多模态基准测试上的修复性能（解决率）。在 SWE-bench Lite 上，RepoRepair 的解决率达到 45.67%，表现具有竞争力，且成本效益良好（每个问题 0.44 美元）。尽管使用 Claude-4 Sonnet 的 ExpeRepair 取得了

[5]https://huggingface.co/sentence-transformers/all-mpnet-base-v2

表1. 性能比 较：基线工具在 SW E-bench Lite 上的表现。

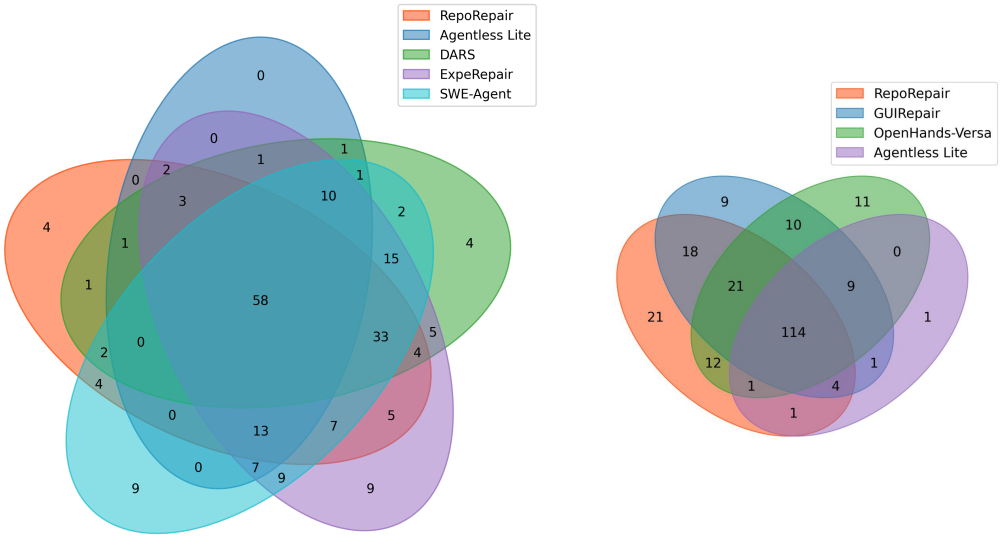
Tool	LLM	解决率	平均成本
RAG	Claude-3 Opus	4.33%	-
SWE-Agent	Claude-3.5 Sonnet	23.00%	1.62
SWE-Agent	Claude-4 Sonnet	56.67%	-
AutoCodeRover	GPT-4o	30.67%	0.65
Moatless Tools	Claude-3.5 Sonnet	38.33%	0.17
Agentless	GPT-4o	27.33%	0.34
Agentless Lite	O3 Mini	32.33%	0.21
OpenHands	CodeAct v2.1	41.67%	2.14
PatchPilot	Claude-3.5 Sonnet	45.33%	0.97
DARS	Claude-3.5 Sonnet	47.00%	12.24
ExpeRepair	Claude-3.5 Sonnet	48.33%	2.07
ExpeRepair	Claude-4 Sonnet	**60.33%**	-
RepoRepair	Deepseek-V3 & Claude-4 Sonnet	45.67%	0.44

表 2. SWE-bench Multimodal 上基线工具的性能比较。

Tool	LLM	解决率	平均成本
RAG	GPT-4o	6.00%	0.17
SWE-Agent	Claude-3.5 Sonnet	12.19%	1.52
SWE-Agent JS	Claude-3.5 Sonnet	11.99%	3.11
SWE-agent M	GPT-4o	12.19%	2.94
Agentless	Claude-3.5 Sonnet	6.19%	0.42
Agentless Lite	Claude-3.5 Sonnet	25.34%	0.38
OpenHands-Versa	Claude-4 Sonnet	34.43%	-
GUIRepair	o4-mini	33.85%	0.36
GUIRepair	o3	35.98%	-
RepoRepair	DeepSeek-V3 & Claude-4 Sonnet	**37.14%**	0.56

最高性能（60.33%），但我们的方案以显著更低的成本，优于若干知名基线，例如 PatchPilot（45.67% 对比 45.33%）、OpenHands（45.67% 对比 41.67%）和 DARS（45.67% 对比 47.00%）。更重要的是，在难度更高的 SWE-bench 多模态基准上，RepoRepair 以 37.14% 的解决率创下最先进性能，超越了包括 GUIRepair（35.98%）、OpenHands-Versa（34.43%）和 Agentless Lite（25.34%）在内的所有现有工具。这些结果表明，RepoRepair 在不同基准测试中均能实现具有竞争力的性能，同时保持较高的成本效益。

独特修复能力。作为整体性能指标的补充，图 5 展示了维恩图分析，揭示了 RepoRepair 的独特能力：修复其他顶尖工具无法解决的漏洞。在 SWE-bench Lite 上，RepoRepair 独自修复了 4 个其他所有四种对比工具（Agentless Lite、DARS、ExpeRepair、SWE-Agent）都未能解决的问题。在 SWE-bench Multimodal 上，这种独特能力更加



(a) SWE-bench Lite (b) SWE-bench Multimodal

图5. SWE-bench Lite（左）和 SWE-bench Multimodal（右）上表现最佳工具的独特问题解决结果。

明显，RepoRepair 解决了 21 个 GUIRepair、OpenHands-Versa 和 Agentless Lite 无法修复的问题。这些独特解决的案例通常涉及描述中几乎没有提及文件的问题，需要深入理解代码仓库结构的语义——这正是 RepoRepair 的优势所在。这项分析表明，RepoRepair 不仅整体性能具有竞争力，还能解决现有方法无法处理的一系列有价值的难题。

此外，我们对 RepoRepair、AgentlessLite 和 GUIRepair 在 SWE-bench Multimodal 上的性能差异进行了比较分析。如表3所示，与 GUIRepair（35.98%）和 Agentless Lite（25.34%）相比，RepoRepair 的整体性能最高（37.14%）。值得注意的是，不同类型代码仓库的性能表现差异显著。RepoRepair 在文本/代码导向型代码仓库（如 'bpmn-io/bpmn-js' 和 'eslint/eslint'）中表现尤为出色。这些代码仓库主要涉及代码逻辑、API 使用和配置问题，而 RepoRepair 能有效地将问题描述关联到相关代码元素。相比之下，GUIRepair 利用屏幕截图中的视觉信息进行复现和验证，在视觉组件较多的代码仓库中表现出相对优势。在 'GoogleChrome/lighthouse' 中，它处理视觉信息的能力为涉及网页渲染和可视化的问题带来了切实益处。同样，在 'PrismJS/prism' 和 'prettier/prettier' 这类语法导向型仓库中，GUIRepair 的视觉验证机制也有助于提升其性能。尽管 GUIRepair 在特定视觉领域具有优势，RepoRepair 更优异的总体表现表明，我们的文档增强方法在 SWE-bench Multimodal 中各类软件项目上具有更广泛的适用性；在这些项目中，代码逻辑问题比纯视觉问题更为普遍。

与基线工具的平均成本相比。如表1所示，RepoRepair 在 SWE-bench Lite 上实现了 45.67% 的竞争性修复率，平均每次修复成本为 0.44 美元，成本效益显著高于大多数基于智能体的方法，例如 SWE-Agent（使用 Claude-3.5 Sonnet 时为 1.62 美元）和 ExpeRepair（使用 Claude-3.5 Sonnet 时为 2.07 美元）。在更具挑战性的

表 3. RepoRepair、Agentless Lite 和 GUIRepair 在 SWE-bench Multimodal 各代码仓库上的修复结果（解决率）对比。

代码仓库	Agentless Lite	GUIRepair	RepoRepair
alibaba-fusion/next	6/39	10/39	10/39
bpmn-io/bpmn-js	27/54	37/54	51/54
carbon-design-system/carbon	9/134	24/134	24/134
eslint/eslint	2/11	6/11	8/11
GoogleChrome/lighthouse	4/54	8/54	6/54
grommet/grommet	2/21	5/21	5/21
highlightjs/highlight.js	3/39	3/39	3/39
openlayers/openlayers	73/79	76/79	76/79
prettier/prettier	0/13	4/13	2/13
PrismJS/prism	5/38	8/38	4/38
quarto-dev/quarto-cli	0/24	5/24	3/24
scratchfoundation/scratch-gui	0/11	0/11	0/11
总计	131/517（25.34%）	186/517（35.98%）	192/517（37.14%）

在 SWE-bench 多模态基准（表2）上，尽管 RepoRepair 的成本略高于 GUIRepair 最经济的配置（使用 o4-mini 时，成本为 0.36 美元，修复率为 33.85%），但额外的 0.20 美元成本因 3.28% 的性能提升而物有所值；对于优先考虑修复成功率的实际部署场景而言，RepoRepair 的性价比更高。此外，RepoRepair 通过复用文档的策略进一步提升了成本效益。具体而言，在 SWE-bench Lite 上，仅有 0.12 美元用于使用 DeepSeek-V3 进行代码文档生成；而在 SWE-bench 多模态基准上，由于 JavaScript/TypeScript 仓库规模更大、复杂度更高，这部分成本增至 0.20 美元，其余成本则用于使用 Claude-4 进行定位与修复任务的多模态大语言模型推理。

与基线工具在正确定位百分比方面的比较。在实际软件开发中，准确识别修改位置，除了直接修复缺陷之外，还能为调试提供重要价值。需要注意的是，对于给定问题，可能存在多个有效的修改位置（这些位置可能与开发者选择的补丁不同），但与开发者补丁进行比较，仍是评估定位效果的一种有意义的近似指标。尽管 RepoRepair、Agentless Lite 和 GUIRepair 采用了不同的定位工作流，但这三种方法都包含文件定位这一基本环节（Agentless Lite 和 RepoRepair 还采用了类似的文件检索步骤）。这一共性使得比较它们的文件定位准确率（正确定位百分比）具有实际意义；表 4 和表 5 分别量化评估了它们在不同代码仓库中的 SWE-bench Lite 和 SWE-bench Multimodal 表现。

在 SWE-bench Lite（表 4）上，RepoRepair 的文件检索准确率（282/300，94.0%）与 Agentless Lite（289/300，96.3%）相当，同时文件定位表现（237/300，79.0%）优于 Agentless Lite（224/300，74.7%）。文件检索方面的这一性能差距可能源于我们采用了文档增强检索方法：RepoRepair 直接将问题描述与生成的代码文档进行匹配，而问题描述通常包含代码片段和变量名，它们可能与代码文档中的自然语言抽象并不完全匹配，从而产生影响检索准确率的语义鸿沟。相比之下，Agentless Lite 直接依据代码内容检索文件，因此可能更准确地捕捉到与问题描述中提及的代码特定特征相关的文件。尽管在文件检索阶段略有劣势，

表 4。SWE-bench Lite 各代码仓库中的文件定位结果（正确定位百分比）：RepoRepair 与 Agentless Lite 的对比。（AL = Agentless Lite；RR = RepoRepair）

代码仓库	文件检索		文件定位	
	AL	RR	AL	RR
astropy/astropy	6/6	6/6	5/6	6/6
django/django	109/114	110/114	97/114	100/114
matplotlib/matplotlib	21/23	22/23	10/23	16/23
mwaskom/seaborn	4/4	4/4	2/4	3/4
pallets/flask	3/3	3/3	3/3	3/3
psf/requests	6/6	6/6	5/6	6/6
pydata/xarray	5/5	5/5	4/5	5/5
pylint-dev/pylint	6/6	4/6	6/6	3/6
pytest-dev/pytest	17/17	15/17	16/17	1/17
scikit-learn/scikit-learn	23/23	22/23	15/23	17/23
sphinx-doc/sphinx	16/16	16/16	12/16	12/16
sympy/sympy	73/77	69/77	49/77	65/77
总计	289/300 (96.33%)	282/300 (94%)	224/300 (74.67%)	237/300 (79%)

表 5. RepoRepair、Agentless Lite 和 GUIRepair 在 SWE-bench Multimodal 各代码仓库中的文件定位结果（正确定位百分比）对比。（文件 R = 文件检索；文件 L =文件定位；AL = Agentless Lite；GR = GUIRepair；RR = RepoRepair）

代码仓库	文件检索		文件 L		
	AL	RR	AL	GR	RR
alibaba-fusion/next	23/39	21/39	14/39	13/39	17/39
bpmn-io/bpmn-js	49/54	51/54	15/54	22/54	31/54
carbon-design-system/carbon	119/134	116/134	32/134	67/134	97/134
eslint/eslint	9/11	10/11	5/11	6/11	10/11
GoogleChrome/lighthouse	37/54	39/54	21/54	14/54	27/54
grommet/grommet	16/21	15/21	6/21	8/21	14/21
highlightjs/highlight.js	39/39	37/39	29/39	31/39	33/39
openlayers/openlayers	57/79	59/79	24/79	25/79	45/79
prettier/prettier	10/13	11/13	3/13	4/13	8/13
PrismJS/prism	37/38	24/38	8/38	24/38	16/38
quarto-dev/quarto-cli	11/24	12/24	0/24	15/24	9/24
scratchfoundation/scratch-gui	1/11	6/11	0/11	0/11	2/11
总计	401/517 （77.56%）	401/517 （77.56%）	157/517 （30.37%）	229/517 （44.29%）	309/517 （59.77%）

RepoRepair 在后续的文件定位阶段仍保持更优表现，这表明 RepoRepair 更有能力正确识别需要修改的可疑文件，从而整体上实现更佳的定位表现。

在难度更高的 SWE-bench 多模态基准测试（表 5）中，RepoRepair 在文件检索和定位方面均展现出显著优势。在文件检索方面，RepoRepair 达到

表 6. RepoRepair 在两个基准测试上的函数级定位分析。

指标	SWE-bench Lite	SWE-bench Multimodal
成功的文件定位	237	309
成功的函数定位	205（86.5%）	273（88.35%）
单轮函数定位	222（93.67%）	302（97.73%）
函数定位需要 >1轮	15（6.33%）	7（2.27%）
平均函数定位轮数	1.06	1.02

与 Agentless Lite 相同的准确率（401/517，77.6%），而在文件定位准确率方面则大幅优于两种基线工具：RepoRepair 的准确率为 309/517（59.8%），Agentless Lite 为 157/517（30.4%），GUIRepair 为 229/517（44.3%）。例如，在“carbon-design-system/carbon”代码仓库中，RepoRepair 正确定位到的问题文件比 Agentless Lite 多 65 个，比 GUIRepair 多 30 个。乍看之下，由于方法不同（RepoRepair 使用 Top-5 文件定位，而 Agentless Lite 使用 Top-1 方法），这种比较似乎有失公平，但实际上，我们的评估框架采用了更宽松的成功判定标准，因此更有利于 Agentless Lite：只要其预测的单个文件出现在开发者的补丁集中即可。相比之下，RepoRepair 必须正确识别开发者补丁中修改的所有文件，才会被判定为正确。这一严格的评估方案证实，观察到的性能差异在统计上显著，且方法上公平。

虽然 Agentless Lite 省略了函数级定位，但 RepoRepair 的层级结构通过使大语言模型能够进一步验证可能相关的文件，提供了关键的细化步骤。具体而言，如果在此步骤中未识别出相关函数/类，则该文件会被排除在修复范围之外，即使它最初在文件定位阶段被选中也是如此。此过程实质上对文件级结果进行严格验证，从而避免不必要的修复。我们的评估采用严格标准：凡是函数定位未能识别相关函数/类的文件，均视为定位失败，无论后续修复结果如何。如表 6 所示，尽管 RepoRepair 的函数定位在 SWE-bench Lite 上排除了 32 个有效修复目标（从 237 个降至 205 个），在 SWE-bench Multimodal 上排除了 36 个（从 309 个降至 273 个），但其精确率分别达到 86.5%（205/237）和 88.4%（273/309），证实了它在这两个基准测试中作为提升精确率步骤的有效性。此外，函数定位效率很高，在 SWE-bench Lite 上有 93.7% 的案例在单轮内完成，在 SWE-bench Multimodal 上则为 97.7%；平均定位轮数仅为 1.06 和 1.02。

5.2 消融研究

我们开展系统的消融研究，以评估 RepoRepair 各关键组件的独立贡献，并验证其设计依据。(1) 我们在定位过程中禁用“文件检索”组件，同时保留基于文档的文件和函数定位，以评估该组件。(2) 我们在定位过程中排除“代码文档”组件（同时保留文件检索），转而采用 Agentless 的基于路径的方法，以评估该组件。值得注意的是，对于 SWE-bench Multimodal（匿名函数较多的 JavaScript/TypeScript 仓库），省略文档会导致这些关键元素也无法使用；我们因此完全跳过函数定位，直接进行修复。而对于 SWE-bench Lite（匿名函数较少的 Python 仓库），我们则保留完整的层级定位流程。(3) 我们通过移除上下文截断步骤，同时保留完整的代码上下文，来分析“代码剪枝”组件。

表 7. RepoRepair 各变体在两个基准测试上的定位与修复计算成本（词元）。（File R = 文件检索；File L = 文件定位；Func L = 函数定位；CP = 代码剪枝）

	SWE-bench Lite				SWE-bench Multimodal			
	Full	不含 FR	无文档、无 CP		Full	无文件检索	无文档、无 CP	
文件检索	0	-	0	0	1.4M	-	1.4M	1.4M
文件 L	2.9M	44.6M	0.6M	2.9M	23.5M	132.6M	2.9M	23.5M
Func L	6.3M	13.5M	0.3M	6.3M	6.2M	16.5M	-	6.2M
修复	2.2M	5.4M	1.9M	3.8M	8.1M	21.6M	9.0M	12.5M
总计	11.4M	63.5M	2.8M	13.0M	39.2M	170.7M	13.4M	43.6M

表 8. RepoRepair 各变体在两个基准测试上的定位（正确定位百分比）和修复（解决率）性能。（文件检索 =文件检索；文件 L = 文件定位；Func L = 函数定位；CP =代码剪枝）

	SWE-bench Lite				SWE-bench Multimodal			
	Full	不含 FR	不含 Doc	不含 CP	Full	不含 FR	无文档	无代码属性
文件检索	282/300	-	282/300	282/300	401/517	-	401/517	401/517
文件 L	237/300	229/300	225/300	237/300	309/517	282/517	180/517	309/517
Func L	205/300	190/300	159/300	205/300	273/517	249/517	-	273/517
修复	137/300	124/300	111/300	128/300	192/517	183/517	136/517	174/517

(4) 为专门探究文档质量问题，我们针对 SWE-bench Lite 中的 django/django（114 个错误）开展专项研究，使用 Qwen2.5-7B（相较于 DeepSeek-V3 能力较弱的模型）重新生成文件级和函数级文档，并执行完整的定位-修复流水线。此外，我们通过随机抽取 100 个文件及其对应的代码文档进行人工评估，以评估文档质量。

文件检索的影响。如表 7 所示，禁用文件检索会显著增加两个基准测试的计算成本：在 SWE-bench Lite 上，文件定位的令牌用量从 2.9M 激增至 44.6M（增加 15.4× ），总成本则从 11.4M 上升至 63.5M；在 SWE-bench Multimodal 上，文件定位的令牌用量从 23.5M 增至 132.6M，总成本则从 39.2M 增至 170.7M。这种成本的指数级攀升并不意外，因为代码仓库通常包含数百至数千个文件，若不进行初步筛选，定位阶段就必须同时处理所有文件的文档。表 8 进一步表明，尽管处理的信息更多，但未使用文件检索的变体性能反而下降：在 SWE-bench Lite 上，文件定位准确率从 237/300 降至 229/300，修复率从 137/300 降至 124/300；在 SWE-bench Multimodal 上，文件定位准确率从 309/517 降至 282/517，修复率从 192/517 降至 183/517。这些结果表明，过长的上下文会削弱大语言模型推理能力，而文件检索能够有效降低噪声，在降低令牌成本的同时提高定位准确率：在 Lite 上降低 82.1%，在 Multimodal 上降低 82.3%——这对实际部署至关重要。

代码文档的影响。如表 7 所示，移除代码文档会显著降低定位阶段的令牌用量：在 SWE-bench Lite 上，文件定位的成本从 2.9M 降至 0.6M（减少 79.3%），总成本从 11.4M 降至 2.8M；在 SWE-bench Multimodal 上，文件定位的成本从 23.5M 降至 2.9M（减少 87.7%），总成本从 39.2M 降至 13.4M。然而，表 8 揭示了一个关键的性能权衡：

表9。文档质量对 RepoRepair 性能的影响（django/django）。（文件检索 = 文件检索；文件 L = 文件定位；Func L = 函数定位）

文档生成器	文件检索	文件 L	Func L	修复
DeepSeek-V3	110/114 (96.5%)	100/114 (87.7%)	93/114 (81.6%)	67/114（58.8%）
Qwen2.5-7B	98/114（86.0%）	82/114（71.9%）	68/114（59.6%）	39/114（34.2%）

表10. 生成文档质量的人工比较评估（django/django）。

文档粒度	样本对	偏好 DeepSeek-V3	偏好 Qwen2.5-7B
文件级	100	96 (96%)	4 (4%)
函数级	627	615 (98.1%)	12 (1.9%)

如果没有代码文档，定位准确率会大幅下降——在 SWE-bench Lite 上，文件定位结果从 237/300 降至 225/300，修复率从 137/300 降至 111/300；在 SWE-bench Multimodal 上，影响更为严重，文件定位结果从 309/517 骤降至 180/517，修复结果则从 192/517 降至 136/517。这种显著下降表明，尽管代码文档会增加计算成本，却能为大语言模型提供超越表层代码分析所必需的语义理解。文档以结构化方式呈现功能、依赖关系和关联，使相关代码元素能够被更准确地识别；对于具有复杂匿名函数和回调函数的 JavaScript/TypeScript 仓库而言，这一点尤为重要。性能与成本分析证实，代码文档对于实现切实有效的修复效果不可或缺。

代码剪枝的影响。表 7 显示，禁用代码剪枝会适度增加计算成本，尤其是在修复阶段：在 SWE-bench Lite 上，修复阶段的令牌用量从 2.2M 增至 3.8M（增幅为 72.7%），总成本从 11.4M 增至 13.0M；在 SWE-bench Multimodal 上，修复成本从 8.1M 增至 12.5M（增幅为 54.3%），总成本从 39.2M 增至 43.6M。这一增长源于在补丁生成期间向大语言模型提供完整且未经剪枝的代码上下文。表 8 揭示了一项重要的性能规律：在 SWE-bench Lite 上，消融代码剪枝后，文件和函数定位准确率保持不变（分别为 237/300 和 205/300），但修复率从 137/300 降至 128/300；在 SWE-bench Multimodal 上，定位准确率同样保持不变（309/517 和 273/517），但修复率从 192/517 降至 174/517。这些结果表明，完整的代码上下文虽然提供了全面的信息，但也会引入大量噪声，使大语言模型难以专注于核心缺陷修复逻辑。代码剪枝组件能够有效过滤无关细节，从而在不牺牲定位准确率的前提下生成更精确的补丁。这表明，相较于提供尽可能多的上下文，有策略地缩减上下文更有利于优化修复流程，尤其适用于需要集中注意力处理特定代码片段的复杂缺陷。

文档质量的影响。为探究文档生成能力如何影响 RepoRepair 的性能，我们使用两种不同的大语言模型（DeepSeek-V3 和 Qwen2.5-7B）生成文档，并开展了有针对性的消融研究。如表9所示，在 django/django 上使用 Qwen2.5-7B 生成文档，会导致定位流程各阶段的性能显著下降：文件检索准确率从 96.5% 降至 86.0%，文件定位率从 87.7% 降至 71.9%，函数定位率从 81.6% 降至 59.6%。定位性能的这种连锁下降直接导致

修复率从 58.8% 降至 34.2%，这表明文档质量对定位过程能否成功起着关键作用，而定位过程又决定了修复效果的上限。

为了解这一定位表现差距的根本原因，我们开展了人工比较评估，由两位拥有丰富 Python 开发经验的作者（研究经验均超过 4 年）在盲测条件下评估文档质量。对于随机抽取的 100 个文件及其对应的 627 个函数/类，评估者分别独立比较两种模型生成的匿名文档，依据以下三项标准，选出对代码功能描述更好的版本：(1) 功能描述的准确性；(2) 所涵盖关键要素的完整性；(3) 表达的清晰度。评估遵循严格的两阶段流程。第一阶段中，两位评估者独立评估所有样本。他们最初的判断在文件级样本中有 92% 一致，在函数级样本中有 96.2% 一致。对于其余存在分歧的情况，我们采用了解决流程：(a) 首先，评估者共同审阅有分歧的案例并讨论各自的判断依据；(b) 若仍无法达成共识，则更深入地查阅源代码；(c) 对于 8 个始终存在争议的案例，最后由 GPT-4o 作为仲裁者提供独立分析（同时对文档来源保持盲态），随后评估者结合这一额外视角重新协商并达成判断。这一多步骤解决流程确保表10中的所有最终判断都经过审慎考虑并达成共识。

如表10所示，DeepSeek-V3 生成的文档显然更受青睐：在文件级和函数级上的偏好率分别为 96% 和 98.1%，而 Qwen2.5-7B 的对应比例仅为 4% 和 1.9%。几乎所有人都更偏好 DeepSeek-V3 生成的文档，这表明 Qwen2.5-7B 生成的文档可能存在不准确、遗漏或语义表示不够精确等问题，进而直接削弱大语言模型在定位过程中准确识别相关代码元素的能力。由于 RepoRepair 仅对已定位的代码片段进行修复，任何妨碍正确定位的文档缺陷都必然会导致修复失败。这一发现强调，高质量文档对于准确定位至关重要——定位是我们方法的基础阶段；同时，这也验证了我们采用先进大语言模型为代码仓库生成精确语义表示的设计选择。

5.3 案例研究

为深入了解 RepoRepair 的文档增强方法的有效性，我们考察了 SWE-bench Multimodal 中一个具有代表性的案例。图6展示了案例“carbon-design-system/carbon-3610”：问题描述指出，React 列表框组件未能向下传递 light 属性。仅凭表层关键词匹配（例如“listbox”），可能会认为需要修改“ListBox.js”——依赖基于代码的定位方法的基线系统便错误地采取了这种做法。相比之下，RepoRepair 借助代码文档进行分层定位，正确地在另一个文件中找到了根本原因：较高层级组件“Dropdown.js”中的属性传递不当。整个过程包括：(1) 首先利用文件级文档选出五个可疑文件，然后 (2) 通过函数级分析将范围缩小到三个文件，最终生成了与开发者修复方案一致的正确补丁。

这个案例展现了 RepoRepair 的核心优势：通过全面的文档，在问题描述与实现代码之间建立语义关联，从而定位那些很少被提及但至关重要的文件。缺乏此类结构化文档时，大语言模型难以超越表面关键词，理解代码间的实际关系，进而导致文件定位和修复均告失败。基线方法在“ListBox.js”中生成的错误补丁（仅添加了一个 CSS 类），凸显了关键词驱动方法的局限性，也印证了 RepoRepair 的文档感知推理对于复杂的、涉及整个代码仓库的问题所具有的价值。

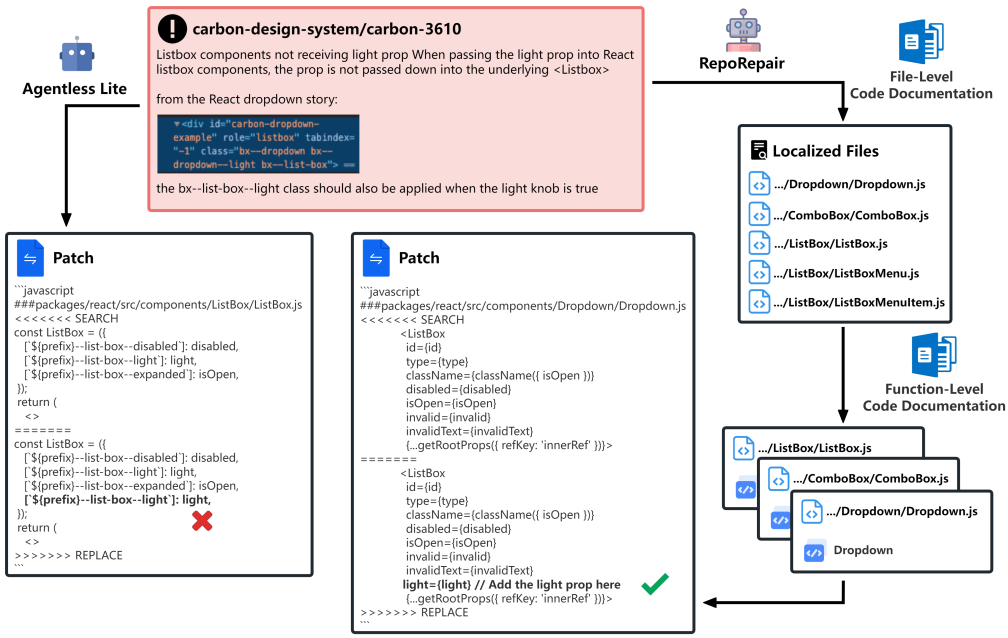


图 6. 案例比较：RepoRepair 与 Agentless Lite 针对问题 carbon-design-system/carbon-3610 生成的补丁。

6 有效性威胁

内部效度。一个主要的内部效度威胁与潜在的数据泄漏有关。尽管 Claude-4 的发布时间晚于 SWE-bench Lite 的构建时间，但它在更新的 SWE-bench 多模态基准上展现出了稳健的泛化能力，减轻了对时间性数据污染的担忧。为严格审查泄漏风险，我们进行了以下检查：（1）从模型输入中移除所有真实上下文信息；（2）确认模型在缺少适当检索上下文时，绝不会输出正确的文件、函数或代码片段。我们的分析发现，SWE-bench Lite 中存在少量泄漏，仅涉及文件定位中的 4 个实例：尽管检索失败，模型仍识别出了正确的文件路径，这可能是由于预训练期间接触过类似的公开代码仓库。SWE-bench 多模态基准中未发现此类泄漏。总的来说，我们认为基准测试污染得到了良好控制，不会对结果的有效性产生实质性影响。

另一项威胁源于模型选择。在各项比较中，我们始终使用 Claude-4，因为竞争性基线（如 SWE-Agent、ExpeRepair）在这些基准测试中广泛采用该模型。消融研究证实，RepoRepair 的性能提升主要源于其文档机制，而非底层模型的选择。

外部效度。主要的外部效度威胁与语言覆盖范围和代码仓库规模有关。我们的评估涵盖了 SWE-bench Lite 和 SWE-bench Multimodal 中两大语言生态系统（Python、JavaScript/TypeScript），但在其他常见语言（如 Java、C#）中的表现尚未得到验证。此外，尽管这些基准测试代表了真实世界的项目，但可能无法体现某些工业代码库的规模或独特模式，而这些环境中可能会出现计算资源限制。

尽管如此，我们在不同语言领域的多个基准测试中取得的结果，表明 RepoRepair 具有良好的泛化能力。

7 局限性

RepoRepair 的设计存在若干局限，限制了其在仓库级程序修复中的广泛适用性：

其代码文档生成机制存在一项根本性局限：由于它仅处理代码文件中的函数/类，不包含功能组件的文件（例如配置文件、静态资源包）无法生成文档。当此类文件对解决问题至关重要时，RepoRepair 很可能会失败。我们的分析在 SWE-bench Multimodal 中发现了 26 个此类案例（4.2%），主要涉及配置文件（例如 prism-sql.js）、静态样式文件（例如 style.css）和入口文件（例如 index.js）。因此，与功能性错误修复相比，RepoRepair 在非功能性变更（例如参数调优、样式变更）方面的效果显著降低。在实际部署中，可以通过互补技术来应对这一局限。例如，可使用专用解析器处理配置文件，以提取结构化信息；维护覆盖整个代码仓库的函数调用图，则可在相关文件发生变更时实现更有针对性的文档更新，避免重新为整个代码仓库生成文档。

此外，RepoRepair 实现的工作流基于这样一个假设：缺陷仅限于对现有文件的修改。因此，该工具无法处理需要创建全新文件的功能请求——这一限制影响了 SWE-bench Multimodal 中的 13 个案例（2.1%）。这并非 RepoRepair 独有的问题，而是所有无代理自动程序修复工具面临的一项根本挑战：无法区分问题类型（修改与创建），并据此调整定位和修复工作流。未来的扩展可以在没有现有文件被判定为相关文件时，加入文件创建阶段，提示大语言模型根据仓库模式生成合适的新文件结构。

此外，我们的增量式文档生成策略会在提交之间复用未发生变更的文件所对应的文档，以提高效率。然而，这种方法可能无法捕捉到周边代码变更后，未变更文件的作用或依赖关系如何演变——这种细微的上下文变化可能会影响定位准确率。通过维护轻量级的依赖关系跟踪，可以更精准地更新文档：不仅为变更的文件重新生成文档，也为架构关系发生变化的文件重新生成文档。

8 结论与未来工作

我们提出了 RepoRepair，这是一种利用代码文档自动修复代码仓库级问题的新方法。遵循无代理方法，RepoRepair 通过三个关键阶段进行修复：分层文档生成、多模态故障定位，以及上下文感知补丁生成。在 SWE-bench Lite 和最新的 SWE-bench 多模态基准上的全面评估表明，我们的方法达到了最先进性能，修复率分别为 45.7% 和 37.1%，同时保持了较高的成本效益，每次修复的成本分别为 $0.44 和 $0.56。消融研究和详细分析证实，代码文档显著提升了定位准确率和修复效果，尤其是在需要深入理解代码仓库的复杂跨文件问题上。

关于未来工作，我们计划从两个方向扩展 RepoRepair，以缓解现有局限：（1）将文档覆盖范围扩展到函数/类之外的多种文件类型，包括配置文件和静态资源；（2）通过扩展工作流或采用受控的基于智能体的方法，支持需要创建文件的功能请求所涉及的增量变更。

这些改进旨在拓宽 RepoRepair 在各种真实开发场景中的适用性和稳健性，同时保持其在理解能力和成本效益方面的核心优势。

致谢

本研究得到国家自然科学基金（62572235）、CCF-华为杨柳基金以及华为-南京大学新型软件技术创意实验室合作基金的资助。

参考文献

[1] Vaibhav Aggarwal、Ojasv Kamal、Abhinav Japesh、Zhijing Jin 和 Bernhard Schölkopf。2025年。DARS：通过自适应树遍历进行动态动作重采样，以提升编码代理性能。*CoRR* abs/2503.14269（2025年）。https://doi.org/ 10 48550 2503 14269 2503 14269 。/ARXIV。。arXiv：。Miltiadis Allamanis、Hao Peng 和 Charles [2] Sutton。。一种用于源代码极限摘要的卷积注意力网络。载于第*33*届国际机器学习会议论文集，*ICML 2016*，纽约市，美国纽约州，*2016*年*6*月*19*日至*24*日（*JMLR*研讨会与会议论文集，第*48*卷），Maria-Florina Balcan 和 Kilian Q. Weinberger（编）。JMLR.org，2091–2100。http://proceedings.mlr.press/v48/allamanis16.html [3] Antonis Antoniades、Albert Örwall、Kexun Zhang、Yuxi Xie、Anirudh Goyal 和 William Yang Wang。2025年。SWE-Search：利用蒙特卡洛树搜索和迭代优化增强软件代理。载于第*13*届国际学习表征会议，*ICLR 2025*，新加坡，*2025*年*4*月*24*日至*28*日。OpenReview.net。https: // openreview.net/forum?id=G7sIFXugTX [4] Islem Bouzenia、Premkumar T. Devanbu 和 Michael Pradel。2024年。RepairAgent：用于程序修复的自主、基于大语言模型的代理。*CoRR* abs/ 2403 17134 2024 10 48550 2403 17134 2403 17134 。（）。 [5] https://doi.org/。/ARXIV。。arXiv：。Dong Chen、Shaoxin Lin、Muhan Zeng、Daoguang Zan、Jian-Gang Wang、Anton Cheshkov、Jun Sun、Hao Yu、Guoliang Dong、Artem Aliev、Jie Wang、Xiao Cheng、Guangtai Liang、Yuchi Ma、Pan Bian、Tao Xie 和 Qianxiang Wang。2024年。CodeR：利用多智能体和任务图进行问题解决。*CoRR* abs/2406.01304（2024年）。https://doi.org/10.48550/ ARXIV.2406.01304 arXiv:2406.01304[6] Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Pondé de Oliveira Pinto、Jared Kaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman、Alex Ray、Raul Puri、Gretchen Krueger、Michael Petrov、Heidy Khlaaf、Girish Sastry、Pamela Mishkin、Brooke Chan、Scott Gray、Nick Ryder、Mikhail Pavlov、Alethea Power、Lukasz Kaiser、Mohammad Bavarian、Clemens Winter、Philippe Tillet、Felipe Petroski Such、Dave Cummings、Matthias Plappert、Fotios Chantzis、Elizabeth Barnes、Ariel Herbert-Voss、William Hebgen Guss、Alex Nichol、Alex Paino、Nikolas Tezak、Jie Tang、Igor Babuschkin、Suchir Balaji、Shantanu Jain、William Saunders、Christopher Hesse、Andrew N. Carr、Jan Leike、Joshua Achiam、Vedant Misra、Evan Morikawa、Alec Radford、Matthew Knight、Miles Brundage、Mira Murati、Katie Mayer、Peter Welinder、Bob McGrew、Dario Amodei、Sam McCandlish、Ilya Sutskever 和 Wojciech Zaremba。2021年。对代码训练的大语言模型进行评估。*CoRR* abs/ 2107 03374 2021 2107 03374 2107 03374 。（）。arXiv：。 [7] 2024 https://arxiv.org/abs/。Yuxiao Chen、Jingzheng Wu、Xiang Ling、Changjiang Li、Zhiqing Rui、Tianyue Luo 和 Yanjun Wu。。当大语言模型面对仓库级自动程序修复时：它们做得有多好？载于*2024*年*IEEE/ACM*第*46*届国际软件工程会议：配套论文集，*ICSE Companion 2024*，里斯本，葡萄牙，*2024*年*4*月*14*日至*20*日。ACM，459–471。https://doi.org/10.1145/3639478.3647633 [8] 塞尔吉奥·科泽蒂 B . de Souza、Nicolas Anquetil 和 Káthia Marçal de Oliveira . 2005. 软件维护所必需的文档研究。载于 第*23*届年度国际传播设计会议论文集：普适信息的文档编写与设计，*SIGDOC 2005*，英国考文垂，*2005*年*9*月*21*日至*23*日，Scott R. Tilley 和 Robert M. Newman（编）。ACM，68–75。 https://doi.org/10.1145/1085313.1085331 [9] DeepSeek - AI、Aixin Liu、Bei Feng、Bing Xue、Bingxuan Wang、Bochao Wu、Chengda Lu、Chenggang Zhao、Chengqi Deng、Chenyu Zhang、Chong Ruan、Damai Dai、Daya Guo、Dejian Yang、Deli Chen、Dongjie Ji、ErhangLi、FangyunLin、Fucong Dai、Fuli Luo、Guangbo Hao、Guanting Chen、Guowei Li、H. Zhang、Han Bao、Hanwei Xu、Haocheng Wang、Haowei Zhang、Honghui Ding、Huajian Xin、Huazuo Gao、Hui Li、Hui Qu、J. L. Cai、Jian Liang、Jianzhong Guo、Jiaqi Ni、Jiashi Li、Jiawei Wang、Jin Chen、Jingchang Chen、Jingyang Yuan、Junjie Qiu、Junlong Li、Junxiao Song、Kai Dong、Kai Hu、Kaige Gao、Kang Guan、Kexin Huang、Kuai Yu、Lean Wang、Lecong Zhang、Lei Xu、Leyi Xia、Liang Zhao、Litong Wang、Liyue Zhang、Meng Li、Miaojun Wang、Mingchuan Zhang、Minghua Zhang、Minghui Tang、Mingming Li、Ning Tian、Panpan Huang、Peiyi Wang、Peng Zhang、Qiancheng Wang、Qihao Zhu、Qinyu Chen、Qiushi Du、R. J. Chen、R. L. Jin、Ruiqi Ge、Ruisong Zhang、Ruizhe Pan、Runji Wang、Runxin Xu、Ruoyu Zhang、Ruyi Chen、S. S. Li、Shanghao Lu、Shangyan Zhou、Shanhuang Chen、Shaoqing Wu、Shengfeng Ye、Shengfeng Ye、Shirong Ma、Shiyu Wang、Shuang Zhou、Shuiping Yu、Shunfeng Zhou、Shuting Pan、T. Wang、Tao Yun、Tian Pei、Tianyu Sun、W. L. Xiao和Wangding

Zeng. 2024年. DeepSeek-V3技术报告. *CoRR* abs/2412.19437 (2024年). https://doi.org/10.48550/ARXIV.2412.19437 arXiv:2412.19437[10] Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xiaocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、Daxin Jiang和Ming Zhou. 2020年. CodeBERT：面向编程与自然语言的预训练模型. 载于计算语言学协会发现：*EMNLP 2020*，在线活动，*2020*年*11*月*16*日至*20*日（*ACL*发现，第*EMNLP 2020*卷），Trevor Cohn、Yulan He和Yang Liu（编）. 计算语言学协会, 1536 1547 10 18653 1 2020 139 – . https://doi.org/ . /V / .FINDINGS-EMNLP. Kai Huang[11]、Jian Zhang、Xiaofei Xie和Chunyang Chen. . 看见即修复2025：面向视觉软件问题修复的多模态大语言模型跨模态推理. *CoRR* abs/2506.16136 (2025年). https://doi.org/10.48550/ARXIV.2506.16136 arXiv:2506.16136[12] Srinivasan Iyer、Ioannis Konstas、Alvin Cheung和Luke Zettlemoyer. 2016年. 使用神经注意力模型摘要源代码. 载于 第*54*届计算语言学协会年会论文集，*ACL 2016*，*2016*年*8*月*7*日至*12*日，德国柏林，第*1*卷：长文. 计算机语言学协会. https://doi.org/ 10 18653 1 16 1195 . /V /P - Carlos E. Jimenez、John Yang、Alexander Wettig、Shunyu Yao[13]、Kexin Pei、Ofir Press和Karthik R. Narasimhan. 2024年. SWE-bench：语言模型能否解决真实世界的GitHub问题？. 载于 第十二届国际学习表征会议，*ICLR 2024*，奥地利维也纳，*2024*年*5*月*7*日至*11*日. OpenReview.net. https://openreview.net/forum? id = 8 66 VTF yNQM Jeff Johnson、Matthijs Douze和Hervé Jégou. 2021年. 基于GPU的十亿级相似性搜索. *IEEE*[14]大数据汇刊 7, 3 2021 535 547 10 1109 2019 2921572 ( ), – . https://doi.org/ . /TBDATA. . René Just、Darioush Jalali和Michael D. Ernst. . Defects J[15]：用于支持Java2014程序受控测试研究的现有缺陷数据库. 载于软件测试与分析国际研讨会，*ISSTA '14*，美国加利福尼亚州圣何塞，*2014*年*7*月*21*日至*26*日，Corina S. Pasareanu和Darko Marinov（编）. ACM, 437–440. https://doi.org/10.1145/ 2610384.2628055[16] Junaed Younus Khan和Gias Uddin. 2022年. 使用GPT-3的自动代码文档生成. 载于 第*37*届*IEEE/ACM*自动化软件工程国际会议，*ASE 2022*，美国密歇根州罗切斯特，*2022*年*10*月*10*日至*14*日. ACM, 174 1 174 6 10 1145 3551349 3559548 : – : . https://doi.org/ . / . Hongwei Li、Yuheng Tang、Shiqi Wang[17]和Wenbo Guo. 2025年. PatchPilot：稳定且高性价比的智能体补丁框架. *CoRR* abs/ 2502 02747 2025 10 48550 2502 02747 2502 02747 . ( ). https://doi.org/ . / [18]ARXIV. . arXiv: . Yujia Li、David H. Choi、Junyoung Chung、Nate Kushman、Julian Schrittwieser、Rémi Leblond、Tom Eccles、James Keeling、Felix Gimeno、Agustin Dal Lago、Thomas Hubert、Peter Choy、Cyprien de Masson d'Autume、Igor Babuschkin、Xinyun Chen、Po-Sen Huang、Johannes Welbl、Sven Gowal、Alexey Cherepanov、James Molloy、Daniel J. Mankowitz、Esme Sutherland Robson、Pushmeet Kohli、Nando de Freitas、Koray Kavukcuoglu和Oriol Vinyals. 2022年. 使用AlphaCode的竞赛级代码生成. *CoRR* abs/2203.07814 (2022年). https://doi.org/10.48550/ARXIV.2203.07814 arXiv:2203.07814[19] Haotian Liu、Chunyuan Li、Qingyang Wu和Yong Jae Lee. 2023年. 视觉指令微调. 载于 神经信息处理系统进展*36*：*2023*年神经信息处理系统年会，*NeurIPS 2023*，美国路易斯安那州新奥尔良，*2023*年*12*月*10*日至*16*日，Alice Oh、Tristan Naumann、Amir Globerson、Kate Saenko、Moritz Hardt和Sergey Levine（编）. http://papers.nips.cc/paper_files/paper/2023/hash/6dcf277ea32ce3288914faf369fe6de0-Abstract- Conference.html[20] Qinyu Luo、Yining Ye、Shihao Liang、Zhong Zhang、Yujia Qin、Yaxi Lu、Yesai Wu、Xin Cong、Yankai Lin、Yingli Zhang、Xiaoyin Che、Zhiyuan Liu和Maosong Sun. 2024年. RepoAgent：一个由大语言模型驱动的开源仓库级代码文档生成框架. *CoRR* abs/2402.16667 (2024年). https://doi.org/10.48550/ARXIV.2402. 16667 arXiv:2402.16667[21] Fangwen Mu、Junjie Wang、Lin Shi、Song Wang、Shoubin Li和Qing Wang. 2025年. EXPEREPAIR：双记忆增强的基于大语言模型的仓库级程序修复. *CoRR* abs/2506.10484 (2025年). https://doi.org/10.48550/ARXIV. 2506.10484 arXiv:2506.10484[22] OpenAI. 2023年. GPT-4技术报告. *CoRR* abs/2303.08774 (2023年). https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774 [23] Rangeet Pan、Ali Reza Ibrahimzada、Rahul Krishna、Divya Sankar、Lambert Pouguem Wassi、Michele Merler、Boris Sobolev、Raju Pavuluri、Saurabh Sinha和Reyhaneh Jabbarvand. 2024年. 翻译中的迷失：大语言模型在代码翻译过程中引入缺陷的研究. 载于 第*46*届 *IEEE/ACM* 国际软件工程会议论文集，*ICSE 2024*，葡萄牙里斯本，*2024*年*4*月*14*日至*20*日. ACM, 82:1–82:13. https://doi.org/10.1145/3597503. 3639226[24] Noah Patton、Kia Rahmani、Meghana Missula、Joydeep Biswas和Isil Dillig. 2024年. 面向长时程机器人任务的示教编程. *ACM*程序设计语言汇刊 8, POPL (2024年), 512–545. https://doi.org/10.1145/3632860

[25] Alec Radford、Jong Wook Kim、Chris Hallacy、Aditya Ramesh、Gabriel Goh、Sandhini Agarwal、Girish Sastry、Amanda Askell、Pamela Mishkin、Jack Clark、Gretchen Krueger 和 Ilya Sutskever。2021年。从自然语言监督中学习可迁移的视觉模型。载于第*38*届国际机器学习会议论文集，*ICML 2021*，*2021*年*7*月*18*日至*24*日，线上会议（机器学习研究进展，第*139*卷），Marina Meila 和 Tong Zhang（编）。PMLR，8748–8763。http://proceedings.mlr.press/v139/radford21a.html [26] Baptiste Rozière、Marie-Anne Lachaux、Lowik Chanussot 和 Guillaume Lample。2020年。编程语言的无监督翻译。载于神经信息处理系统进展*33*：*2020*年神经信息处理系统年会，*NeurIPS 2020*，*2020*年*12*月*6*日至*12*日，线上，Hugo Larochelle、Marc'Aurelio Ranzato、Raia Hadsell、Maria-Florina Balcan 和 Hsuan-Tien Lin（编）。https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html[27] Baptiste Rozière、Jie Zhang、François Charton、Mark Harman、Gabriel Synnaeve 和 Guillaume Lample。2022年。利用自动化单元测试进行无监督代码翻译。载于第十届国际学习表征会议，*ICLR 2022, 25 29, 2022* = 6 4 4 线上会议，*4* 月—。[28] OpenReview.net。https://openreview.net/forum?id cmt- KtR c Haifeng Ruan、Yuntong Zhang 和 Abhik Roychoudhury。. SpecRover：通过大语言模型进行代码意图提取。载于第届 *IEEE/ACM* 国际软件工程会议，*ICSE 2025*，渥太华，加拿大安大略省，*2025*年*4*月*26*日至*5*月*6*日。IEEE， 963 974 10 1109 55347 2025 00080 – 。https://doi.org/ 。 /ICSE 。 。 André Silva、Sen Fang[29] 和 Martin Monperrus。. RepairLLaMA：用于程序修复的高效表示与微调适配器。*CoRR* abs/ 2312 15698 2023 10 48550 2312 15698 2312 15698 。（）。https://doi.org/[30] 。 /ARXIV. 。 arXiv：。David Silver、Aja Huang、Chris J. Maddison、Arthur Guez、Laurent Sifre、George van den Driessche、Julian Schrittwieser、Ioannis Antonoglou、Vedavyas Panneershelvam、Marc Lanctot、Sander Dieleman、Dominik Grewe、John Nham、Nal Kalchbrenner、Ilya Sutskever、Timothy P. Lillicrap、Madeleine Leach、Koray Kavukcuoglu、Thore Graepel 和 Demis Hassabis。2016年。运用深度神经网络和树搜索掌握围棋。*Nat.* 529，7587（2016年），484–489。https://doi.org/ 10 1038 16961 。/NATURE Aditya Bharat Soni、Boxuan Li、Xingyao Wang、Valerie Chen 和 Graham Neubig[31]。2025年。具备多模态浏览能力的编程代理是通用问题求解器。*CoRR* abs/2506.03011（2025年）。https://doi.org/10.48550/ARXIV.2506.03011 arXiv:2506.03011[32] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N. Gomez、Lukasz Kaiser 和 Illia Polosukhin。2017年。注意力机制就是你所需的一切。载于神经信息处理系统进展*30*：*2017*年神经信息处理系统年会，*2017*年*12*月*4*日至*9*日，美国加利福尼亚州长滩，Isabelle Guyon、Ulrike von Luxburg、Samy Bengio、Hanna M. Wallach、Rob Fergus、S. V. N. Vishwanathan 和 Roman Garnett（编）。5998–6008。https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html [33] Xingyao Wang、Boxuan Li、Yufan Song、Frank F. Xu、Xiangru Tang、Mingchen Zhuge、Jiayi Pan、Yueqi Song、Bowen Li、Jaskirat Singh、Hoang H. Tran、Fuqiang Li、Ren Ma、Mingzhang Zheng、Bill Qian、Yanjun Shao、Niklas Muennighoff、Yizhe Zhang、Binyuan Hui、Junyang Lin 等。2025年。OpenHands：面向人工智能软件开发者的通用代理开放平台。载于第*13*届国际学习表征会议，*ICLR 2025*，新加坡，*2025*年*4*月*24*日至*28*日。OpenReview.net。https://openreview.net/forum?id=OJd3ayDDoF [34] Yue Wang、Weishi Wang、Shafiq R. Joty 和 Steven C. H. Hoi。2021年。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。载于*2021*年自然语言处理经验方法会议论文集，*EMNLP 2021*，线上会议/多米尼加共和国蓬塔卡纳，*2021*年*11*月*7*日至*11*日，Marie-Francine Moens、Xuanjing Huang、Lucia Specia 和 Scott Wen-tau Yih（编）。计算语言学协会， 8696 8708 10 18653 1 2021 685 – 。https://doi.org/ 。 /V / .EMNLP-MAIN. Zhou Wang、Alan C. Bovik、Hamid R. Sheikh[35] 和 Eero P. Simoncelli。. 图像质量评估：从误差可见性到结构相似性。*IEEE Trans. Image Process.* 13, 4 2004 600 612 10 1109 2003 819861 （），– 。https://doi.org/[36] 。 /TIP. 。 Jason Wei、Xuezhi Wang、Dale Schuurmans、Maarten Bosma、Brian Ichter、Fei Xia、Ed H. Chi、Quoc V. Le 和 Denny Zhou。2022年。思维链提示引发大语言模型的推理。载于神经信息处理系统进展*35*：*2022*年神经信息处理系统年会，*NeurIPS 2022*，美国路易斯安那州新奥尔良，*2022*年*11*月*28*日至*12*月*9*日，Sanmi Koyejo、S. Mohamed、A. Agarwal、Danielle Belgrave、K. Cho 和 A. Oh（编）。http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html[37] Ratnadira Widyasari、Jia Wei Ang、Truong Giang Nguyen、Neil Sharma 和 David Lo。2024年。揭开错误代码的神秘面纱：用于可解释故障定位的逐步推理。载于*IEEE* 国际软件分析、演化与再工程会议，*SANER 2024*，芬兰罗瓦涅米，*2024*年*3*月*12*日至*15*日。IEEE，568–579。https://doi.org/10. 1109/SANER60148.2024.00064[38] W. Eric Wong、Joseph R. Horgan、Saul London 和 Hiralal Agrawal。1997年。实践中有效回归测试的研究。载于第八届国际软件可靠性工程研讨会，*ISSRE 1997*，美国新墨西哥州阿尔伯克基，*1997*年*11*月*2*日至*5*日。IEEE计算机学会，264–274。https://doi.org/10.1109/ISSRE.1997.630875

[39] Chunqiu Steven Xia、Yinlin Deng、Soren Dunn 和 Lingming Zhang。2024年。Agentless：揭开基于大语言模型的软件工程代理的神秘面纱。*CoRR* abs/ 2407 01489 2024 10 48550 2407 01489 2407 01489。（）。

[40] https://doi.org/。/ARXIV.。arXiv：。Chunqiu Steven Xia、Yuxiang Wei 和 Lingming Zhang。2023.。大型预训练语言模型时代的自动程序修复。载于第*45*届 *IEEE/ACM* 国际软件工程会议，*ICSE 2023*，澳大利亚墨尔本，*5*月 *14 20, 2023* 1482 1494 10 1109 48619 2023 00129 -。IEEE，–。https://doi.org/。/ICSE。。

[41] Chunqiu Steven Xia 和 Lingming Zhang。2022.。少训练，多修复：通过零样本学习重新审视自动程序修复。载于第*30*届 *ACM* 欧洲软件工程大会与软件工程基础研讨会联合会议论文集，*ESEC/FSE 2022*，新加坡，新加坡，*2022*年*11*月*14*日至*18*日，Abhik Roychoudhury、Cristian Cadar 和 Miryung Kim（编）。ACM， 959 971 10 1145 3540250 3549101 –。

[42] https://doi.org/。/。Chunqiu Steven Xia 和 Lingming Zhang。2023.。让对话继续：使用 ChatGPT 修复漏洞，每个漏洞收费 $ 162 337 0 42.。*CoRR* abs/ 2304 00385 2023 10 48550 2304 00385 2304 00385。（）。

[43] https://doi.org/。/ARXIV.。arXiv：。Xin Xia、Lingfeng Bao、David Lo、Zhenchang Xing、Ahmed E. Hassan 和 Shanping Li。2018.。衡量程序理解：针对专业人员开展的大规模实地研究。载于第*40*届国际软件工程会议论文集，*ICSE 2018*，瑞典哥德堡，*2018*年*5*月*27*日至*6*月*3*日，Michel Chaudron、Ivica Crnkovic、Marsha Chechik 和 Mark Harman（编）。ACM， 584 10 1145 3180155 3182538。https://doi.org/。/。Aidan Z. H. Yang、

[44] Claire Le Goues、Ruben Martins 和 Vincent J. Hellendoorn。2024.。用于无测试故障定位的大语言模型。载于第*46*届 *IEEE/ACM* 国际软件工程会议论文集，*ICSE 2024, 14 20, 2024* 17 1 17 12 10 1145 3597503 3623342 葡萄牙里斯本，*4*月—。ACM，：–：。https://doi.org/。/。Dayu Yang、Antoine Simoulin、Xin Qian、Xiaoyi Liu、Yuwei Cao、Zhaopu Teng 和 Grey Yang。2025.。DocAgent：用于自动代码文档生成的多智能体系统。*CoRR* abs/2504.08725（2025年）。https://doi.org/10. 48550/ARXIV.2504.08725 arXiv:2504.08725[46] John Yang、Carlos E. Jimenez、Alexander Wettig、Kilian Lieret、Shunyu Yao、Karthik Narasimhan 和 Ofir Press。2024年。SWE-agent：代理-计算机接口实现自动化软件工程。载于神经信息处理系统进展第*38*卷：*2024*年神经信息处理系统年会，*NeurIPS 2024*，加拿大不列颠哥伦比亚省温哥华，*2024*年*12*月*10*日至*15*日，Amir Globersons、Lester Mackey、Danielle Belgrave、Angela Fan、Ulrich Paquet、Jakub M. Tomczak 和 Cheng Zhang（编）。http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html[47] John Yang、Carlos E. Jimenez、Alex L. Zhang、Kilian Lieret、Joyce Yang、Xindi Wu、Ori Press、Niklas Muennighoff、Gabriel Synnaeve、Karthik R. Narasimhan、Diyi Yang、Sida Wang 和 Ofir Press。2025年。SWE-bench Multimodal：人工智能系统能否推广到视觉软件领域？载于第*13*届国际学习表征会议，*ICLR 2025*，新加坡，*2025*年*4*月*24*日至*28*日。OpenReview.net。https://openreview.net/forum?id=riTiq3i21b[48] Yuntong Zhang、Haifeng Ruan、Zhiyu Fan 和 Abhik Roychoudhury。2024年。AutoCodeRover：自主程序改进。载于第*33*届 *ACM SIGSOFT* 国际软件测试与分析研讨会论文集，*ISSTA 2024*，奥地利维也纳，*2024*年*9*月*16*日至*20*日，Maria Christakis 和 Michael Pradel（编）。ACM，1592–1604。https://doi.org/ 10 1145 3650212 3680384。/。Junji Zhi、Vahid

[49] Garousi-Yusifoglu、Bo Sun、Golara Garousi、S. M. Shahnewaz 和 Günther Ruhe。2015年。软件开发文档的成本、效益与质量：一项系统映射研究。*J. Syst. Softw.* 99（2015年），175–198。https://doi.org/10.1016/J.JSS.2014.09.042

收稿日期：2007年2月20日；修改日期：2009年3月12日；录用日期：2009年6月5日