软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(4):1690−1714 [doi: 10.13328/j.cnki.jos.007474] [CSTR: 32375.14.jos.007474]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于大语言模型的故障复现测试用例生成方法*

汪 莹, 字千成, 彭 鑫, 娄一翎

(复旦大学 计算机科学技术学院, 上海 200438)

通信作者: 娄一翎, E-mail: yilinglou@fudan.edu.cn

摘 要: GitHub 是目前最流行的开源项目管理平台之一. 由于团队协作的需要, GitHub 引入了问题报告跟踪功能以方便项目使用者提交和追踪项目中出现的问题或新功能请求. 问题报告贡献者在解决问题报告时, 通常需要执行故障复现测试用例来复现问题报告中提到的问题并验证问题报告是否解决. 然而, 在 SWE-bench Lite 数据集上进行实证研究发现, 有近 90% 的问题报告在用户提交时没有附带故障复现测试用例, 这导致问题报告贡献者在解决问题报告时还需额外编写故障复现测试用例, 带来了额外的工作负担. 现有的故障复现测试用例生成技术通常依赖错误栈信息, 然而 GitHub 问题报告中并未明确要求有这类信息. 因此, 提出基于大语言模型的故障复现测试用例生成方法, 旨在自动化地为 GitHub 问题报告生成故障复现测试用例, 帮助问题报告贡献者复现、理解并验证问题报告, 提升问题报告解决效率. 该方法首先通过检索与问题报告相关的多样化代码上下文信息, 包括报错根函数、import 语句和测试用例样本, 随后构建精确的 prompt, 以引导大语言模型生成有效的故障复现测试用例. 开展对比实验和消融实验, 验证所提方法在面向 GitHub 问题报告的故障复现测试用例生成任务上的有效性.

关键词: 故障复现; 测试用例生成; 大语言模型; 检索增强生成

中图法分类号: TP311

中文引用格式: 汪莹,字千成,彭鑫,娄一翎.基于大语言模型的故障复现测试用例生成方法.软件学报,2026,37(4):1690–1714.http://www.jos.org.cn/1000-9825/7474.htm

英文引用格式: Wang Y, Zi QC, Peng X, Lou YL. Failure Reproducing Test Case Generation Method Based on Large Language Model. Ruan Jian Xue Bao/Journal of Software, 2026, 37(4): 1690–1714 (in Chinese). http://www.jos.org.cn/1000-9825/7474.htm

Failure Reproducing Test Case Generation Method Based on Large Language Model

WANG Ying, ZI Qian-Cheng, PENG Xin, LOU Yi-Ling

(School of Computer Science, Fudan University, Shanghai 200438, China)

Abstract: GitHub is one of the most popular open-source project management platforms. Due to the need for team collaboration, GitHub introduced an issue tracking function to facilitate project users in submitting and tracking problems or new feature requests. When resolving issues, contributors of open-source projects typically need to execute failure reproducing test cases to reproduce the problems mentioned in the issue and verify whether the issue has been resolved. However, empirical research conducted on the SWE-bench Lite dataset reveals that nearly 90% of issues are submitted without failure reproducing test cases, leading contributors to write additional failure reproducing test cases when resolving the issues, bringing additional work burden. Existing failure reproducing test case generation methods usually rely on stack trace information, but GitHub issues do not explicitly require such information. Therefore, this study proposes a failure reproducing test case generation method based on a large language model, aimed at automatically generating failure reproducing test cases for GitHub issues, assisting issue contributors in reproducing, understanding, and verifying issues, and improving the efficiency of issue resolution. This method first retrieves diverse code context information related to the issue, including error root functions, import statements, and test case examples, then constructs precise prompts to guide the large language model in generating effective failure reproducing test cases. This study conducts comparative and ablation experiments to verify the effectiveness of this method in generating failure reproducing test cases for GitHub issues.

Key words: failure reproducing; test case generation; large language model (LLM); retrieval-augmented generation (RAG)

* 收稿时间: 2024-11-09; 修改时间: 2025-03-25, 2025-05-06; 采用时间: 2025-05-26; jos 在线出版时间: 2025-09-28
CNKI 网络首发时间: 2025-09-30

GitHub[1]是目前最流行的开源项目管理平台之一, 凭借其完整的版本控制、便捷的协作工具和丰富的社区资源, 吸引了大量开发者和团队的参与, 汇聚了大量的开源项目. 为了满足团队协作的需求, GitHub 引入了问题报告(issue) 跟踪功能, 开发者可以便捷地报告、追踪、管理问题和功能请求. GitHub 中的问题报告通常包含标题、问题描述、代码上下文、问题报告状态和评论等内容, 开发者可以根据这些信息编写代码补丁, 以解决问题报告中提出的问题或实现新的功能特性. 问题报告跟踪功能极大地简化了团队间的沟通并降低协作开销, 提高了软件开发的整体效率和质量.

在解决 GitHub 问题报告时, 测试用例起着至关重要的作用. 一方面, 测试用例需要复现问题报告中描述的问题场景, 帮助问题报告贡献者更好地理解该问题; 另一方面, 测试用例通过设置断言等方式来预测正确的执行结果, 从而验证问题报告是否得到解决. 本文将用于复现问题报告所描述问题并验证问题报告是否解决的测试用例称为“故障复现测试用例”. 当故障复现测试用例在补丁提交前的代码上运行失败, 而在补丁提交后的代码上运行通过时, 表明问题报告贡献者提交的代码补丁成功地解决了给定的问题报告.

然而, 由于 GitHub 中问题报告的报告格式和内容缺乏固定的约束, 许多问题报告在提交时通常未附带故障复现测试用例. 因此, 问题报告贡献者在提交解决该问题报告的代码补丁之外, 还需额外编写故障复现测试用例补丁, 以复现并验证该问题报告是否解决, 这无疑增加了开发人员的工作负担. 为了进一步验证 GitHub 问题报告中故障复现测试用例不足的现象, 本文在 SWE-bench Lite[2]数据集上进行了实证研究, 分析并统计了该数据集中 300 个 GitHub 问题报告中缺乏故障复现测试用例的数量. 结果表明, 300 个 GitHub 问题报告中有 268 个在提交时未包含故障复现测试用例, 反映出 GitHub 问题报告在故障复现测试用例方面存在显著不足. 为了解决这一问题, 本文提出了一种基于大语言模型的故障复现测试用例生成方法, 旨在自动化地为 GitHub 问题报告生成故障复现测试用例, 从而减轻问题报告贡献者的工作负担, 并提高 GitHub 问题报告的解决效率.

现有的故障复现测试用例生成方法主要采用基于覆盖率[3]、模型检查[4]和符号执行[5]等技术方法, 这些方法通常依赖于错误栈追踪信息. 然而, GitHub 问题报告中并不直接提供此类信息, 因此这些方法无法直接应用于 GitHub 问题报告中的故障复现测试用例生成任务. 为了解决这一问题, Kang 等人[6]提出的 Libro 方法采用大语言模型和 few-shot[7]技术生成故障复现测试用例. 该方法通过在 prompt 中提供固定的问题描述和对应的故障复现测试用例, 帮助大语言模型理解故障复现测试用例的内容. 然而, Libro 方法仅为大语言模型提供简单且固定的输入输出样本, 缺乏与 GitHub 问题报告相关的上下文信息, 未能从根本上提高模型对问题报告内容的理解, 并不能很好地解决复杂的问题报告.

为了解决面向 GitHub 问题报告的故障复现测试用例生成问题, 尤其是针对问题报告复杂性和缺乏上下文信息等难点, 本文提出了一种基于大语言模型的故障复现测试用例生成方法. 该方法结合大语言模型、检索增强生成和提示工程技术, 旨在自动化地为 GitHub 问题报告生成故障复现测试用例. 具体来说, 给定一个 GitHub 问题报告, 本文在其所在代码仓库中检索 3 类关键信息: 问题报告描述中的报错根函数、与问题报告相关的 import 语句和与问题报告相关的测试用例样本. 首先, 对于问题报告描述中的报错根函数, 本文通过抽取错误栈并找到最后一个项目本身的函数代码, 将其视为问题报告最终出错的位置, 从而实现问题的定位. 其次, 本文通过检索与问题报告最相关的测试文件, 提取其中的 import 语句, 以帮助大语言模型理解项目中实际 API 的调用方式. 最后, 本文从代码仓库中逐级检索与问题报告相关的测试用例样本. 具体而言, 先抽取仓库中的所有测试文件, 并利用大语言模型筛选出与问题报告最相关的测试文件, 然后提取其中的测试函数, 计算测试函数和问题报告标题的 embedding 相似度, 最终筛选出最相关的测试函数作为样例, 提供给大语言模型以提高测试用例生成的质量. 通过获得上述 3 类信息, 本文利用提示工程技术将这些信息整合, 构建精确的 prompt, 引导大语言模型生成能够复现问题报告并验证问题报告是否解决的故障复现测试用例. 该方法通过结合上下文检索和大语言模型生成, 显著提升了故障复现测试用例生成的准确性和有效性.

为验证本文方法的可行性和有效性, 本文在 SWE-bench Lite 数据集上与 Libro 方法进行了对比实验. 实验结果显示, 在 SWE-bench Lite 数据集的 300 个 GitHub 问题报告中, Libro 方法仅成功为 20 个问题报告生成了故障复现测试用例, 占比 6.57%; 而本文的方法成功为 67 个问题报告生成了故障复现测试用例, 占比提升至 22.33%.

这一结果表明, 本文方法显著提升了故障复现测试用例生成的效果, 证明了其有效性. 此外, 为了进一步分析本文检索增强生成部分中 3 类信息的各自作用, 本文还设计了消融实验, 在 prompt 中分别去除 3 类检索信息, 以验证各自的影响. 为降低实验过程中大模型的经济成本, 消融实验在 SWE-bench Lite 数据集的 300 个 GitHub 问题报告中随机挑选了 100 个问题报告作为数据集. 实验结果表明, 当 3 类检索信息都使用时, 成功生成故障复现测试用例的问题报告占比为 27%; 去除问题报告的报错根函数后, 占比降至 24%; 去除与问题报告相关的 import 语句后, 占比降至 21%; 去除与问题报告相关的测试用例样本后, 占比显著降至 14%. 这些结果表明, 本文提出的检索增强生成方法中最为有效的信息来源是与问题报告相关的测试用例样本. 该测试用例样本不仅为大语言模型提供了与问题报告相关的测试函数样例, 还更好地反映了问题报告所在仓库的代码结构和 API 调用方法. 这使得大语言模型能够更好地学习特定代码仓库中测试函数的写法, 从而提高为新问题报告生成故障复现测试用例的效果.

本文主要贡献总结如下.

(1) 通过在 SWE-bench Lite 数据集上进行实证研究, 发现该数据集 300 个 GitHub 问题报告中, 有 268 个在报告时缺少故障复现测试用例, 占比高达 89.33%, 这一现象反映出 GitHub 问题报告在故障复现测试用例方面存在显著的不足问题.

(2) 提出了一种结合大语言模型、检索增强生成和提示工程技术的面向 GitHub 问题报告的故障复现测试用例生成方法, 以自动化生成 GitHub 问题报告的故障复现测试用例.

(3) 在 SWE-bench Lite 数据集上开展了对比实验、消融实验和准确性分析实验, 以证明本文方法的有效性.

本文第 1 节介绍研究背景及相关工作. 第 2 节描述在 SWE-bench Lite 数据集上针对 GitHub 问题报告故障复现测试用例不足问题的实证研究, 揭示本文的研究动机. 第 3 节阐述本文所提出的结合大模型、检索增强生成和提示工程技术的创新方法. 第 4 节展示对比实验、消融实验及准确性分析实验的结果, 验证本文方法的有效性. 第 5 节讨论本文方法的局限性及未来发展方向, 并在最后对全文进行总结.

1 研究背景与相关工作

本节首先介绍面向 GitHub 问题报告的故障复现测试用例生成的研究背景, 探讨该研究在软件开发中的实际价值和意义. 随后, 本节将详细阐述测试用例生成、大语言模型在代码方面的应用、检索增强生成以及提示工程领域的相关工作, 并分析这些技术方向与本文提出的方法之间的关系. 其中, 测试用例生成部分将介绍当前测试用例生成方法及其在故障复现测试用例生成任务中的不足, 引出本文的研究动机. 大语言模型在代码方面的应用部分将探讨大语言模型在代码生成、程序修复任务中的最新进展, 分析其在理解和处理复杂代码逻辑方面的优势与局限. 检索增强生成部分将探讨如何通过检索相关上下文信息提高大语言模型生成内容的准确性和相关性. 提示工程部分则介绍优化提示的关键技术, 结合不同的提示策略提升大语言模型的生成效果. 这些研究方向为本文提出的基于大语言模型的故障复现测试用例生成方法提供了理论基础和技术支持.

1.1 研究背景

GitHub 是全球知名的代码托管平台, 用户可以在该平台上托管项目代码, 并借助其丰富的工具提升协作开发体验. 为了集中化地管理开源项目中的问题和功能请求, GitHub 引入了问题报告跟踪机制. 通过这一机制, 开发者能够在项目中提出遇到的代码问题或新的功能需求, 并由 GitHub 社区中的成员共同协作解决. 这种方式不仅提高了问题解决的效率, 也促进了社区内的知识共享和技术交流.

近年来, 已有研究人员对 GitHub 问题报告中的代码问题展开研究, 研究工作主要集中在自动化地解决 GitHub 问题报告上. Jimenez 等人[8]收集了 GitHub 上 12 个备受欢迎的 Python 代码仓库及其问题报告信息, 包括问题报告的描述、正确代码补丁及问题报告相关故障复现测试用例, 并以此构建了 SWE-bench 数据集. SWE-bench 数据集在筛选 GitHub 问题报告数据时, 采用了一个重要规则: 选择那些 Pull Request 中对代码仓库中的测试文件进行了修改的问题报告. 这意味着, 开发者在提交代码补丁解决问题报告的同时, 还提交了相应的测试用例补丁, 以复现问题报告中的问题并验证问题报告是否得到解决. 通过这种筛选方式, SWE-bench 数据集中的每一个问题报告

都有对应的故障复现测试用例, 从而提高了数据集的可验证性和完整性. 同时, Jimenez 等人[8]还提出了一个评估框架, 给定一个代码库以及要解决的问题描述, 大语言模型的任务是编辑代码库以解决问题, 并在与该问题报告相关的故障复现测试用例上进行测试, 验证问题报告是否解决. 解决 SWE-bench 中的问题通常需要同时理解和协调多个函数、类甚至文件的更改, 要求模型与执行环境交互, 处理极长的上下文并执行远远超出传统代码生成的复杂推理. 因此, SWE-bench 数据集中每个数据实例的运行需要消耗大量时间和计算资源. SWE-bench Lite 是 SWE-bench 数据集的子集, 其在 SWE-bench 数据集中挑选了 300 个典型实例, 侧重评估独立的功能性错误修复, 降低了实验开销.

SWE-bench 数据集凭借其高度的挑战性和实际应用价值, 吸引了众多研究者在其上开展自动化代码修复的研究. 首先, Jimenez 等人[8]在构建 SWE-bench 数据集的同时, 提出了一种微调 Code LLaMA[9]模型以生成解决 GitHub 问题报告的代码补丁的方法. 该方法将 GitHub 问题报告的问题描述和相关代码文件作为输入, 以解决给定问题报告的代码补丁作为输出, 从而对 Code LLaMA 模型进行微调, 实现自动化代码修复. 随着大语言模型能力的不断提升, 许多研究开始将大语言模型作为智能代理 (agent), 利用其解决 GitHub 问题报告中的问题. 例如, Yang 等人[10]提出了 SWE-agent 方法, 这是一种利用大语言模型与计算机交互来完成软件工程任务的自主系统. SWE-agent 通过为智能代理提供丰富的接口, 使其能够与计算机深度互动, 执行创建和编辑文件、定位代码仓库中的问题、执行测试用例等操作. 该系统能够在接收到相关需求后, 自主决策并逐步生成代码补丁. 另一方面, Zhang 等人[11]提出了 AutoCodeRover 方法, 将大语言模型与代码搜索结合, 用于自动生成解决问题报告的代码补丁. AutoCodeRover 的核心包括两个阶段: 上下文检索和补丁生成. 在上下文检索阶段, 智能代理根据问题报告信息, 通过检索接口搜索与问题报告高度相关的 API, 以获取丰富的上下文信息. 在补丁生成阶段, 智能代理依据补丁应用的反馈结果, 判断生成的补丁是否合适, 并进一步优化, 直到补丁成功解决问题报告或达到优化次数上限. 除了智能代理方法外, Xia 等人[12]提出了 Agentless 方法, 这是一种简化的解决方案, 采用了两阶段过程: 定位和修复. 与复杂的基于智能代理的方案不同, Agentless 不需要让大语言模型做出未来行动的决策或使用复杂的工具. 具体来说, 在定位阶段, 系统从粗粒度到细粒度地检索信息; 而在修复阶段, 利用检索到的信息生成多个补丁, 并借助 SWE-bench 数据集中的测试函数进行筛选, 最终通过重排序选取最佳补丁作为代码修复方案. 通过这种简化流程, Agentless 在保证修复效果的同时, 显著降低了系统的复杂度.

由上述研究可见, GitHub 问题报告的自动修复任务引起了研究者们的广泛关注. 然而, 在生成给定 GitHub 问题报告的代码补丁后, 现有的研究通常需要依赖 SWE-bench 数据集中提供的故障复现测试用例 (在 SWE-bench 数据集中为 fail-to-pass 测试用例) 来复现问题报告中描述的问题, 并验证生成的代码补丁是否正确. 这一过程突显了故障复现测试用例在解决 GitHub 问题报告任务中的重要性. 因此, 为了进一步探讨 GitHub 问题报告中故障复现测试用例的现状, 本文在 SWE-bench Lite 数据集上进行了实证研究. 研究结果表明, 在 SWE-bench Lite 数据集的 300 个 GitHub 问题报告中, 有 268 个在报告时未附带故障复现测试用例, 占比高达 89.33%. 在这些问题报告中, 大多数故障复现测试用例是由问题报告贡献者在解决问题报告时额外编写, 或是在原有的代码仓库测试用例基础上修改而来. 这一现象反映出 GitHub 问题报告在故障复现测试用例方面存在明显的不足, 给问题报告贡献者带来了额外的工作负担. 针对这一问题, 本文展开了深入研究, 并提出了自动生成 GitHub 问题报告故障复现测试用例的有效解决方案.

1.2 相关工作

1.2.1 测试用例生成

在软件开发过程中, 测试用例是验证软件功能和性能是否符合预期的重要工具, 发挥着至关重要的作用. 通过对各个功能模块执行测试用例, 开发团队能够有效发现并修复潜在的缺陷, 从而提升软件的质量和可靠性. 然而, 手动编写测试用例既费时又费力, 并且需要大量的人力成本. 因此, 许多研究工作集中于自动化生成测试用例, 以便开发者可以专注于更具创新性的任务, 从而提升软件开发效率.

传统的测试用例生成技术通常使用基于搜索、基于约束和基于随机等策略来提高测试覆盖率. 基于搜索的测

试 (search-based testing) 利用搜索算法生成测试用例, 旨在通过智能搜索策略提升测试效率与覆盖率. 其核心思想是将测试用例生成视为一个优化问题, 通过定义适应度函数 (fitness function) 来评估测试用例的质量, 以引导搜索过程. 例如, Fraser 等人[13]提出的 EvoSuite 工具通过遗传算法生成测试用例, 首先定义适应度函数以评估测试用例质量, 然后随机生成一组初始测试用例, 接着通过选择、交叉和变异操作不断优化适应度函数, 迭代生成新的测试用例, 直到达到预设的覆盖率或迭代次数. Baars 等人[14]提出了一种构建适应度函数的算法, 该算法将符号信息与动态分析相结合, 可在尝试生成分支充分测试数据时提高基于搜索的测试的效率. 基于约束的测试 (constraint-based testing) 通过定义特定约束条件生成测试用例, 其核心在于使用约束求解器自动生成满足约束的输入, 从而确保测试用例覆盖了特定的程序行为或代码路径. 例如, Blasi 等人[15]提出的 CallMeMaybe 方法使用自然语言处理技术分析 Javadoc 注释, 从中识别时间约束, 并指导测试用例生成器执行遵守时间约束的方法调用序列. DeMilli 等人[16]开发的 Godzilla 工具则使用代数约束描述旨在发现特定类型故障的测试用例, 自动生成并解决约束以创建单元和模块测试用例. 基于随机测试 (random-based testing) 则主要通过随机生成输入数据来测试程序行为. 这种随机生成测试用例的方法通常效果不佳, 因此 Pacheco 等人[17]通过结合从执行测试输入中获得的反馈改进了生成效果. 传统的 3 类测试用例生成方法都旨在提升测试用例的测试覆盖率, 测试覆盖率工具主要关注的是代码路径, 然而由于 GitHub 问题报告均来自实际软件开发场景, 往往只提供问题描述和上下文信息, 缺少必要的代码路径信息, 难以用测试覆盖率来进行测试. 因此, 传统测试用例生成方法在面向 GitHub 问题报告时存在较大的局限性.

随着深度学习技术的发展, 近年来许多研究在传统测试用例生成技术的基础上, 引入模型作为辅助, 取得了显著效果. Dinella 等人[18]提出了 TOGA 方法, 这是一种基于 Transformer 的神经方法 (neural approach), 可以根据焦点方法的上下文推断异常和断言测试预言. Schafer 等人[19]则尝试利用 API 文档的信息来增强模型对代码的理解, 通过在模型输入中添加额外的文档信息, 从而生成更有效的单元测试. 上述方法利用深度学习技术, 将模型在大量代码数据上进行训练, 使模型具备了从上下文推断潜在异常和测试预言的能力, 在一定程度上克服了传统测试用例生成方法中对特征依赖较强、难以泛化的问题. 随着大语言模型的出现, Yuan 等人[20]采用大语言模型技术, 通过迭代反馈生成测试函数, 不断将报错信息反馈给模型, 直到没有错误反馈或达到迭代次数上限. 然而, 由于 GitHub 问题报告没有明确的内容规范要求, 其通常缺乏明确的报错信息或 API 文档, 无法直接给模型提供额外上下文信息. 此外, 在解决 GitHub 问题报告时, 模型需要同时理解和协调多个函数、类甚至文件的更改, 并处理极长的上下文, 推理难度远远超出模型的能力范围, 因此这些方法在实际应用中并不十分适用.

针对故障复现测试用例生成的问题, 已有一些研究提出了相应的解决方案. Soltani 等人[3]提出了基于搜索的 EvoCrash 方法, 用于生成复现崩溃的测试用例. 该方法通过一个适应度函数引导搜索过程, 适应度函数结合了 3 种启发式方法: 代码覆盖率、崩溃覆盖率以及错误栈相似度. 此外, EvoCrash 还探索了多目标优化方法, 以提高生成测试用例的多样性. Nayrolles 等人[4]提出的 JCHARMING 方法结合崩溃追踪和模型检查, 旨在通过自动化手段复现软件崩溃. 该方法首先分析崩溃堆栈数据, 识别与崩溃相关的程序状态, 并利用模型检查技术进一步确认哪些程序语句是复现崩溃所必需的, 进而复现程序崩溃. Chen 等人[5]则提出了一种基于收集到的崩溃栈追踪的崩溃复现框架, 该框架结合了高效的逆向符号执行和创新的序列方法组合技术, 生成能够复现原始崩溃的单元测试用例, 并且避免了增加额外的运行时开销. 在该框架中, 逆向符号执行用于分析崩溃栈, 识别复现崩溃所需的关键执行路径, 而序列方法组合则帮助生成准确的单元测试用例. 上述研究均聚焦于程序崩溃的复现技术, 其方法都依赖于崩溃栈追踪信息, 然而 GitHub 问题报告中并不直接提供此类信息, 因此这些方法不适用于解决 GitHub 问题报告中的故障复现测试用例生成问题.

为解决这一问题, Kang 等人[6]提出的 Libro 方法使用大语言模型和 few-shot[7]技术生成故障复现测试用例. 该方法通过在 prompt 中提供固定的问题描述和对应的故障复现测试用例, 帮助大语言模型理解如何生成测试用例. 然而, Libro 方法仅为大语言模型提供简单且固定的输入输出样本, 缺乏与 GitHub 问题报告相关的上下文信息, 未能从根本上帮助模型更好地理解问题报告内容. 此外, 现有一些基于智能代理 (agent) 的研究, 如 SWE-agent[10]、OpenDevin[21]、CodeR[22]等, 也在解决 GitHub 问题报告的过程中关注了故障复现问题. 这些方法均依赖于大语言模型和 prompt 技术, 通过向智能代理提供 GitHub 问题报告的问题描述, 要求其生成复现问题报告的代码, 进而帮

助大语言模型理解和定位问题报告中的问题. 然而, 这些方法并不专注于生成故障复现测试用例, 它们的目标仅是复现问题报告中的问题, 而非生成故障复现测试用例以验证问题的解决.

相比于上述基于大语言模型的故障复现测试用例生成方法, 本文方法具有以下创新之处.

(1) 本文工作聚焦于 GitHub 问题报告的故障复现测试用例生成问题, 与传统的单元测试和回归测试不同, 本文工作关注的问题涉及更长的上下文, 需要对问题报告及其所在代码仓库有更深入的理解, 更具有挑战性.

(2) 本文方法不依赖于问题报告中的错误栈信息, 而是仅将错误栈视为潜在原因, 辅助大语言模型定位故障问题发生的原因, 为生成测试用例提供支持.

(3) 本文方法并非在 prompt 中提供简单、固定的指令或输入输出示例, 而是通过检索增强生成技术, 获取与给定问题报告相关的多元上下文信息 (报错根函数、import 语句、测试用例样本), 帮助大语言模型更好地理解问题报告的项目背景和相关测试用例特征, 从而生成更加有效且贴合实际的故障复现测试用例.

(4) 相较于 Agent 框架中智能代理生成的故障复现代码, 本文方法不仅关注如何复现问题报告中的问题, 还要求大语言模型通过生成的测试用例验证问题是否已被有效修复, 从而提高修复的准确性和测试用例的实用性.

1.2.2 大语言模型在代码方面的应用

近年来, 随着 AI 领域和算力的快速发展, 以 ChatGPT[23]为代表的大语言模型经过大规模语料的训练, 掌握了各个领域的基础知识, 使其能够出色地完成许多自然语言任务. 这一进展也吸引了大量研究者将大语言模型应用于代码相关任务中.

在代码生成任务方面, 许多方法通过对已有模型进行微调, 得到了更为优秀的模型. 例如, Li 等人[24]通过在 Python 数据上微调 StarCoderBase 模型, 生成了 StarCoder 模型, 该模型在 Python 代码生成任务上表现出色, 并且支持多种编程语言. Luo 等人[25]则采用了 Evol-Instruct 方法, 将复杂指令微调引入代码领域, 训练出 WizardCoder 模型, 显著提升了其对指令的理解能力和代码生成效果. Fried 等人[26]则提出了一种将因果语言建模 (causal language modeling) 和掩码语言建模 (masked language modeling) 相结合的训练方法, 采用因果掩码 (causal masking) 的方法生成了 InCoder 模型, 使得模型具备了同时进行代码生成和代码填充的能力.

在程序修复任务方面, 研究者们也提出了不少创新方法. Jin 等人[27]提出的 InferFix 方法, 将大语言模型在有监督程序修复数据上进行微调, 并结合 few-shot 技术和静态分析技术, 开发了一种端到端的程序修复解决方案. Xia 等人[28]提出了 ChatRepair 方法, 通过先将测试失败的信息提供给大语言模型, 并利用之前修补类似错误的失败或成功案例进行学习, 以对话方式实现程序修复. 通过这些方法的不断改进, 大语言模型在程序修复任务中的表现得到了显著提升.

1.2.3 检索增强生成

检索增强生成 (retrieval-augmented generation, RAG) 是当前热门的大语言模型前沿技术之一, 旨在提升生成模型的能力与准确性. 该方法结合信息检索和生成模型的技术, 通过在生成过程中引入外部知识库或文档, 提供更丰富的上下文信息, 从而丰富大语言模型的知识基础. 这种策略有效改善了模型在特定任务中的表现, 使其在处理复杂问题时更具优势.

2020 年, Lewis 等人[29]首次提出了检索增强生成 (RAG) 框架, 将信息检索与序列生成相结合. 该方法先检索与输入相关的文档, 然后将这些文档与输入一起输入到生成模型中, 有效扩展了模型的知识范围, 提高了回答的准确性和多样性. 近年来, 许多研究者在软件工程领域将检索增强生成技术应用于代码相关任务. Zhou 等人[30]提出了 DocPrompting 方法, 给定自然语言描述意图, 通过检索器从文档池中获取相关文档, 并利用这些文档与意图一起生成代码, 从而提高代码生成效果. Lu 等人[31]提出了基于检索增强生成的代码补全框架 ReACC, 该框架由源代码检索器和自回归语言模型组成. ReACC 在给定待补全代码片段时, 首先结合余弦相似度和 BM25 算法, 从源代码数据库中检索语义相似的代码, 并将其与待补全代码拼接, 生成完整代码. Zhang 等人[32]提出了库级别代码补全框架 RepoCoder, 该框架利用迭代式检索生成范式, 首先使用待补全代码执行检索生成中间补全代码, 然后基于中间结果进行第 2 次检索, 减少了检索上下文和目标代码补全之间的差距, 进一步提高最终代码生成效果. 此外, Luan 等人[33]提出了 Aroma 方法, 通过结构化代码搜索进行代码推荐. 该方法首先对包含数千个开源项目的大型

代码库进行索引, 然后根据给定的代码片段在构建的代码库中搜索相关方法体, 最后对搜索出的多个方法体进行聚类和交叉, 得到最终推荐的一组方法体.

1.2.4 提示工程

提示工程 (prompt engineering) 是一种通过设计和优化输入提示 (prompt) 来引导大语言模型生成所需输出的技术. 它的主要目标是在与大语言模型进行交互时, 通过精心设计的提示, 使模型能够准确理解任务要求, 并生成高质量的响应, 从而实现高效的人机交互. 通过设计合适的提示, 提示工程可以显著提升模型在文本生成、问题回答、代码生成、翻译等任务中的表现.

近年来, 研究者们提出了一系列优化 prompt 的策略, 以提升大语言模型的能力. Wei 等人[34]提出了思维链 (chain-of-thought, CoT) 策略, 通过将问题分解为一系列逻辑步骤, 引导大语言模型逐步完成中间推理过程, 从而提高模型在解决复杂问题时的表现, 并让问题的解决过程更加系统和透明. White 等人[35]提出了结构化提示模式的框架, 这一框架通过将提示按照一定的逻辑结构组织成结构化形式, 让模型能够更清晰地理解任务要求, 并生成高质量的回答. Xu 等人[36]提出了 ExpertPrompting 方法, 旨在挖掘大语言模型作为杰出专家回答问题的潜力. 该方法利用上下文学习自动合成每个特定指令的专家身份, 并要求大语言模型以特定专家身份回答问题, 从而提升模型的回答效果. 此外, Brown 等人[7]提出了少样本学习 (few-shot learning) 策略, 通过在提示中提供少量示例, 让模型能够学习任务的结构和要求, 从而在执行任务时表现得更加准确和有效. Zhao 等人[37]进一步研究发现, 少样本学习存在不稳定因素, 即样本的选择和顺序都会显著影响大语言模型的准确度. 通过这些不同的优化策略, 提示工程在不断推动大语言模型的能力提升, 扩大了其在各类任务中的应用范围. 本文也在上述研究基础上, 在构建 prompt 时, 使用了结构化指令、思维链、设定专家身份、少样本学习等策略, 帮助大语言模型理解问题需求, 充分挖掘大语言模型的能力, 以提升其在故障复现测试用例生成任务上的效果.

2 关于 GitHub 问题报告故障复现测试用例不足问题的实证研究

为了深入研究 GitHub 问题报告故障复现测试用例不足问题, 本文在 SWE-bench Lite 数据集上进行了实证研究. 本节首先介绍实证研究的方法设计, 然后介绍实证研究的数据来源, 最后介绍实证研究的结果.

2.1 实证研究的方法设计

本文希望通过对 SWE-bench Lite 数据集中 300 个 GitHub 问题报告进行实证研究, 探索 GitHub 问题报告的故障复现测试用例的数量情况及构建方式. 为此, 本节围绕以下两个核心研究问题 (research question, RQ) 展开研究.

RQ1: GitHub 问题报告在提交时, 缺乏故障复现测试用例的比例如何?

RQ2: 若 GitHub 问题报告在提交时没有故障复现测试用例, 开发人员以什么样的方式生成故障复现测试用例?

为探究 RQ1, 本文基于 SWE-bench Lite 数据集, 对 300 个 GitHub 问题报告在提交时是否已包含故障复现测试用例进行了统计与分析. 具体而言, 首先, 通过数据集中的 base_commit 字段, 将问题报告所在的代码仓库回溯至问题报告提交时的代码版本, 以获取问题报告提交时的真实软件环境, 从而避免后续代码修改对测试用例状况的影响. 在此基础上, 进一步检查 base_commit 版本下是否已存在与该问题报告相关的故障复现测试用例. 具体方法是分析 FAIL_TO_PASS 字段, 判断其中列出的测试函数是否已在 base_commit 版本中存在. 如果该版本下至少包含一个与问题报告相关的故障复现测试用例, 则认为该问题报告在提交时已具备故障复现测试用例; 反之, 则判定该问题报告提交时缺乏故障复现测试用例. 最后, 本文统计了 300 个问题报告中缺乏故障复现测试用例的问题报告数量, 并计算其占比, 以量化 GitHub 问题报告在提交时的故障复现测试用例覆盖情况.

为探究 RQ2, 本文针对 RQ1 中发现的提交时缺乏故障复现测试用例的问题报告, 进一步研究其在修复 commit 提交后新增的故障复现测试用例, 旨在探讨问题报告贡献者是如何生成这些故障复现测试用例的. 具体而言, 首先, 通过对比 base_commit 状态的代码仓库版本与问题报告关联的 commit (patch 字段) 提交后的代码仓库版本,

确定 FAIL_TO_PASS 字段中新增加的故障复现测试用例. 随后, 通过分析 patch 中对测试文件的修改, 并比对 FAIL_TO_PASS 中新增的故障复现测试用例, 判断这些新增的测试用例是完全新编写的, 还是基于现有测试用例进行修改得到的. 最后, 本文统计所有新增故障复现测试用例的构建方式比例, 以分析 GitHub 问题报告贡献者在修复缺失的故障复现测试用例时, 更倾向于采用全新编写还是修改现有测试用例的方式. 通过这一统计结果, 本文将深入探讨开发者在实际开发过程中如何生成故障复现测试用例, 从而为优化故障复现测试用例的生成方法提供理论依据.

2.2 实证研究的数据来源

本文实证研究数据来源于 SWE-bench Lite 数据集, 后续的实验方法也是基于该数据集进行的. SWE-bench 数据集包含来自 12 个流行 Python 仓库的 2 294 个问题报告-commit 对. SWE-bench Lite 数据集是 SWE-bench 数据集的构建者对其 2 294 条数据进行筛选优化后得到的精简子集, 专注于评估独立的功能性错误修复. 该数据集在筛选过程中严格控制数据质量, 确保所保留的实例具有代表性和可靠性. 最终, SWE-bench Lite 数据集涵盖了来自 12 个流行 Python 开源仓库的 300 个问题报告数据实例, 涉及不同类型的故障及其对应的修复信息, 在一定程度上可以反映真实的 GitHub 仓库情况, 其在 Hugging Face 网站上的下载量达 59.8k (SWE-bench 数据集的下载量为 55.4k), 反映出了 SWE-bench Lite 数据集具有一定的可靠性, 进而让本文实证研究得到可靠的结果. SWE-bench Lite 数据集中各个仓库的数据分布如图 1 所示.

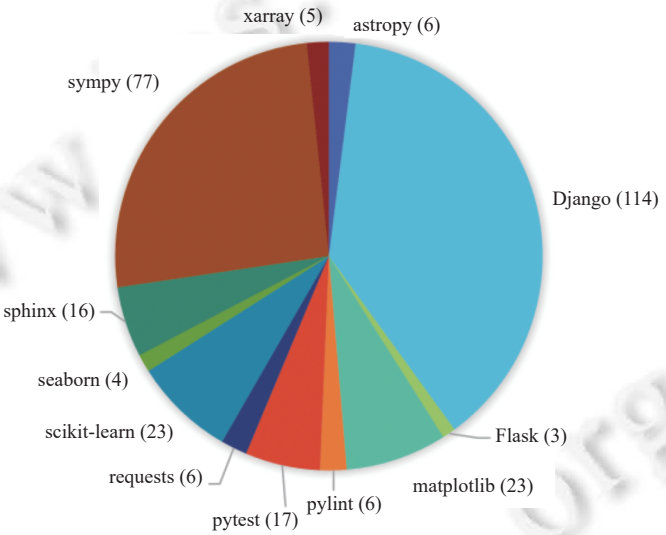

图 1 SWE-bench Lite 数据集中各仓库数据分布

SWE-bench Lite 数据集中的每一条数据均来源于 GitHub 中的一个问题报告及其相关的 commit, 并且该 commit 成功解决了该问题报告. 每条数据由多个部分组成, 包括: instance_id、repo、created_at、version、base_commit、hints_text、patch、test_patch、problem_statement、FAIL_TO_PASS 和 PASS_TO_PASS、environment_setup_commit.

各个部分的具体含义如表 1 所示. 其中, base_commit、FAIL_TO_PASS 和 patch 这 3 个字段是本文实证研究的核心依据. base_commit 记录了代码仓库在问题报告提交时的状态, 可用于回溯代码版本, 进而判断问题报告在提交时是否已包含故障复现测试用例. FAIL_TO_PASS 字段包含该问题报告关联的 commit 提交后, 所有相关的故障复现测试用例集合, 并列出所有故障复现测试用例的函数名, 可作为衡量问题报告是否具备故障复现测试用例的比对基准. patch 字段记录了该问题报告关联 commit 的修复代码补丁, 涵盖对源代码文件及测试文件的修改信息. 通过分析 patch 字段在测试文件中的修改位置, 可以判断故障复现测试用例是问题报告贡献者在修复问题

报告时新编写的, 还是基于仓库中已有测试函数进行修改的. 基于上述 3 个关键字段, 本文将围绕 GitHub 问题报告的故障复现测试用例是否充足这一问题展开实证研究, 以探讨现有测试用例的覆盖情况及其改进方向.

表 1 SWE-bench Lite 数据集中字段含义

字段名称	字段含义
instance_id	格式化的数据实例标识符, 通常为repo_owner__repo_name-PR-number
repo	问题报告在GitHub中所在仓库, 格式为owner/name
created_at	问题报告所在pull request (PR)创建的日期
version	用于运行评估的安装版本
base_commit	在问题报告解决前, 问题报告所在仓库的commit hash
hints_text	在解决方案PR首次提交日期之前对问题报告所发表的评论
patch	解决方案PR生成的补丁
test_patch	解决方案PR生成的测试文件补丁
problem_statement	问题报告的标题和正文
FAIL_TO_PASS	一个测试用例字符串列表, 测试用例在解决方案PR应用前不通过, 应用后通过, 用来验证问题报告是否解决
PASS_TO_PASS	一个测试用例字符串列表, 测试用例在解决方案PR应用前后都应通过
environment_setup_commit	环境设置和安装的commit hash

2.3 实证研究的结果分析

在深入探讨两个研究问题之前, 本文首先对 SWE-bench Lite 数据集中的问题报告及其对应的故障复现测试用例进行了整体分析. 该数据集中的每个问题报告均可通过修改单个代码文件进行修复, 反映了问题报告本身具有适中的复杂程度. 此外, 与这些问题报告相关的故障复现测试用例均为单元测试, 体现出其关注代码层面功能的精细验证. 进一步统计发现, 这些故障复现测试用例平均长度为 19.5 行代码, 平均涉及约 2.4 个外部库或内部 API 的调用, 表明其在结构上具有一定的实现复杂度, 需要调用多个组件协同工作. 这些特征为自动化生成故障复现测试用例提出了挑战, 也为本文方法的设计提供了依据.

针对研究问题 1 (RQ1), 本文对 SWE-bench Lite 数据集进行分析, 统计发现该数据集 300 个问题报告中, 有 268 个问题报告在提交时没有故障复现测试用例, 占比高达 89.33%. 这一现象反映了 SWE-bench Lite 数据集中问题报告存在故障复现测试用例不足的问题, 也映射出整个 GitHub 问题报告中可能普遍存在类似问题. 许多问题报告在提交时缺乏相关的故障复现测试用例, 这可能导致开发人员在修复问题报告时无法准确复现和验证问题, 从而影响软件质量的提升.

进一步分析 GitHub 问题报告缺乏故障复现测试用例的原因, 可以归结为两个主要因素. 首先, GitHub 问题报告机制本身并未明确要求问题报告提交者在提交时提供相应的故障复现测试用例. 这一机制的缺失使得问题报告提交者缺乏明确的动机和指引去准备详尽的故障复现步骤. 其次, 编写故障复现测试用例往往是一个相对耗时且复杂的过程, 涉及对问题的深入理解和对测试环境的精确还原. 因此, 许多问题报告提交者倾向于直接提出问题, 而不愿意投入额外的时间和精力来编写相应的验证方法. 这种情况下, 故障复现测试用例的缺失成为影响问题修复效率和质量的一个重要因素. 基于此, 本文将深入研究面向 GitHub 问题报告的故障复现测试用例生成方法, 探索如何自动生成适用于 GitHub 问题报告的故障复现测试用例, 以解决这一关键问题.

针对研究问题 2 (RQ2), 本文对问题报告提交后新生成的故障复现测试用例进行了分析. 结果显示, 在 300 个问题报告中, 共有 440 个新生成的故障复现测试用例. 其中, 179 个故障复现测试用例是通过修改代码仓库中已有的测试函数生成的, 占比 40.68%; 261 个故障复现测试用例是由开发人员全新编写的, 占比 59.32%. 这一结果表明, 在为问题报告生成故障复现测试用例时, 几乎有一半的机会可以通过修改现有的测试函数来实现, 反映出了现有相关测试用例对于生成故障复现测试用例的作用.

进一步分析这一现象, 本文发现可以通过复用原有测试文件中的 import 语句和已有的测试用例来生成新的

故障复现测试用例. 具体来说, 现有测试用例中的 import 语句为测试环境提供了必要的上下文 API 信息, 而测试用例本身则构建了与特定故障相关的验证逻辑. 因此, 在生成新的故障复现测试用例时, 只需在现有基础上进行适当的调整或扩展, 就可以快速生成适用于新问题报告的故障复现测试用例. 这一现象表明, 现有的测试用例对于复现一些常见故障具有较强的适用性, 开发人员可以通过修改和优化现有测试函数来减少编写新测试用例的工作量.

因此, 本文提出了一种将检索增强生成与大语言模型相结合的方法. 在设计时, 充分考虑了可以复用现有测试用例的特点, 提出通过检索与给定问题报告相关的 import 语句以及测试用例样本, 并将其作为参考信息, 辅助大语言模型生成更准确的故障复现测试用例. 这种方法能够有效地加快新故障复现测试用例的生成速度, 并提高其准确性, 从而帮助开发人员更高效地解决问题. 对于 59.32% 的开发人员新编写的故障复现测试用例的问题报告实例, 本文方法也可通过检索与问题报告相关性较高的代码上下文, 辅助大语言模型生成故障复现测试用例, 因此本文方法可应用于所有的问题报告实例.

3 基于大语言模型和多元检索增强生成的故障复现测试用例生成方法

基于实证研究中发现的 GitHub 问题报告故障复现测试用例不足问题, 以及开发人员在为问题报告编写故障复现测试用例时, 有 40.68% 的情况是基于现有测试函数进行修改, 本文提出了一种基于大语言模型和多元检索增强生成的故障复现测试用例生成方法, 旨在为 GitHub 问题报告等类似的软件开发问题生成故障复现测试用例. 该方法的核心思想是基于给定的问题报告内容和代码仓库, 挖掘多元的上下文信息, 包括报错根函数、import 语句和测试用例样本等, 通过使用提示工程技术, 构建 prompt 来引导大语言模型生成故障复现测试用例.

本文方法的框架如图 2 所示, 输入为一个 GitHub 问题报告及其所在代码仓库, 输出为复现问题报告内容并验证问题报告是否解决的故障复现测试用例. 具体而言, 本文方法主要分为 4 个步骤: 报错根函数定位、测试文件选择与 import 语句抽取、基于相似度计算的测试用例样本选取, 以及最终的故障复现测试用例生成. 通过这些步骤, 本文方法有效获取了与问题报告相关的上下文信息, 并将其整合在 prompt 中, 增强了大语言模型在故障复现测试用例生成中的表现.

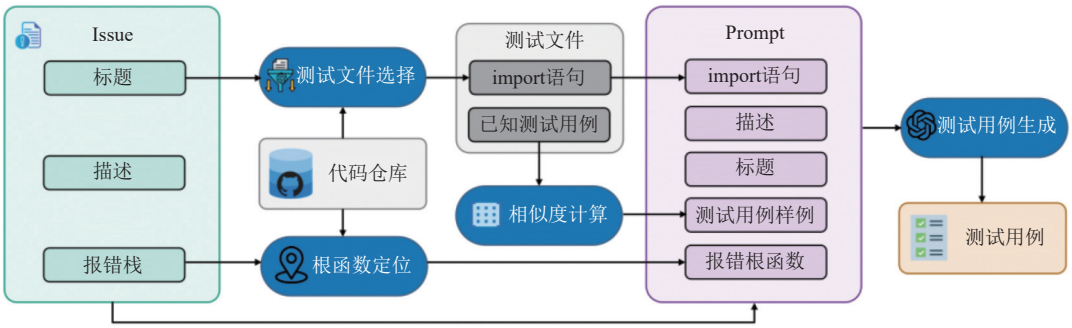

图 2 基于大语言模型和多元检索增强生成的故障复现测试用例生成方法框架图

下面是对上述 4 个步骤的方法概述.

(1) 报错根函数定位方法概述: GitHub 问题报告在某些情况下会包含报错栈信息, 这些信息在一定程度上能够揭示故障发生的原因. 因此, 本文通过分析 GitHub 问题报告中的报错栈内容, 定位出错的函数. 具体方法为: 对于给定的 GitHub 问题报告, 首先利用正则表达式规则抽取问题报告描述中的错误栈信息, 然后解析错误栈中的调用链, 识别出最终在代码仓库中出错的函数所在位置. 根据该函数路径信息, 进一步抽取出对应的函数代码内容. 这些抽取出的函数代码可能揭示故障发生的关键区域, 有助于定位故障发生的根本原因, 并为后续的故障复现测试用例生成提供重要的上下文信息.

(2) 测试文件选择与 import 语句抽取方法概述: 为了让大语言模型能够更好地理解并利用 GitHub 问题报告所在代码仓库中的现有方法与外部依赖库, 在为 GitHub 问题报告生成故障复现测试用例时, 本文进一步提出了检索与问题报告内容最相关的测试文件, 并从中抽取 import 语句的方法. import 语句信息有助于大语言模型理解项目中的 API 调用方式和外部依赖, 从而提高生成测试用例的质量. 具体而言, 本文首先根据启发式规则抽取 GitHub 问题报告所在代码仓库中的所有测试文件, 包括每个测试文件的文件名及其路径. 然后, 将所有测试文件名组成一个集合, 并通过大语言模型的理解能力选择出与问题报告最相关的测试文件. 在选择过程中, 采用多次生成的方式, 最终选取出现频率最高的测试文件作为最相关的测试文件. 最后, 使用正则表达式规则抽取选定测试文件中的所有 import 语句. 通过这一方法, 本文为大语言模型提供了与给定 GitHub 问题报告相关的 API 接口上下文信息, 帮助模型更好地理解仓库中的现有方法和外部依赖, 从而生成更准确的故障复现测试用例.

(3) 基于相似度计算的测试用例样本选取方法概述: 考虑到 GitHub 问题报告通常来自真实软件开发过程, 具有较高的复杂性, 本文进一步通过在现有代码仓库中检索与问题报告内容相关的测试函数作为样例, 帮助大语言模型理解特定代码仓库中测试用例写法, 从而提升为 GitHub 问题报告生成故障复现测试用例的效果. 具体而言, 本文首先根据检索出的与问题报告最相关的测试文件, 利用程序分析技术抽取测试文件中的所有测试函数. 接着, 采用基于 embedding 的相似度计算方法, 计算每个测试函数与给定问题报告之间的相似度. 最后, 根据相似度得分对测试函数进行排序, 筛选出与当前问题报告最相关的 3 个测试函数作为样例, 提供给大语言模型, 用于生成故障复现测试用例. 通过这一方法, 本文进一步提高了生成的测试用例的准确性和适用性.

(4) 故障复现测试用例生成方法概述: 在检索到报错根函数、import 语句以及测试用例样本后, 本文将这些检索到的信息整合进生成测试用例的 prompt 中. 为进一步提升大语言模型的生成效果, 本文采用了多种提示工程方法, 包括角色扮演、思维链、few-shot 学习等. 这些方法通过精心设计的提示, 引导大语言模型理解任务要求, 准确生成能够复现并验证 GitHub 问题报告是否解决的故障复现测试用例.

3.1 报错根函数定位

当 GitHub 问题报告涉及代码报错问题时, 描述中通常会包含错误栈, 反映出代码的报错调用链. 本文将报错调用链中最后一个在代码仓库中的函数定义为报错根函数. 如图 3 所示, 展示了 Django 仓库[38]的一个问题报告描述, 其中包含了相关的错误栈信息. 基于启发式经验, 本文认为只有在问题报告所在仓库中的函数才可能是导致问题的原因, 因此不考虑第三方库调用的函数. 例如, 最后一行的 urlsplit 函数属于第三方库 urllib. 因此, 该问题报告的报错根函数为 Django/core/validators.py 文件中的__call__函数.

```
URLField throws ValueError instead of ValidationError on clean
Description

forms.URLField( ).clean('////]@N.AN')
results in:
        ValueError: Invalid IPv6 URL
        Traceback (most recent call last):
         File "basic_fuzzer.py", line 22, in TestOneInput
         File "fuzzers.py", line 350, in test_forms_URLField
         File "django/forms/fields.py", line 151, in clean
         File "django/forms/fields.py", line 136, in run_validators
         File "django/core/validators.py", line 130, in __call__
         File "urllib/parse.py", line 440, in urlsplit
```

图 3 GitHub 问题报告描述中错误栈举例图

本文采用基于规则的方法来实现报错根函数的定位, 分为两步: 抽取报错根函数信息和根函数定位. 在抽取报错根函数信息阶段, 首先使用字符串处理方法识别问题报告描述中的“Traceback”字符串, 以判断是否包含错误栈信息, 并提取所有相关的错误栈内容. 接着, 利用正则表达式逐行解析错误栈, 提取每一行中的报错文件、报错代码行和报错函数信息. 最后, 过滤掉第三方库调用的函数, 仅保留最后一个报错函数作为报错根函数. 以图 3 中的错误栈信息为例, 本文抽取出的报错文件为 Django/core/validators.py, 报错代码行为第 130 行, 报错根函数

为__call__函数. 在根函数定位阶段, 首先将问题报告所在的仓库克隆到本地, 并根据该问题报告的 base_commit 信息切换到问题报告提出时的版本. 随后, 使用 tree-sitter 工具[39], 根据抽取出的报错文件和报错根函数名, 提取报错根函数的内容. 如图 4 所示, 依据图 3 中抽取的报错根函数信息, 定位到 Django/core/validators.py 文件中的__call__函数, 具体报错位置为第 130 行. 至此, 本文将整个__call__函数的内容作为报错根函数信息, 后续将用于构建 prompt, 为大语言模型提供问题报告的报错信息.

```
128         else:
129             # Now verify IPv6 in the netloc part
130             host_match = re.search(r'^\[(.+)\](?::\d{1,5})?$', urlsplit(value).netloc)
131             if host_match:
132                 potential_ip = host_match[1]
133                 try:
134                     validate_ipv6_address(potential_ip)
135                 except ValidationError:
136                     raise ValidationError(self.message, code=self.code, params={'value': value})
```

图 4 报错根函数定位示意图

此外, 本文进一步研究发现, SWE-bench Lite 数据集中, 300 个问题报告中并不都存在错误栈信息, 仅有 58 个问题报告包含错误栈信息, 占比 19.3%. 因此, 本文方法不依赖于问题报告中的错误栈信息, 而是仅将错误栈视为参考信息, 辅助大语言模型定位故障问题的原因, 为生成测试用例提供支持. 若问题报告中包含错误栈, 本文将使用静态分析工具抽取错误栈相关代码上下文, 提供给大语言模型. 若不存在错误栈信息, 本文检索增强生成策略中添加了额外的上下文信息 (import 语句、测试用例样本) 辅助大语言模型理解问题报告. 通过这样的方式, 本文方法弥补了 GitHub 问题报告中错误栈缺失问题, 提升了大语言模型生成故障复现测试用例的效果.

3.2 测试文件选择与 import 语句抽取

在第 2 节的实证研究中, 本文发现开发人员在编写故障复现测试用例以复现问题报告并验证问题报告是否解决时, 有 40.68% 的情况是基于现有测试函数进行修改的. 这一发现表明, 代码仓库中现有的测试函数蕴含着一定的有效信息, 能够为开发人员提供有价值的参考. 利用这些已有测试用例, 开发人员不仅可以节省编写新故障复现测试用例的时间, 还能有效提升开发效率.

基于这一发现, 本文设计了测试文件选择与 import 语句抽取的方法, 旨在通过检索与问题报告最相关的测试文件, 充分利用其中的现有信息, 从而提升大语言模型的故障复现测试用例生成效果. 在这一过程中, 测试文件中的 import 信息是本文重点关注的内容, 因为大语言模型在面对庞大的代码仓库时, 往往难以理解其文件结构, 这可能导致无法正确调用仓库内的 API. 因此, 在选择相关测试文件后, 本文将抽取其 import 语句, 并将其作为检索增强生成的内容, 纳入后续测试用例生成的 prompt 中, 以帮助大语言模型更准确地调用仓库内的 API. 当大语言模型生成故障复现测试用例后, 本文将在所选测试文件的相同路径下创建新的测试文件, 将生成的测试用例放入其中. 由于抽取出的 import 语句代表了代码仓库中正确的调用方式, 这样的处理能够有效避免潜在的 import 语句调用语法错误, 从而提高生成故障复现测试用例的准确性和可靠性.

3.2.1 测试文件选择

考虑到大语言模型具备强大的语义理解能力, 本文将测试用例选择这一粗粒度的检索任务交由大语言模型处理. 具体步骤包括测试文件的抽取、文件路径的截取、prompt 的构建, 以及结果的多级选择. 这一方法旨在高效挖掘与问题报告相关的测试文件, 从而为后续的故障复现测试用例生成提供支持.

在测试文件抽取阶段, 本文对 SWE-bench Lite 数据集中的 12 个代码仓库逐个进行了分析, 发现每个仓库的测试文件在存储位置上具有相对规范和一致的特点. 为了避免将代码文件错误地抽取出来, 本文为每个代码仓库人工设计了一套测试文件判断规则, 如表 2 所示. 基于这些规则, 本文遍历每个代码仓库中的所有 Python 文件, 抽取出所有符合条件的测试文件路径.

表 2 SWE-bench Lite 数据集中各个代码仓库测试文件判断规则

仓库名	测试文件判断规则
scikit-learn/scikit-learn astropy/astropy matplotlib/matplotlib sympy/sympy pydata/xarray	文件路径中包含“/tests/”字段
pytest-dev/pytest	在代码仓库的testing文件夹下
pallets/Flask Django/Django pylint-dev/pylint mwaskom/seaborn sphinx-doc/sphinx	在代码仓库的tests文件夹下
psf/requests	—

在测试文件路径截取阶段, 由于每个测试文件的路径较长且前缀通常相同, 为了在保留文件路径语义的基础上尽量减少重复或无意义的信息, 本文对每个测试文件的路径进行处理. 具体方法是按照“/”字符对路径进行切割, 保留最后 3 个字符串, 然后再用“/”将它们拼接在一起, 形成一个更简洁且具代表性的路径信息. 这一处理方式有效地提升了测试文件路径的简洁性, 同时保留了关键的语义信息. 所有截取后的测试文件路径将以字符串列表的形式进行存储, 以便于后续的处理与检索.

在 prompt 构建阶段, 本文对多个 GitHub 问题报告进行了实验, 以探究在测试文件选择过程中, 使用问题报告的哪一部分内容作为参考最为有效. 研究发现, 使用问题报告标题的效果最佳. 这主要是因为 GitHub 问题报告的格式较为自由, 其具体描述通常较为复杂, 可能包含大量干扰信息. 相比之下, 问题报告标题往往能够有效提炼出关键信息, 提供更为直接的上下文. 因此, 本文决定仅将问题报告标题作为依据, 避免将所有描述信息纳入 prompt, 以提升测试文件选择的准确率. 具体而言, 本文将 GitHub 问题报告的标题与经过截取的测试文件路径列表作为输入, 结合适当的提示信息, 构建结构化的 prompt. 这样的设计使得大语言模型能够更清晰地理解任务要求, 并有效选择出最有可能用于测试该问题报告的 n 个测试文件路径. 为确保结果的准确性和一致性, 输出结果也被要求按照特定格式呈现.

prompt 1. 在测试文件列表中选择 n 个最可能测试给定问题报告的测试文件.

This is an issue title: {issue_title}.
Below I will give you a list of test files in the project path, please find out n paths most likely to test this issue.
Please note that what you need to find is the test file that tests the issue, not the most relevant file for the issue.
if n=1, the answer format is: “Most likely related path is: ‘xxx/xxx/xxx.py’”
if n=2, the answer format is: “Most likely related path is: ‘xxx/xxx/xxx.py’, ‘xxx/xxx/xxx.py’”
The list is:
{file_paths}

由于各个代码仓库的测试文件数量较多, 本文设计了多级选择策略来优化测试文件的选取过程. 具体而言, 对于截取后的测试文件路径, 本文将其按照每 300 个路径为一组进行分组, 并以字符串列表的形式将每组路径放入 prompt 1 中的 file_paths 字段, 同时将 n 设置为 1, 以便让大语言模型从每一组中选出一个测试文件. 接着, 本文将模型在每组测试文件中选择的结果放入一个新的空列表中, 然后再次将该列表放入 prompt 1 中, 将 n 设置为 2, 并让模型重复选择 5 次, 每次的结果均存入列表中. 最后, 统计列表中出现次数最高的测试文件路径, 并将其作为最终结果. 这一策略有效地提高了选择的准确性, 并减小了 prompt 的长度, 确保所选测试文件能够更好地满足测试问题报告的需求.

图 5 展示了一个具体的测试文件选择 prompt 的例子, 该例子中的问题报告来源于 Django 仓库, 标题是

“URLField 报错内容有误”. 为此, 本文首先对 Django 仓库的所有测试文件进行了抽取, 并对每个测试文件的路径进行了截取处理, 将保留后的路径信息以列表形式嵌入到 prompt 中. 在这个例子中, 测试文件路径列表中包含了正确的测试文件路径——forms_tests/field_tests/test_urlfield.py. 大语言模型在接收到该 prompt 后, 成功选择了这一正确答案. 这一结果不仅验证了大语言模型在处理粗粒度文件检索任务时的有效性, 也展示了通过合适的 prompt 设计, 可以显著提高模型的选择准确性.

```
This is an issue title: "URLField throws ValueError instead of ValidationError on clean".
Below I will give you a list of test files in the project path, please find out only one path most
likely to test this issue.
Please note that what you need to find is the test file that tests the issue, not the most relevant
file for the issue.
The answer format is: "Most likely related path is : 'xxx/xxx/xxx.py'"
The list is:
['forms_tests/field_tests/test_booleanfield.py','forms_tests/field_tests/test_filefield.py','forms_
tests/field_tests/test_uuidfield.py','forms_tests/field_tests/test_urlfield.py','tests/urlpatterns/
path_urls.py','tests/urlpatterns_reverse/urls.py','tests/utils_tests/test_autoreload.py','tests/fie
ld_defaults/tests.py','tests/field_defaults/tests.py','tests/forms_tests/tests/test_forms.py',...]
```

图 5 测试文件选择 prompt 示意图

3.2.2 import 语句抽取

故障复现测试用例生成任务面临的一个主要挑战在于, 输入的上下文是整个代码仓库, 这往往导致上下文信息过长, 使得大语言模型在理解整个文件结构时变得困难. 这种情况下, 模型可能无法正确调用仓库中已有的 API 接口, 从而影响生成故障复现测试用例的质量. 为了解决这一问题, 本文在选择出最可能测试给定问题报告的测试文件后, 特别提取了该测试文件中的 import 语句. 这一做法的双重目的在于: 一方面, 提供给大语言模型可能使用的 API 信息, 增强其调用的准确性; 另一方面, 通过分析 import 语句中的引用路径, 帮助模型更好地理解代码仓库的结构和关系. 这样, 模型在生成测试用例时能够更加准确地定位和利用相关的 API 功能, 提高了故障复现测试用例生成的有效性与质量.

为了有效提取测试文件中的 import 语句, 本文采用了正则表达式匹配技术. 具体流程如下: 首先, 利用大语言模型选择出的截取后的测试文件路径, 将其还原为完整的文件路径. 接着, 根据问题报告的 base_commit 信息, 将克隆后的代码仓库恢复到问题报告提出时的版本. 这一过程确保能够准确定位到当时的代码状态. 随后, 根据恢复的完整文件路径, 找到对应的测试文件, 并将该文件的内容全部读取为一个字符串. 最后, 通过正则表达式匹配技术, 本文从中抽取出所有的 import 语句.

在图 5 的基础上, 当大语言模型选出了 test_urlfield.py 这个测试文件后, 本文利用正则表达式匹配技术成功抽取了该测试文件中的所有 import 语句, 如图 6 所示. 抽取出的 import 语句不仅包含了如何正确调用 URLField 类, 还涉及了问题报告标题中提到的 ValidationError 类的调用方法. 这些 import 语句与给定的问题报告高度相关, 为大语言模型在生成故障复现测试用例时提供了重要的参考信息. 在大语言模型生成故障复现测试用例后, 本文只需在该测试文件的相同路径下创建一个新的测试文件, 将生成的故障复现测试用例放入其中. 由于已抽取的 import 语句包含了所需的 API, 这样的处理避免了潜在的语法错误. 因此, 本文通过这一方式, 旨在降低大语言模型在生成故障复现测试用例时出现的 API 调用语法错误, 从而提高故障复现测试用例生成的准确性和有效性.

```
from django.core.exceptions import ValidationError
from django.forms import URLField
from django.test import SimpleTestCase
from . import FormFieldAssertionsMixin
```

图 6 import 语句抽取结果示意图

3.3 基于相似度计算的测试用例样本选取

在选出最可能测试给定问题报告的测试文件后, 本文采用基于相似度计算的方法在该测试文件中选取测试用

例样本, 旨在从已有的代码仓库中检索出与测试给定问题报告相关的测试函数. 这些提取出的测试函数将作为检索增强生成的内容纳入 prompt 中, 为大语言模型提供可参考的测试用例, 从而提升其生成故障复现测试用例的效果. 该方法具体分为两个主要步骤: 首先是测试函数的抽取, 其次是基于相似度计算的测试用例样本选取, 最终选取与问题报告相关性最高的 3 个测试函数.

在测试函数抽取阶段, 前期的准备工作与 import 语句的抽取阶段相似. 首先, 将大语言模型选择出的截取后的测试文件路径还原为完整路径, 并根据问题报告的 base_commit 信息将克隆后的代码仓库恢复到问题报告提出时的版本. 接下来, 根据还原后的文件路径定位到相应的测试文件, 并将该文件的全部内容读取为一个字符串. 获得测试文件中的完整代码后, 本文使用 tree-sitter 工具对该代码字符串进行解析. 通过分析各个代码段的节点属性, 成功提取出测试文件中所有的测试函数. 这些提取的函数片段将以字符串列表的形式保存, 便于后续的相似度计算与排序过程.

在基于相似度计算的测试用例样本选取阶段, 考虑到抽取出的测试函数与问题报告内容之间最重要的是语义相关性, 而非单纯的字符相似性, 本文采用了稠密检索方法中的基于 embedding 的相似度计算. 具体而言, 本文对抽取出的每个测试函数的代码字符串与问题报告标题之间计算 embedding 相似度得分. 根据这些相似度得分, 对抽取出的测试函数进行排序, 最终选取得分最高的 3 个测试函数作为测试用例样本. 这种方法有效地确保了所选样本在语义上的相关性, 从而为后续的故障复现测试用例生成提供了更为精准的参考.

如图 7 所示, 该例子展示了本文抽取的测试用例样本, 依然以上文 Django 仓库的问题报告为例, 提问为问题报告的标题, 即“URLField 报错内容有误”. 在测试文件选择阶段选出的 test_urlfield.py 文件中, 将所有测试函数与问题报告标题进行了 embedding 相似度计算, 最终得分最高的 3 个测试函数如图 7 所示. 其中, 相似度得分最高的测试函数为 test_urlfield_clean_invalid, 该函数定义了一个 URLField 类, 并设计了多个非法 URL 列表, 以测试调用 clean 方法时是否会输出 ValidationError. 这一测试函数恰好与问题报告标题中所表达的问题高度相关. 因此, 可以得出结论, 代码仓库中已有的测试函数往往包含与特定问题报告相关的内容, 通过有效地抽取这些测试函数并提供给大语言模型, 能够帮助其更好地理解如何生成符合需求的故障复现测试用例代码.

Query: URLField throws ValueError instead of ValidationError on clean

Top1:

```
def test_urlfield_clean_invalid(self):
    f = URLField()
    tests = ['foo','com.',
    ...
    ]
    msg = "'Enter a valid URL.'"
    for value in tests:
        with self.subTest(value=value):
            with self.assertRaisesMessage(ValidationError, msg):
                f.clean(value)
```

Top2:

```
def test_urlfield_clean_required(self):
    f = URLField()
    msg = "'This field is required.'"
    with self.assertRaisesMessage(ValidationError, msg):
        f.clean(None)
    with self.assertRaisesMessage(ValidationError, msg):
        f.clean('')
```

Top3:

```
def test_urlfield_clean_not_required(self):
    f = URLField(required=False)
    self.assertEqual(f.clean(None), '')
    self.assertEqual(f.clean(''), '')
```

图 7 测试用例样本示意图

3.4 故障复现测试用例生成

随着大语言模型的不断发展, 其在代码理解和生成任务上的表现日益出色. 与此同时, 提示工程技术在应用大语言模型时发挥着重要作用, 通过精心设计的提示, 引导模型产生期望的输出. 基于这一背景, 本文使用了大语言模型和提示工程技术生成面向 GitHub 问题报告的故障复现测试用例, 所选用的大语言模型为 OpenAI 提供的 GPT-4 模型. 具体而言, 如上文所述, 本文首先检索与给定问题报告相关的报错根函数 (如问题报告描述中包含错误栈)、import 语句以及测试用例样本. 随后, 结合提示工程领域的相关策略, 构建一个引导大语言模型生成故障复现测试用例的 prompt. 最终, 将该 prompt 输入大语言模型, 以生成复现给定问题报告并验证问题报告是否解决的故障复现测试用例.

在设计 prompt 时, 本文采用了多种提示工程领域的策略, 以提升大语言模型的理解能力和生成质量. 如 prompt 2 所示, 首先, 本文将 prompt 设计为结构化形式, 这样能够更清晰地描述多个方面的内容, 增强模型对问题的理解. 其次, 本文采用了 few-shot 方法, 通过在 prompt 中提供与期望输出类似的例子, 帮助大语言模型更好地理解需求, 从而生成高质量的结果. 在 prompt 2 中, 本文包含了一个 example_tests 字段, 列出了在代码仓库中检索出的与给定问题报告相关的 3 个测试函数, 作为已有测试用例的参考信息. 此外, 本文还运用了大语言模型角色扮演策略, 明确告诉模型它是一位软件测试专家, 擅长根据需求生成故障复现测试函数, 通过这种角色设定来挖掘模型的潜在能力. 最后, prompt 中还运用了思维链方法, 将故障复现测试用例生成这一复杂任务拆分为 4 个步骤: 理解问题的主要内容、分析造成问题的原因、思考预期的正常表现及其测试方式, 以及最终生成故障复现测试用例. 通过这种方式, 大语言模型能够逐步完成复杂的任务, 提升测试用例生成的质量. 总之, 为解决故障复现测试用例生成这一复杂任务, 本文在 prompt 中采用了多种提示工程策略, 期望通过一系列的优化, 能够让大语言模型更好地理解用户需求、生成准确且高质量的测试用例, 从而提升故障复现软件测试的效率和效果.

prompt 2. 生成测试给定问题报告的故障复现测试用例.

```
# {issue_title}
## Issue Description
{issue_description}
## Bug Context
{bug_context}
## Imports List
{imports_list}
## Example Tests
{example_tests}
## Introduction of a Test Case
A test function can be thought of as containing three parts:
The first part is the preconditions that the test case must meet. For a certain test case, you must first build the preconditions it needs to use. For example, you must create an object of a certain class and set some properties on it. Otherwise, subsequent tests will not be able to proceed or the expected results will be obtained.
The second part is the steps required to execute the current test case, which usually includes necessary assignments, a series of API calls, etc. The lines of code that exist in steps that can trigger bugs are called critical lines of code.
The third part is the expected execution result of the test case, usually corresponding to the assert statement
## Generate one test.
You are an expert in software testing, you are really good at writing test cases and test functions given a requirement.
Now I will offer you an issue from GitHub as well as the code context the issue relates to.
```

Please summarize and understand the main content of the issue based on the issue title and description, analyze the reasons for the issue, the expected normal performance of the issue and how to test it, and finally write one test case to test whether the bug has been fixed.

Tips

1. You should generate test case based on the given issue_title, issue_description and context.
2. The given ##Example Tests are test cases that may be related to the issue. You can refer to it.
3. When necessary, you can import some libraries or APIs from ## Imports List.
4. The generated test case must be a class.
5. Please use the special code format for code snippet.
6. If you call some APIs, please pay attention to the data type of the parameters.

本文的 prompt 2 模板需要填充 5 个字段, 分别是 issue_title、issue_description、bug_context、imports_list 和 example_tests. 这些字段对应于问题报告的标题、描述、报错根函数、import 语句以及测试用例样本. 其中, 问题报告标题和描述来自 SWE-bench Lite 数据集中每个问题报告的基本信息, 而报错根函数、import 语句和测试用例样本则是基于给定问题报告和代码仓库检索出的信息. 在所有字段填充完成后, 本文利用 OpenAI 提供的 API 接口, 调用 GPT-4 模型生成 5 个格式化为类的测试用例, 以实现针对特定问题报告的故障复现测试用例自动生成.

图 8 展示了大语言模型生成的故障复现测试用例, 针对的是上文提到的 Django 仓库中的问题报告, 标题为 “URLField 报错内容有误”. 图中展示的两个测试用例均由大语言模型直接生成, 并成功复现问题报告并验证问题报告是否得到解决. 生成的测试用例 1 首先定义了一个 URLField 类, 然后设计了一个非法 URL 字符串“////]@N.AN”, 并将其传入 clean 函数, 期望模型输出 ValidationError. 这一测试用例有效复现了问题报告中遇到非法 URL 的场景, 并通过断言方式验证问题报告是否正确解决, 构成了一个正确的故障复现测试用例. 生成的测试用例 2 与测试用例 1 的整体结构相似, 但额外定义了一个 expected_error_message 变量, 内容为 “Enter a valid URL.”, 进一步优化了错误输出格式. 这不仅要求测试结果能返回 ValidationError, 还添加了错误原因, 即输入了非法 URL. 这两个测试用例成功实现了面向 GitHub 问题报告的自动化故障复现测试用例生成任务, 展示了大语言模型在这一领域的有效性.

Query: 见附件1

生成的测试用例1:

```
from django.core.exceptions import ValidationError
from django.forms import URLField
from django.test import SimpleTestCase

class URLFieldTest(SimpleTestCase):
    def test_urlfield_clean_invalid_url(self):
        f = URLField()
        invalid_url = '////]@N.AN'
        with self.assertRaises(ValidationError):
            f.clean(invalid_url)
```

生成的测试用例2:

```
from django.core.exceptions import ValidationError
from django.forms import URLField
from django.test import SimpleTestCase

class URLFieldTest(SimpleTestCase):
    def test_urlfield_clean_invalid_url(self):
        f = URLField()
        invalid_url = '////]@N.AN'
        expected_error_message = 'Enter a valid URL.'
        with self.assertRaisesMessage(ValidationError, expected_error_message):
            f.clean(invalid_url)
```

图 8　大语言模型生成的故障复现测试用例

此外, 本文进一步分析发现, 图 8 中大语言模型生成的两个测试用例与图 7 中展示的第 1 个测试用例样本高度相似. 这表明, 在 prompt 中添加与问题报告相关的有效测试用例样本可以为大语言模型提供有价值的参考信息, 从而提升其生成效果. 这一发现也充分证明了本文方法的合理性与有效性.

4 实验结果与分析

为了验证本文提出的基于大语言模型的故障复现测试用例生成方法的有效性, 本文设计了对比实验和消融实验. 对比实验主要通过将本文方法与 Kang 等人[6]提出的 Libro 方法在相同的数据集上进行比较, 以验证本文方法的有效性. 同时, 消融实验则逐步去除本文检索增强生成的 3 部分内容 (报错根函数、import 语句和测试用例样本), 以观察其对生成效果的影响, 从而进一步明确各个内容的作用. 此外, 本文还采用数据分析与统计方法来验证检索内容的准确性. 基于此, 本文将围绕以下 3 个问题展开实验验证.

RQ3: 本文提出的方法是否优于现有方法?

RQ4: 本文方法中检索增强生成的 3 部分内容各自的作用如何?

RQ5: 本文检索增强生成的内容准确性如何?

4.1 实验设置

4.1.1 环境配置

由于本文所解决的任务是面向 GitHub 问题报告的故障复现测试用例生成, 每一个问题报告都属于不同的代码仓库, 因此本文为每个问题报告搭建了一个虚拟的 conda 环境, 以便在该环境中执行生成的故障复现测试用例. 如表 1 所示, SWE-bench Lite 数据集中每个问题报告都有对应的 repo 和 version, 其中 repo 表示该问题报告所在的代码仓库, 而 version 则指用于运行和评估该问题报告时的代码仓库安装版本. 根据每个问题报告的 repo 和 version, 本文为其创建了满足特定配置要求的 conda 环境. 图 9 展示了 Django 仓库和 Flask 仓库的部分版本的环境配置要求. 通过选择相应的 version 版本, 本文能够确定 conda 环境中的各项配置和安 装包内容. 其中, Python 字段表示安装的 Python 版本, packages 字段表示所需的依赖文件, install 字段表示在 conda 环境中安装代码仓库的指令, pip_packages 字段表示对 Python 包的版本要求. 基于这些信息, 本文可以为每个问题报告创建可运行的 conda 虚拟环境, 并在其中执行测试用例.

```
MAP_VERSION_TO_INSTALL_DJANGO = {
    k: {
        "python": "3.5",
        "packages": "requirements.txt",
        "install": "python setup.py install",
    }
    for k in ["1.7", "1.8", "1.9", "1.10", "1.11", "2.0", "2.1", "2.2"]
}
MAP_VERSION_TO_INSTALL_DJANGO.update(
    {
        k: {
            "python": "3.9",
            "packages": "requirements.txt",
            "install": "python -m pip install -e .",
        }
        for k in ["4.1", "4.2"]
    }
)
```

(a) Django仓库

```
MAP_VERSION_TO_INSTALL_FLASK.update(
    {
        k: {
            "python": "3.11",
            "packages": "requirements.txt",
            "install": "pip install -e .",
            "pip_packages": [
                "click==8.1.3",
                "itsdangerous==2.1.2",
                "Jinja2==3.1.2",
                "MarkupSafe==2.1.1",
                "Werkzeug==2.3.7",
            ],
        }
        for k in ["2.2", "2.3"]
    }
)
```

(b) Flask仓库

图 9 代码仓库 conda 环境配置要求示意图

4.1.2 模型和数据集

本文使用的大语言模型主要包括 OpenAI 提供的 GPT-4 模型和 GPT-3.5-Turbo 模型. 在测试文件选择阶段, 由于输入中包含每个代码仓库中所有测试文件的截取路径, 这些路径通常较长, 考虑到成本因素, 本文在此阶段使用了 GPT-3.5-Turbo 模型. 相对而言, 在故障复现测试用例生成阶段, 为了提高生成效果, 本文选择使用 GPT-4 模型. 整个模型运行环境为 x86-64 位的 Linux 操作系统. 这一组合的选择旨在充分利用不同模型的优势, 以实现最佳的故障复现测试用例生成效果.

在数据集方面, 考虑到故障复现测试用例生成任务中每个数据实例的运行都需要消耗大量时间和计算资源,

本文选择了 Jimenez 等人[8]开源的 SWE-bench Lite 数据集. 该数据集是 SWE-bench 数据集的子集, 共包含 300 条典型的 GitHub 问题报告数据, 这些问题报告来自 12 个热门的 Python 代码仓库, 具体的数据分布如图 1 所示. 每条数据除了包含问题报告的基本信息, 如所在仓库、问题报告标题和问题报告描述外, 还包括与解决该问题报告相关的重要信息, 如 patch、test_patch、FAIL_TO_PASS 和 PASS_TO_PASS 等. 这一数据集为本文的研究提供了宝贵的基础, 帮助评估本文方法生成的测试用例有效性.

4.1.3 对比方法

为验证本文提出的基于大语言模型的故障复现测试用例生成方法的效果, 本文将其与 Kang 等人[6]提出的 Libro 方法在 SWE-bench Lite 数据集上进行对比. 尽管一些基于智能代理 (agent) 的研究, 如 SWE-agent、OpenDevin、CodeR 等, 也在解决 GitHub 问题报告过程中关注了故障复现问题, 但由于这些方法本质上与 Libro 方法相似, 它们同样依赖大语言模型和 prompt 技术来生成故障复现代码, 并且方法较为简化, 因此本文以 Libro 作为该类工作的代表技术, 在实验中将 Libro 作为主要对比技术.

Libro 方法同样旨在解决故障复现测试用例生成任务, 但其是在 Defects4J 数据集上进行验证, 该数据集从 17 个 Java 项目中人工收集了真实的代码问题, 并且每个问题通常包含错误栈信息. 因此, 相较于 SWE-bench Lite 数据集, Defects4J 的数据集相对较为简单. Libro 方法利用大语言模型和提示工程技术生成测试用例, 但在 prompt 构建中使用 one-shot 技术, 提供了固定的问题报告-测试用例对, 而未提供与给定问题报告相关的上下文信息. 图 10 展示了 Libro 方法论文中的 prompt 模板示意图, 该 prompt 中仅包含问题报告的标题和描述、复现问题报告的指令以及待补全的测试函数. 由于 Libro 方法并未开源, 本文根据其 prompt 模板, 并结合对论文的理解, 对其测试用例生成部分进行了复现, 并将其应用于 SWE-bench Lite 数据集中, 以便与本文方法进行比较.

```
# NaN in "equals" methods
## Description
In "MathUtils", some "equals" methods will return true if both argument
are NaN.
Unless I'm mistaken, this contradicts the IEEE standard.
If nobody objects, I'm going to make the changes.

## Reproduction
>Provide a self-contained example that reproduces this issue.

public void test
```

图 10 Libro 方法的 prompt 模板示意图

4.2 测试用例生成效果对比 (RQ3)

本文使用成功生成故障复现测试用例的问题报告比例这一指标来评估方法的有效性. 具体而言, 本文将 Libro 方法与所提出的方法应用于 SWE-bench Lite 数据集中的 300 条 GitHub 问题报告, 每个问题报告生成 5 个故障复现测试用例. 当一个测试用例在解决给定问题报告的代码补丁应用前未通过, 而在应用后通过, 则该测试用例被判定为故障复现测试用例. 对于每个方法生成的 5 个测试用例, 只要其中有任意一个为故障复现测试用例, 便认为该方法成功为该问题报告生成了故障复现测试用例. 最终, 通过计算成功生成故障复现测试用例的问题报告在 300 个问题报告中的比例, 来评估两种方法的效果.

对比实验结果如表 3 所示. 针对 SWE-bench Lite 数据集中的 300 个 GitHub 问题报告, Libro 方法仅成功为 20 个问题报告生成故障复现测试用例, 占比 6.57%; 而本文提出的方法成功为 67 个问题报告生成故障复现测试用例, 占比提升至 22.33%. 此外, 从各个代码仓库的结果来看, Libro 仅在 3 个代码仓库的问题报告上成功生成故障复现测试用例, 而本文方法在 9 个代码仓库的问题报告上成功生成故障复现测试用例. 在 Libro 方法结果为 0 的 9 个代码仓库中, 本文方法有 6 个仓库的结果不为 0, 且问题报告测试用例生成的平均成功比例达 23.42%. 在 Libro 方法结果不为 0 的 3 个代码仓库中, 本文方法的结果均有提升, 平均提高了 14.23%. 这些结果表明, 基于给定问题报告检索报错根函数、import 语句和测试用例样本等内容的方法是有效的, 进一步证明了本文方法的有效性.

表 3 本文方法与 Libro 方法的故障复现测试用例生成评估指标对比

代码仓库名	问题报告数量	成功生成故障复现测试用例的问题报告数量		成功生成故障复现测试用例的问题报告比例 (%)	
		Libro	本文方法	Libro	本文方法
astropy/astropy	6	0	1	0	16.67
Django/Django	114	11	22	9.65	19.30
pallets/Flask	3	0	0	0	0
matplotlib/matplotlib	23	8	11	34.78	47.83
pylint-dev/pylint	6	0	1	0	16.67
pytest-dev/pytest	17	0	3	0	17.65
psf/requests	6	0	2	0	33.33
scikit-learn/scikit-learn	23	0	0	0	0
mwaskom/seaborn	4	0	1	0	25.00
sphinx-doc/sphinx	16	0	0	0	0
sympy/sympy	77	0	24	0	31.17
pydata/xarray	5	1	2	20.00	40.00
总计	300	20	67	6.57	22.33

此外, 针对本文方法成功生成故障复现测试用例的问题报告数量为 0 的 3 个代码仓库, 本文对其问题报告实例进行了深入的分析, 发现本质原因是代码仓库的领域存在差异性以及问题报告较为复杂. 以下是对 3 个生成成功率为 0 的代码仓库的详细分析: (1) Flask 是一个基于 Python 的轻量级 Web 框架, 其问题报告中通常涉及框架底层机制、请求处理流程、路由解析等较为隐蔽的框架知识. 由于大语言模型在这类框架知识方面的积累有限, 往往难以准确理解问题报告的上下文及相关代码逻辑, 从而导致故障复现测试用例的生成效果不佳. (2) scikit-learn 是一个基于 Python 的机器学习库, 包含较多的机器学习算法和工具, 问题报告中往往涉及模型参数调整、算法实现细节、性能优化等难度较高的问题, 这些问题通常需要深入理解算法原理和模型行为. 由于大语言模型在处理复杂数学推理和特定算法实现方面的能力有限, 因此在生成故障复现测试用例时, 难以准确捕获这些高级问题的上下文和逻辑关系, 不能正确生成故障复现测试用例. (3) sphinx 是一个基于 Python 的文档生成工具, 通常用于生成 API 文档、技术手册、项目文档等. 其问题报告通常涉及对文件的读写操作, 且代码中往往包含与文件系统相关的真实路径信息. 由于大语言模型缺乏本地的环境上下文, 无法访问或模拟真实的文件系统环境, 因此在处理与文件读写相关的任务时, 模型难以准确理解路径、文件内容及构建流程, 因此生成的故障复现测试用例往往不够准确或无法有效执行.

综上所述, 不同领域和复杂度的代码仓库会显著影响大语言模型生成故障复现测试用例的效果. 例如, 在本文的实验中, 效果最差的代码仓库故障复现测试用例生成成功率为 0, 而最高可达 47.83%. 其中, matplotlib 仓库的效果最好, 因为其问题报告相对较为简单, 功能模块独立, 大语言模型更擅长处理这类上下文直接、逻辑清晰的问题报告, 因此生成效果较优. 然而, 对于复杂框架、算法库或涉及环境上下文的问题报告, 大语言模型仍面临较大的挑战. 这也反映出, 在仓库级别的故障复现测试用例生成任务中, 现有方法尚未完全解决模型理解复杂代码上下文的难题, 未来的研究仍需进一步探索更有效的上下文建模和生成策略, 以提升模型在不同领域和复杂任务中的适用性.

4.3 消融实验 (RQ4)

本文在使用检索增强生成策略时共检索了 3 部分内容, 包括报错根函数、import 语句和测试用例样本. 为了探究各个内容对实验结果的影响, 本文设计了消融实验. 在该实验中, 本文依次将每个检索的内容去除, 以观察不同情况下的效果. 具体而言, 本文将去除报错根函数后的方法称为“–报错根函数”, 去除 import 语句后的方法称为“–import 语句”, 去除测试用例样本后的方法称为“–测试用例样本”. 为进行消融实验, 本文在 SWE-bench Lite 数据集中随机抽取了 100 条数据进行测试.

消融实验结果如表 4 所示. 在随机抽取的 100 个 GitHub 问题报告中, 本文方法成功为 27 个问题报告生成故障复现测试用例, 占比 27%; 而去除报错根函数的方法成功生成 24 个问题报告的故障复现测试用例, 占比 24%,

较本文方法降低了 3%. 去除 import 语句的方法成功为 21 个问题报告生成故障复现测试用例, 占比 21%, 相较于本文方法降低了 6%. 最后, 去除测试用例样本的方法仅成功为 14 个问题报告生成故障复现测试用例, 占比 14%, 相较于本文方法降低了 13%. 这些结果表明, 本文检索增强生成的 3 部分内容对故障复现测试用例生成均发挥了积极作用, 且其中测试用例样本的贡献最大.

表 4 消融实验结果

代码仓库名	问题报告数量	成功生成故障复现测试用例的问题报告数量				成功生成故障复现测试用例的问题报告比例 (%)			
		–报错根函数	–import语句	–测试用例样本	本文方法	–报错根函数	–import语句	–测试用例样本	本文方法
Django/Django	37	8	8	6	11	21.62	21.62	16.22	29.73
matplotlib/matplotlib	6	3	2	2	2	50.00	33.33	33.33	33.33
pylint-dev/pylint	1	0	0	0	0	0	0	0	0
pytest-dev/pytest	8	2	0	1	1	25.00	0	12.50	12.50
psf/requests	1	0	0	0	1	0	0	0	100
scikit-learn/scikit-learn	7	0	0	0	0	0	0	0	0
mwaskom/seaborn	1	0	0	0	0	0	0	0	0
sphinx-doc/sphinx	7	0	0	0	0	0	0	0	0
sympy/sympy	29	10	10	4	11	34.48	34.48	13.79	37.93
pydata/xarray	3	1	1	1	1	33.33	33.33	33.33	33.33
总计	100	24	21	14	27	24.00	21.00	14.00	27.00

此外, 本文观察到, 在某些情况下, 去除报错根函数的方法效果优于本文提出的方法. 例如, 在 matplotlib 代码仓库中, 去除报错根函数的方法成功为 3 个 GitHub 问题报告生成了故障复现测试用例, 而本文方法仅为 2 个问题报告成功生成. 该结果表明, 基于 GitHub 问题报告中的错误栈信息抽取报错根函数可能会误导大语言模型, 从而降低其测试用例生成的效果. 这进一步说明, GitHub 问题报告中的错误栈根函数信息可能并不总是导致问题的根本原因, 反映出了 GitHub 问题报告的复杂性.

4.4 检索增强生成内容准确性分析 (RQ5)

为了进一步分析本文检索增强生成内容的准确性, 本文设计了新的评估指标, 以验证检索内容与给定问题报告之间的相关性. 考虑到报错根函数是基于错误栈中的文件路径信息直接获取的, 且表 4 中的结果显示其对测试用例生成的影响有限, 因此本文将重点分析 import 语句和测试用例样本与问题报告之间的关联性. 帮助研究者更好地理解这些检索内容在故障复现测试用例生成中的作用.

如表 1 所示, SWE-bench Lite 数据集中每个问题报告都有对应的 FAIL_TO_PASS 和 PASS_TO_PASS 测试用例信息. 具体而言, FAIL_TO_PASS 测试用例在解决问题报告的补丁应用前未能通过, 但在应用后成功通过, 这些测试用例用来复现问题报告并验证问题报告是否解决. 而 PASS_TO_PASS 测试用例则是在解决问题报告的补丁应用前后均能通过, 代表代码仓库中原本与问题报告相关的测试用例, 类似于回归测试. 因此, 本文认为, 在问题报告提交时, 与该问题报告相关的所有测试用例都包含在这两类测试用例中.

FAIL_TO_PASS 和 PASS_TO_PASS 列表中的测试用例均包含测试函数名和所在测试文件的信息, 例如“test_urlfield_clean_invalid (forms_tests.field_tests.test_urlfield.URLFieldTest)”表示测试函数名为 test_urlfield_clean_invalid, 所在测试文件为 forms_tests.field_tests.test_urlfield.py. 因此, 为评估检索到的测试文件与给定问题报告的相关性, 本文判断该测试文件是否在 FAIL_TO_PASS 或 PASS_TO_PASS 测试用例列表中. 若检索出的测试文件存在于列表中, 则认为该测试文件与问题报告相关, 进一步认为从该测试文件中抽取出的 import 语句与给定问题报告相关. 同理, 若检索出的测试用例样本中有任意一个属于 FAIL_TO_PASS 或 PASS_TO_PASS 中的测试函数, 则认为这些样本对生成给定问题报告的故障复现测试用例是有帮助的. 本文将检索到的测试文件在 FAIL_TO_PASS 或 PASS_TO_PASS 测试用例列表中的问题报告数量记为 F_N, 比例记为 F_P; 将检索到的测试用例样本有

任意一个在 FAIL_TO_PASS 或 PASS_TO_PASS 测试用例列表中的问题报告数量记为 E_N, 比例记为 E_P.

表 5 展示了本文在 SWE-bench Lite 数据集中检索测试文件和测试用例样本的准确性结果. 从整个 SWE-bench Lite 数据集来看, 在 300 个 GitHub 问题报告中, 本文为 110 个问题报告成功检索出了相关的测试文件, 占比 36.67%; 为 105 个问题报告检索出了和问题报告相关的测试用例样本, 占比 35%. 从 12 个代码仓库数据来看, 在 8 个代码仓库中, 成功检索出相关的测试文件的问题报告占比达到了 50% 以上; 在 7 个代码仓库中, 成功检索出相关的测试用例样本的问题报告占比同样达到 50% 以上. 这些结果反映出本文检索方法的有效性, 能够为给定问题报告提供相关的代码上下文信息, 从而提升大语言模型对问题报告的理解能力, 进而增强故障复现测试用例的生成效果.

表 5 检索到的测试文件和测试用例样本准确性结果

代码仓库名	问题报告数量	F_N	F_P (%)	E_N	E_P (%)
astropy/astropy	6	3	50.00	3	50.00
Django/Django	114	34	29.82	33	28.95
pallets/Flask	3	2	66.67	1	33.33
matplotlib/matplotlib	23	13	56.52	13	56.52
pylint-dev/pylint	6	3	50.00	3	50.00
pytest-dev/pytest	17	7	41.18	6	35.29
psf/requests	6	6	100	6	100
scikit-learn/scikit-learn	23	15	65.22	15	65.22
mwaskom/seaborn	4	3	75.00	3	75.00
sphinx-doc/sphinx	16	3	18.75	2	12.50
sympy/sympy	77	17	22.08	16	20.78
pydata/xarray	5	4	80.00	4	80.00
总计	300	110	36.67	105	35.00

5 有效性威胁

本文提出的基于大语言模型和检索增强生成的故障复现测试用例生成方法, 旨在自动化生成 GitHub 问题报告的故障复现测试用例. 该方法通过检索与给定问题报告相关的多元信息, 并结合提示工程技术构建 prompt, 以引导大语言模型生成故障复现测试用例. 经过分析, 本文的工作可能面临以下 3 个有效性威胁.

首先, 本文使用 SWE-bench Lite 数据集来验证方法的有效性, 该数据集中的 300 个 GitHub 问题报告均来自一些比较热门的 Python 仓库. 在使用大语言模型为这些问题报告生成测试用例时, 存在数据泄露的风险, 这可能会影响方法的泛化能力. 为减轻这一威胁, 未来本文的研究可以考虑使用更广泛的代码仓库和不同来源的数据集, 以增强本文方法在多样化场景下的适应性.

其次, SWE-bench Lite 数据集是 SWE-bench 数据集的子集, 其筛选标准包括选择代码补丁只涉及一个文件的问题报告. 然而, 实际的 GitHub 问题报告在生成代码补丁时可能涉及多个文件. 这一筛选标准可能在一定程度上降低了数据集中问题报告的难度, 从而影响了本文方法在真实环境中的表现. 为此, 本文未来将尝试使用更真实的 GitHub 问题报告数据, 以提高本文方法的挑战性和实用性.

最后, 本文方法依赖于大语言模型生成测试用例, 但由于大语言模型的黑盒特性, 生成的故障复现测试用例可解释性较低. 此外, 模型在不同的问答过程中可能生成不完全相同的测试用例, 这降低了研究的可复现性. 为降低该有效性威胁, 本文在构建生成测试用例的 prompt 后, 让大语言模型生成了 5 次测试用例, 并记录多个生成结果, 以提供更全面的分析.

6 总结与展望

本文通过实证研究发现, 当前 GitHub 问题报告中普遍存在故障复现测试用例不足的问题, 导致开发者在提交

解决问题报告的代码补丁时, 还需额外提交用于复现问题报告并验证问题报告是否解决的测试用例补丁, 从而增加了开发者的工作负担. 为了解决这一问题, 本文提出了一种结合大语言模型和检索增强生成的故障复现测试用例生成方法. 该方法首先通过检索与问题报告相关的多元代码上下文信息, 包括报错根函数、import 语句和测试用例样本, 随后构建精确的 prompt, 以引导模型生成有效的故障复现测试用例. 在与现有的 Libro 方法进行对比实验时, 本文的方法显著优于 Libro, 成功生成故障复现测试用例的问题报告比例从 6.57% 提升至 22.33%. 此外, 消融实验进一步表明, 本文检索增强生成的 3 部分内容对故障复现测试用例生成均发挥了积极作用, 其中测试用例样本的贡献最大. 尽管本文方法在故障复现测试用例生成任务中展现了有效性, 但仍有改进空间, 未来的展望如下.

(1) 提升方法的有效性. 在使用检索增强生成策略时, 本文检索了报错根函数、import 语句和测试用例样本这 3 种信息, 未来可以挖掘更多有价值的信息. 当面对项目中测试函数样本不足的情况时, 可以检索其他与问题报告相关的上下文信息, 包括代码注释、项目历史提交记录、开发者评论以及项目文档等, 为大语言模型提供更丰富的背景信息和开发者意图, 从而提升故障复现测试用例的生成质量. 同时, 可以优化检索策略, 以提高检索内容的准确性, 从而更好地辅助模型理解问题报告, 提升故障复现测试用例生成效果.

(2) 提升方法的通用性. 本文方法目前应用于 SWE-bench Lite 数据集, 该数据集仅包含 300 个 Python 代码仓库的问题报告. 未来, 可以考虑将本文方法扩展到其他编程语言的 GitHub 问题报告, 例如 Java、C、C++等. 此外, 本文的方法也可考虑应用于本地代码仓库中, 用户只需提交问题报告描述和代码仓库路径, 便可自动生成故障复现测试用例.

(3) 提升方法的可解释性. 本文方法使用大语言模型生成故障复现测试用例时, 缺乏对测试用例的解释. 未来可以考虑在生成故障复现测试用例的同时, 让模型解释测试用例的信息来源及其与问题报告的关联. 此外, 还可以考虑建立用户反馈机制, 允许用户对生成的测试用例进行评价与反馈, 增强用户对测试用例的理解和信任.

References

[1] GitHub. 2024. https://github.com/

[2] Jimenez CE, Yang J, Geng JY. SWE-bench Lite. 2024. https://www.swebench.com/lite.html

[3] Soltani M, Derakhshanfar P, Panichella A, Devroey X, Zaidman A, van Deursen A. Single-objective versus multi-objectivized optimization for evolutionary crash reproduction. In: Proc. of the 10th Int'l Symp. Search Based Software Engineering. Montpellier: Springer, 2018. 325–340. [doi: 10.1007/978-3-319-99241-9_18]

[4] Nayrolles M, Hamou-Lhadj A, Tahar S, Larsson A. JCHARMING: A bug reproduction approach using crash traces and directed model checking. In: Proc. of the 22nd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering. Montreal: IEEE, 2015. 101–110. [doi: 10.1109/SANER.2015.7081820]

[5] Chen N, Kim S. STAR: Stack trace based automatic crash reproduction via symbolic execution. IEEE Trans. on Software Engineering, 2015, 41(2): 198–220. [doi: 10.1109/TSE.2014.2363469]

[6] Kang S, Yoon J, Yoo S. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2312–2323. [doi: 10.1109/ICSE48619.2023.00194]

[7] Brown TB, Mann B, Ryder N, *et al*. Language models are few-shot learners. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 159.

[8] Jimenez CE, Yang J, Wettig A, Yao SY, Pei KX, Press O, Narasimhan KR. SWE-bench: Can language models resolve real-world GitHub issues? In: Proc. of the 12th Int'l Conf. on Learning Representations. Vienna: OpenReview.net, 2024.

[9] Rozière B, Gehring J, Gloeckle F, *et al*. Code LLaMA: Open foundation models for code. arXiv:2308.12950, 2024.

[10] Yang J, Jimenez CE, Wettig A, Lieret K, Yao SY, Narasimhan K, Press O. SWE-agent: Agent-computer interfaces enable automated software engineering. In: Proc. of the 38th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2024. 1601.

[11] Zhang YT, Ruan HF, Fan ZY, Roychoudhury A. AutoCodeRover: Autonomous program improvement. In: Proc. of the 33rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Vienna: ACM, 2024. 1592–1604. [doi: 10.1145/3650212.3680384]

[12] Xia CS, Deng YL, Dunn S, Zhang LM. Agentless: Demystifying LLM-based software engineering agents. arXiv:2407.01489, 2024.

[13] Fraser G, Arcuri A. EvoSuite: Automatic test suite generation for object-oriented software. In: Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering. Szeged: ACM, 2011. 416–419. [doi: 10.1145/2025113.2025179]

[14] Baars A, Harman M, Hassoun Y, Lakhotia K, McMinn P, Tonella P, Vos T. Symbolic search-based testing. In: Proc. of the 26th IEEE/ACM Int'l Conf. on Automated Software Engineering. Lawrence: IEEE, 2011. 53–62. [doi: 10.1109/ASE.2011.6100119]

[15] Blasi A, Gorla A, Ernst MD, Pezzè M. Call me maybe: Using NLP to automatically generate unit test cases respecting temporal constraints. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: IEEE, 2022. 19. [doi: 10.1145/3551349.3556961]

[16] DeMilli RA, Offutt AJ. Constraint-based automatic test data generation. IEEE Trans. on Software Engineering, 1991, 17(9): 900–910. [doi: 10.1109/32.92910]

[17] Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. In: Proc. of the 29th Int'l Conf. on Software Engineering. Minneapolis: IEEE, 2007. 75–84. [doi: 10.1109/ICSE.2007.37]

[18] Dinella E, Ryan G, Mytkowicz T, Lahiri SK. TOGA: A neural method for test oracle generation. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 2130–2141. [doi: 10.1145/3510003.3510141]

[19] Schäfer M, Nadi S, Eghbali A, Tip F. An empirical evaluation of using large language models for automated unit test generation. IEEE Trans. on Software Engineering, 2024, 50(1): 85–105. [doi: 10.1109/TSE.2023.3334955]

[20] Yuan ZQ, Liu MW, Ding SJ, Wang KX, Chen YX, Peng X, Lou YL. Evaluating and improving ChatGPT for unit test generation. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 76. [doi: 10.1145/3660783]

[21] OpenDevin: Code less, make more. 2024. https://github.com/OpenDevin/OpenDevin/

[22] Chen D, Lin SX, Zeng MH, Zan DG, Wang JG, Cheshkov A, Sun J, Yu H, Dong GL, Aliev A, Wang J, Cheng X, Liang GT, Ma YC, Bian P, Xie T, Wang QX. CodeR: Issue resolving with multi-agent and task graphs. arXiv:2406.01304, 2024.

[23] OpenAI. ChatGPT. 2024. https://openai.com/index/chatgpt/

[24] Li R, Allal LB, Zi YT, *et al*. StarCoder: May the source be with you! arXiv:2305.06161, 2023.

[25] Luo ZY, Xu C, Zhao P, Sun QF, Geng XB, Hu WX, Tao CY, Ma J, Lin QW, Jiang DX. WizardCoder: Empowering code large language models with evol-instruct. In: Proc. of the 12th Int'l Conf. on Learning Representations. Vienna: OpenReview.net, 2024.

[26] Fried D, Aghajanyan A, Lin J, Wang SD, Wallace E, Shi F, Zhong RQ, Yih S, Zettlemoyer L, Lewis M. InCoder: A generative model for code infilling and synthesis. In: Proc. of the 11th Int'l Conf. on Learning Representations. Kigali: OpenReview.net, 2023.

[27] Jin M, Shahriar S, Tufano M, Shi X, Lu S, Sundaresan N, Svyatkovskiy A. InferFix: End-to-end program repair with LLMs. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 1646–1656. [doi: 10.1145/3611643.3613892]

[28] Xia CS, Zhang LM. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In: Proc. of the 33rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Vienna: ACM, 2024. 819–831. [doi: 10.1145/3650212.3680323]

[29] Lewis P, Perez E, Piktus A, Petroni F, Karpukhin V, Goyal N, Küttler H, Lewis M, Yih WT, Rocktäschel T, Riedel S, Kiela D. Retrieval-augmented generation for knowledge-intensive NLP tasks. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 793.

[30] Zhou SY, Alon U, Xu FF, Jiang ZB, Neubig G. DocPrompting: Generating code by retrieving the docs. In: Proc. of the 11th Int'l Conf. on Learning Representations. Kigali: OpenReview.net, 2023.

[31] Lu S, Duan N, Han H, Guo DY, Hwang SW, Svyatkovskiy A. ReACC: A retrieval-augmented code completion framework. In: Proc. of the 60th Annual Meeting of the Association for Computational Linguistics. Dublin: ACL, 2022. 6227–6240. [doi: 10.18653/v1/2022.acl-long.431]

[32] Zhang FJ, Chen B, Zhang Y, Keung J, Liu J, Zan DG, Mao Y, Lou JG, Chen WZ. RepoCoder: Repository-level code completion through iterative retrieval and generation. In: Proc. of the 2023 Conf. on Empirical Methods in Natural Language Processing. Singapore: ACL, 2023. 2471–2484. [doi: 10.18653/v1/2023.emnlp-main.151]

[33] Luan SF, Yang D, Barnaby C, Sen K, Chandra S. Aroma: Code recommendation via structural code search. Proc. of the ACM on Programming Languages, 2019, 3(OOPSLA): 152. [doi: 10.1145/3360578]

[34] Wei J, Wang XZ, Schuurmans D, Bosma M, Ichter B, Xia F, Chi EH, Le QV, Zhou D. Chain-of-thought prompting elicits reasoning in large language models. In: Proc. of the 36th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2022. 1800.

[35] White J, Hays S, Fu QC, Spencer-Smith J, Schmidt DC. ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In: Nguyen-Duc A, Abrahamsson P, Khomh F, eds. Generative AI for Effective Software Development. Cham: Springer, 2024. 71–108. [doi: 10.1007/978-3-031-55642-5_4]

[36] Xu BF, Yang A, Lin JY, Wang Q, Zhou C, Zhang YD, Mao ZD. ExpertPrompting: Instructing large language models to be distinguished experts. arXiv:2305.14688, 2025.

[37] Zhao ZH, Wallace E, Feng S, Klein D, Singh S. Calibrate before use: Improving few-shot performance of language models. In: Proc. of the 38th Int'l Conf. on Machine Learning. PMLR, 2021. 12697–12706.
[38] Django project. Django GitHub repository. 2024. https://github.com/django/django
[39] tree-sitter. 2024. https://tree-sitter.github.io/tree-sitter/

作者简介

汪莹, 硕士生, 主要研究领域为智能化软件开发.
字千成, 硕士生, 主要研究领域为智能化软件开发.
彭鑫, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能化软件开发, 云原生与智能化运维, 泛在计算软件系统.
娄一翎, 博士, 副研究员, 主要研究领域为智能化软件工程, 软件测试与分析.