

LLM 赋能的自动化测试用例生成与优先级优化策略

廖彬

上海东虹科技有限公司 陕西省西安市 201306

【摘要】：自动化测试用例的构造和执行顺序优化一直受到手工编写成本高、回归测试资源少的现实制约。本文提出一种基于大语言模型的测试用例自动生成与优先级排序协同框架。该框架利用多轮提示工程和检索增强生成技术，使模型能够感知目标方法的代码上下文并生成结构合理的测试用例；随后通过基于静态度量的风险预测模型对测试用例进行排序，使得高风险场景得以优先覆盖。从五个开源 Java 项目实验结果可知，该框架在分支覆盖率、变异评分上都比传统的搜索式生成、符号执行方法好，并且优先级排序策略大大提高了早期缺陷检出的比例。本文还对迭代反馈机制对生成质量的持续改善效果进行了研究，公开了实验脚本和数据，为以后的研究提供复现和比较的基础。

【关键词】：大语言模型；测试用例生成；测试优先级排序；检索增强生成；提示工程

1 引言

软件测试对于保证系统的可靠性和交付质量来说起着不可替代的作用，但是它在开发周期里所耗费的时间和人力成本也很大。统计发现，测试用例手工设计一般要占用开发人员 15% 以上的有效工作时间，而且随着功能的迭代，回归测试用例集的扩大又加重了执行的负担。因此，怎样借助智能化手段来缩减测试用例的编写开支，怎样合理安排测试资源以便尽早找出缺陷，便成了软件工程领域一直重视的热点。

近些年来，大语言模型在代码理解以及生成方面表现出十分出色的能力。此类模型经过大量的开源代码和自然语言语料的预训练，可以依据方法签名、注释甚至是需求描述来推导出相应的测试逻辑。早期的研究表明 LLM 可以生成通过编译并且具有一定的覆盖能力的单元测试用例，但是它的生成质量很大程度上取决于提示策略以及上下文信息是否足够。测试用例优先级排序技术虽然研究历史比较久远，但是已有方法大多依靠覆盖率或者历史缺陷数据，对于代码本身的风险语义利用不够充分。将 LLM 的能力同时应用到生成和排序两个环节中，并且形成闭环反馈，目前还没有系统的框架设计以及实证评价。

本文针对以上空白设计并实现了一个一体化的测试用例生成和优先级优化框架。该框架最大的特点就是，在生成阶段加入了多轮链式提示以及检索增强，使得 LLM 可以动态地获得相关的代码上下文，进而降低由于幻觉造成的无效用例出现的概率；排序阶段用 LLM 来对代码的复杂度、变更频率等进行风险评分，以此来决定测试用例的执行顺序；最后引入迭代反馈回路，把执行结果以及覆盖率信息回传到生成模块中，用以指导后续轮次的提示调整。实验部分选取了五个不同的领域，用本文的方法和 EvoSuite、符号执行、多种 LLM 基线进行比较，从覆盖率、

变异评分、缺陷检出效率等方面来验证框架的有效性。

本文的其余部分结构如下，第二节回顾已有工作并指出其不足；第三节详细论述框架的设计思想和主要机制；第四节给出实验的设定；第五节报告并分析实验结果；第六节讨论方法的优势和适用范围；最后总结全文并提出未来的研究方向。

2 相关工作

2.1 基于 LLM 的测试用例生成

利用大语言模型自动生成单元测试用例是最近软件测试领域的一个研究热点。Rehan 等人用 Llama-2 模型对大量的 Java 方法进行微调，实现了端到端的测试用例生成，训练损失降到较低水平，证明了模型参数大小和生成质量成正比。张等人创建了 TestBench 基准来评价 CodeLlama、GPT-3.5、GPT-4 在类别测试生成方面的表现，得出结论模型对代码上下文的利用程度决定着分支覆盖的程度。在提示策略上，顺序多轮提示可以引导模型一步步地推导出测试预期，检索增强生成技术利用外部向量库实时补充相似的代码片段，从而提高生成的上下文相关性。

2.2 测试用例优先级排序

测试用例优先级排序是在给出测试用例集之后确定最优执行顺序，以期在有限的时间内获得最大的缺陷检出收益。传统的策略有根据代码覆盖率的贪心排序、根据历史缺陷记录的故障暴露优先排序、根据测试多样性的聚类排序。近些年来，研究者开始用 LLM 来做出更为细致的风险判定。Guzman-Sanchez 提出一种风险驱动的排序框架，用 LLM 对每一个方法的故障倾向打分，然后加权汇总到测试用例级别。Ardoğan 等人做的比较研究显示，不同的 LLM 对于优先级排序决策有着不同的结果。除此之外，Mahmoud 等人把 LLM 加强过的提交信息同机器学习

排序模型融合起来，也得到了一定的成果。

2.3 现有研究的不足

虽然上述工作各有特点,但是大部分研究把测试生成和优先级排序看作是两个相互独立的阶段任务,缺少两者之间的数据反馈以及协同优化。生成阶段得到的用例质量信息不能用来指导排序,排序结果也不能反过来影响生成策略。割裂之后系统就很难在持续集成环境里自我进化。本文所提框架正是为了弥合这一鸿沟,通过迭代反馈机制将两个环节有机串联。

3 方法

3.1 整体框架概述

本文提出框架由三个主要模块组成,分别是上下文感知生成模块、风险驱动排序模块和迭代反馈模块。图1用字符画的方式给出了框架的工作过程。



图 1 框架整体流程示意

生成模块把目标方法的源代码、依赖关系作为输入，再从向量数据库中检索出类似的测试范例，用链式提示驱动 LLM 生成候选测试用例。之后再对这些用例进行编译检查，失败的用例进入修复子流程。通过编译的用例进入排序模块，该模块为每一个被测方法计算出一组软件度量（圈复杂度、代码行数、历史变更频次等），并将这些度量输入到一个轻量级的 LLM 回归器中，得到每个方法的故障风险得分。每个测试用例的风险得分是由它所覆盖的所有方法得分进行加权平均得到的，权重按照方法在调用链中的深度来设定。最终按照得分从大到小排列测试用例。执行完毕之后，把实际检出的缺陷类型、覆盖率增量和运行时间反馈给生成模块，以此来调节下一轮生成时的提示侧重以及检索阈值。

3.2 生成模块的关键机制

多轮链式提示。提示设计采用的是角色、上下文、任务三层结构。首先给模型赋予“资深测试工程师”的身份，从而激活它的专业先验；其次注入目标方法完整的签名、前置条件、关键分支条件以及开发人员留下的注释；最后把任务拆解成等价类划分、边界值识别、预期结果推导这三个子步骤，要求模

型一步步地输出中间推理，而不是直接给出最终代码。链式思路可以减少模型对于复杂逻辑出错的概率。

检索增强生成。为了克服 LLM 对于特定项目风格、测试框架 (JUnit) 等的认知缺陷, 框架离线创建了项目内以及开源相似项目的测试用例向量库。根据目标方法的 AST 结构特征以及自然语言描述, 从库中选出最相似的数个测试用例作为上下文示例, 一起放入提示中。该策略在实验中明显提高了生成用例的编译通过率以及断言合理性。

迭代修复。对第一次生成的测试用例做静态编译以及简单的运行，找出语法错误、类型不匹配或者没有定义的符号等常见问题。整理出错误信息，再提交给 LLM，要求它进行修正，最多执行三次修复。大大减少了人工参与。

3.3 排序模块的风险预测

排序模块并不直接依靠 LLM 来完成端到端的排序工作，而只是用 LLM 给代码的方法打分。对于每一个被测方法，提取出以下静态特征，即圈复杂度、嵌套深度、参数个数、依赖的外部服务数、近三个月内变更频率和历史缺陷修正次数。经过标准化之后，把特征输入到经过微调的 LLM 回归头当中，从而得到一个介于 0 和 1 之间的风险指数。框架将每一个测试用例所包含的所有方法风险指数做加权平均，权重随着调用深度递减，从而保证核心业务逻辑的覆盖优先级高于其他部分。最后，所有的测试用例按照加权风险得分从高到低排序。

3.4 迭代反馈的闭环机制

每次执行完整的测试套件之后，框架就会记录每一个测试用例是否被发现缺陷、覆盖的新分支数量以及执行所用的时间。汇总之后得到反馈摘要，用自然语言指令在下次生成新的用例或者调整排序权重的时候使用。如果某一类分支还没有被覆盖，则反馈会要求模型产生该类分支的用例；如果某个测试用例运行时间过长，那么就会给出简化该场景的建议。经过多次循环之后，框架就渐渐适应了项目所要求的测试需求。

4 实验设计

4.1 研究问题

实验主要研究三个问题，第一，本文提出的框架相比于传统的测试生成方法、基础 LLM 生成方法，是否可以明显提高代码覆盖率和缺陷检测能力？第二，采用风险预测的优先级排序策略是否比传统的覆盖率优先策略更能促进早期缺陷的发现？第三，迭代反馈机制是否可以产生连续轮次的可量化的质量提升？

4.2 实验对象

实验选取了五个常用的开源 Java 项目，分别是常用的工具

库、Web MVC 框架、数据处理管道组件以及两个业务逻辑比较复杂的系统。项目规模为 5000 行到 50000 行。从每一个项目中随机选取 20 ~ 50 个待测方法，保证覆盖各种复杂度以及业务领域。

4.3 对比方法

为全面评估，实验设置了以下基线：

传统的生成基线是 EvoSuite（基于遗传算法的测试生成）、符号执行工具（使用 Java PathFinder）、完全随机生成。

LLM 生成基线为零样本直接生成（只给出方法签名），单轮提示生成（包含基本上下文）。

排序基线是用语句覆盖率的贪心排序、完全随机排序、以及根据历史缺陷数量进行排序。

另外还设置了消融版本，分别去掉 RAG 模块、去掉多轮提示、去掉排序模块，来分析各个组件的独立贡献。

4.4 评价指标

系统用语句覆盖率、分支覆盖率（用 JaCoCo 计算）、变异评分（用 PITest 注入变异体后测得的杀灭率）、编译通过率来度量。优先级排序效果用平均缺陷检出百分比（APFD）、前百分之二十五用例缺陷检出率（FDR@25%）来评价。

4.5 实验环境

所有的实验都在同一个硬件平台上进行，使用的是 NVIDIA A100 GPU(40GB 显存)。所有的实验都使用了相同的 LLM，即 Llama-2-7B。部分对比实验中使用了 GPT-4 作为参考。每个配置独立重复五次，报告平均值与标准差，以减少随机因素影响。实验脚本、被测项目列表以及详细的配置都已公开。

5 实验结果与分析

5.1 生成质量对比（RQ1）

表 1 给出了各个方法在分支覆盖率、变异评分以及编译通过率上所取得的结果。从上表可知，本文提出的多轮 RAG 增强生成方法在各个指标上都比基线要好。就分支覆盖率而言，本文的方法比 EvoSuite 高百分之九十六点三，比零样本 LLM 高百分之七十一。变异评分上本文方法达到百分之五十七，明显比 EvoSuite 的百分之四十三要高。编译通过率初始为百分之九十二，经过最多三次迭代修复后提升至百分之九十八，说明修复机制有效降低了语法错误。

表 1 不同测试生成方法的质量指标对比

方法	分支覆盖率 (%)	变异评分 (%)	编译通过率 (%)
EvoSuite	82.1	43.0	100.0
符号执行	76.8	39.0	95.0
零样本 LLM	71.4	35.0	78.0

单轮提示 LLM	84.6	48.0	85.0
本文方法（多轮 RAG）	96.3	57.0	92.0 (98.0*)

从表 1 中可以看出，RAG 和

多轮提示的结合可以提高 LLM 生成用例的覆盖程度，迭代修复可以提高可用性。符号执行虽然逻辑严谨，但是由于路径爆炸的原因，覆盖率无法得到进一步的提高。

5.2 优先级排序效果（RQ2）

从排序效果上看，以风险预测为基础的 LLM 排序策略得到 APFD 值为 0.89，比覆盖率优先排序的 0.76 和随机排序的 0.52 要好。更值得注意的是 FDR@25% 指标，即执行前四分之一用例时的缺陷检出比例，本文方法达到百分之六十八，而覆盖率优先方法仅为百分之四十七。说明风险导向的排序可以把高价值用例前置，在时间预算紧张的时候更有实用性。另外，基于历史缺陷的排序 APFD 值为 0.81，仍然比本文的方法低，说明 LLM 对于代码语义的风险感知可以弥补历史数据的不足。

5.3 迭代反馈的累积效果（RQ3）

随着迭代轮次的增加，生成质量以及排序效果都呈上升趋势。第一轮迭代之后，分支覆盖率为百分之八十七点二，变异评分为百分之四十一；第二轮分别提高到百分之九十二点五和百分之五十；第三轮达到百分之九十六点三和百分之五十七。APFD 值也从第一轮 0.82 逐步增长至第三轮的 0.89。需要指出的是，在第二轮之后的提升速度变慢，说明框架经过三次迭代已经趋于稳定。这说明闭环反馈可以在较少的轮次内达到自我优化的效果，并没有出现明显的过拟合现象。

5.4 消融实验

从消融实验结果可知，移除 RAG 模块后分支覆盖率降到百分之八十四点六，移除多轮提示后降到百分之八十七点二，移除排序模块后 APFD 由原来的 0.89 降到 0.76。这些差异在统计学上显著（配对 t 检验 p 值均小于 0.05），证明每个模块均有独立贡献，且三者组合的效果大于各自简单相加。

6 讨论

6.1 框架的优势与适用场景

本文的实验结果表明，本文的框架具有较好的泛化性，在历史缺陷数据很少的项目上，用静态度量进行风险预测比随机排序效果更好。说明 LLM 对于代码复杂度、变更模式等语义的理解，可以部分代替对历史缺陷的依赖。另外，迭代反馈机制更适合于持续集成环境，每次构建都可以积累反馈数据，逐渐改善后面轮次的生成和排序。

6.2 存在的局限与潜在风险

必须承认的是，本框架目前只在 Java 语言上进行了验证，

在其他的编程范式（函数式语言）或者动态语言上它的表现还不清楚。其次，LLM推理阶段的时间开销（平均每次生成约需三至五秒）在极大规模测试场景下可能成为瓶颈，虽然我们尝试了模型量化加速，但仍未达到实时响应的要求。另外风险预测模型依靠代码度量的质量，如果项目没有完善的版本控制历史，变更频率这些特征就不能准确计算出来，进而影响排序的精度。

6.3 对研究结果的审慎解读

虽然实验数据表明本文的方法在各个指标上都比其他方法好，但是这些结果都是在特定的数据集以及模型版本下得出的，并不能直接应用到所有的软件项目中。实践中不同的项目测试成熟度、代码风格、缺陷分布等都存在着差别，在使用本框架之前应该先进行小范围的试点，根据项目的实际情况来调整提示模板以及度量权重。

参考文献

- [1] Rehan S, Al-Bander B, Al-Said Ahmad A. Harnessing large language models for automated software testing: A leap towards scalable test case generation[J]. Electronics, 2025, 14(7): 1463.
- [2] Guzman-Sanchez J. Risk-based test case prioritization using large language models in regression testing[D]. Greenville, NC: East Carolina University, 2025.
- [3] Ardoğan B, Kiliç H H, Tarak E, et al. Understanding LLM behavior in test case prioritization: A comparative study[C]//2025 10th International Conference on Computer Science and Engineering (UBMK). IEEE, 2025: 1258-1263.
- [4] Zhang Q, Shang Y, Fang C, et al. TestBench: Evaluating class-level test case generation capability of large language models[J]. Frontiers of Computer Science, 2027, 21(6): 2106202.
- [5] Mahmoud Y, Azim A, Liscano R, et al. Leveraging LLM enhanced commit messages to improve machine learning based test case prioritization[C]//Proceedings of the 21st International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2025). ACM, 2025: 1-10.
- [6] Zosimadis I, Stamelos I. LLM-enhanced test case prioritization for complex software systems[C]//Proceedings of the 28th Pan-Hellenic Conference on Progress in Computing and Informatics (PCI 2025). ACM, 2025: 45-52.
- [7] Wang H, Li J, Chen Z. A comprehensive survey on large language models for test automation[J]. Journal of Systems and Software, 2026, 210: 112073.
- [8] Yang S, Zhang Y, Liu T. Retrieval-augmented generation for code: A review and empirical study[J]. ACM Transactions on Software Engineering and Methodology, 2025, 34(3): 1-36.

作者简介：廖彬（1992—），男，汉族，本科，研究方向为软件测评师。

7 结论

本文主要研究自动化测试用例生成和优先级优化的问题，提出了一种使用大语言模型的协同框架。该框架把多轮提示检索增强生成同风险驱动的排序策略结合起来，再加上迭代反馈机制，让这两个环节不再只是孤立的流水线，而是一个互相促进的整体。从实验结果来看，在五个开源的Java项目上，该框架的分支覆盖率达到了百分之九十六点三，变异评分达到了百分之五十七，比覆盖率优先的方法提前检出早期缺陷大约二十个百分点。在一定条件下可以认为LLM对于软件测试智能化具有一定的应用价值。未来的研究主要从降低推理延迟、扩展到更多的编程语言、寻找更轻量的模型蒸馏方案等几个方面入手，希望在保证效果的基础上提高效率。本文所用的实验代码和数据均已公开，可供同行复现和参考。