好的，我将帮你检索和整理近五年内关于“利用大型语言模型（LLM）结合故障测试报告进行软件修复”的相关论文，重点聚焦以下内容：

- LLM 是否能根据故障报告定位错误代码段（故障定位能力）
- 故障报告参与是否能提升模型的修复效果
- 不同类型的故障报告（如是否含有堆栈、是否结构清晰）对 LLM 修复效果的影响

我将优先选取来自 arXiv、ACM、IEEE、Springer 等平台的论文，并提炼每篇的研究方法、使用的数据集和关键发现。
完成后我会整理成清晰的列表供你参考。稍后见！

利用大型语言模型结合故障报告进行软件修复的近五年研究综述

近年来，大型语言模型（LLM）在自动程序修复（APR）领域显示出潜力，特别是在结合自然语言故障报告（如问题描述、错误堆栈、测试失败信息）来定位和修复软件缺陷方面。以下精选了近五年的中英文研究成果（优先选取 arXiv、ACM、IEEE、Springer 数据库），重点关注以下三个方向：**(1)** LLM 根据自然语言故障报告进行故障定位的能力；**(2)** 故障报告信息对 LLM 自动修复 bug 效果的提升；**(3)** 不同类型故障报告（是否包含堆栈信息、是否结构化）的影响。每篇论文均提炼了研究背景、方法、数据集、实验设计、关键结论和贡献，并提供了公开代码/数据集链接（如有）。

DEVLoRe：融合多种软件信息的 LLM 故障定位与修复框架 (Feng et al., TOSEM 2024) (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair) (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)

- **背景**：现有基于 LLM 的自动程序修复通常只利用单一类型的信息（例如仅使用问题描述或错误堆栈），未充分模仿开发者调试时综合多源信息的做法 (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)。为缩小这一差距，作者提出利用多种软件工件（issue 描述、讨论、错误堆栈和调试信息）来改进 LLM 的故障定位和修复性能 (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)。

- **方法**：提出了 **DEVLoRe** 框架（Developer Localization and Repair）。流程包括两步：首先，LLM 利用问题报告的内容（issue描述及讨论）和错误堆栈信息定位出可能的**有缺陷的方法**；接着，结合代码中的动态调试信息、问题报告和堆栈，LLM 定位具体**有缺陷的行**并生成补丁 (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)。所使用的 LLM 采用了插入式补全或直接生成代码片段的方式生成修复补丁。
- **数据集**：在 Java 缺陷数据集 **Defects4J v2.0** 上评估，包括单方法缺陷和多方法缺陷两类 (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)。
- **实验设计**：对比不同信息组合对故障定位和修复的影响，分别提供：(a) 无任何故障报告信息，仅代码；(b) 仅 issue 描述；(c) 仅错误堆栈；(d) 仅调试信息；以及各种组合（issue+堆栈、堆栈+调试、issue+调试等），评估LLM定位缺陷方法和缺陷行的Top-n准确率及生成有效补丁的比例 (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair) (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)。
- **关键结论**：不同类型的软件工件各有所长，**结合**使用效果最佳。提供问题描述使LLM比不提供时**多定位19个**缺陷方法；提供错误堆栈则多定位**37个**缺陷方法，证实不同信息可互补 (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)。在 Defects4J v2.0 上，DEVLoRe 成功**定位了49.3%的单方法bug，生成56.0%的合理补丁；对于跨方法的bug，定位率为47.6%，补丁生成率14.5% (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)。端到端地，DEVLoRe 修复了单方法bug的39.7%**，多方法bug的**17.1%**，整体性能优于当前最新APR方法 (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)。
- **研究贡献**：DEVLoRe 是首个将**问题报告、错误堆栈和调试数据**三类信息融合用于 LLM 自动修复的框架，显著提高了故障定位精度和修复成功率。结果揭示：issue内容对定位bug作用大，错误堆栈对生成正确补丁作用大，不同信息组合能覆盖不同类型的bug (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)。此外，DEVLoRe 提供了从**bug源码定位到完整修复**的一站式流程。
- **开放资源**：作者提供了复现所需的源码和实验结果，已在GitHub开源 (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)。

SWE-bench：大型语言模型解决真实 GitHub Issue 的能力 (Jimenez et al., ICLR 2024) ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?) ([2310.06770] SWE-bench: Can

Language Models Resolve Real-World GitHub Issues?)

- **背景**：尽管近年LLM在代码生成任务上表现优异，但其在**真实软件工程问题**上的能力尚缺乏有效评测。作者指出真实项目中的Issue（包括bug报告和特性请求）涉及复杂的多文件修改和长上下文，超出传统代码补全任务的范围 ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。为推动更实用的LLM发展，作者构建了一个包含真实Issue的评测基准SWE-bench ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。
- **方法**：提出 **SWE-bench** 基准，用于评价LLM在真实场景下解决GitHub Issue的能力。数据集涵盖12个流行Python项目的**2,294个**已解决Issue以及对应的PR（补丁） ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。给定完整代码库和Issue描述，模型需**自动修改代码**来解决该Issue ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。作者还 fine-tune 了一个专用模型“SWE-Llama”（基于CodeLlama）来尝试解决这些Issue ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。
- **数据集**：SWE-bench 是目前规模最大的真实软件问题数据集，每个样例都包含问题的自然语言描述、相关代码变更和测试用例，覆盖**错误修复**和**功能新增**两类任务 ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。此外作者提供了一个简化版本“SWE-bench-lite”供快速实验。
- **实验设计**：评估了多种**最新商用模型**（如 GPT-4、Claude 2 等）和研究模型（如作者fine-tune的SWE-Llama）在 SWE-bench 上的表现，度量指标为Issue成功解决率 ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。实验重点分析模型对**长上下文、多模块修改**的处理能力，并比较fine-tune与零样本/少样本提示的差异。
- **关键结论**：当前最先进的模型在真实软件Issue上表现远未达到实用水平。最好模型Anthropic Claude 2仅解决了**1.96%的问题，其他模型（包括GPT-4）也只能解决极少量最简单的Issue ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。SWE-Llama虽经微调，但仍只能解决少部分简单问题。总体而言，现有LLM只能应对最简单的2%以内**的问题，凸显解决真实软件问题的巨大挑战 ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。这一结果说明需要面向复杂上下文和多步推理的显著改进，SWE-bench为此提供了评测平台。
- **研究贡献**：SWE-bench开创了**真实世界软件问题**评测的新基准，涵盖多个项目的真实故障报告及修复，具有高度挑战性和可持续性。该工作揭示了**现有LLM在复杂Bug修复上的不足**，为下一代更智能、自主的代码AI指明了努力方向 ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。作者也开放了数据、代码和排行榜，方便社区评测和改进模型 ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。

- **开放资源**：SWE-bench 数据集与评测框架已公开在项目网站 ([2310.06770] SWE-bench: Can Language Models Resolve Real-World GitHub Issues?)。

ChatRepair：通过对话式反馈提升LLM自动修复能力 (Xia & Zhang, ICSE 2024) ([2304.00385] Keep the Conversation Going: Fixing 162 out of 337 bugs for ParseError: KaTeX parse error: Expected 'EOF', got '#' at position 58: …/abs/2304.00385#:~:text=,For%20…) ([2304.00385] Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT)

- **背景**：传统LLM-based修复采用“生成-验证”范式：先让模型根据buggy code生成大量补丁，再用测试验证，但这种一次性生成忽略了测试失败提供的重要信息，且可能反复生成类似错误的补丁 ([2304.00385] Keep the Conversation Going: Fixing 162 out of 337 bugs for ParseError: KaTeX parse error: Expected 'EOF', got '#' at position 58: …/abs/2304.00385#:~:text=Languag…)。为充分利用**测试反馈**并减少冗余尝试，作者提出对话式交互的修复方法 ([2304.00385] Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT)。
- **方法**：提出 **ChatRepair**，首次将**对话式交互**引入自动程序修复。ChatRepair 使用 ChatGPT 作为对话代理，流程：首先将失败测试的信息（包括断言失败消息、堆栈等）与buggy代码一起作为初始提示，生成一个候选补丁；若补丁未通过测试，则将该**补丁及对应的测试失败信息**反馈给 ChatGPT，让其“学习”先前补丁的错误，再生成新补丁 ([2304.00385] Keep the Conversation Going: Fixing 162 out of 337 bugs for ParseError: KaTeX parse error: Expected 'EOF', got '#' at position 58: …/abs/2304.00385#:~:text=,For%20…)。如此循环迭代，避免重复犯错。对于已经通过所有测试的**可行补丁**（plausible patch），ChatRepair 进一步请求 ChatGPT 基于此生成**变体**，以找到潜在更正确或更优化的修复 ([2304.00385] Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT)。整个过程自动进行，直到找到通过测试的正确补丁或达到轮次上限。
- **数据集**：在经典缺陷数据集 **Defects4J (337 个 bug)** 上验证方法有效性 (Fixing 162 out of 337 bugs for ParseError: KaTeX parse error: Expected 'EOF', got '#' at position 66: …/abs/2304.00385#:~:text=Keep%20…) ([2304.00385] Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT)。ChatRepair将每个缺陷对应的测试失败信息都纳入初始prompt，并逐步对话生成修复。
- **实验设计**：比较ChatRepair与传统生成验证式方法在修复数量和代价上的差异，统计ChatRepair成功修复的Bug数量和调用LLM的次数/成本。同时，与以往APR工具（如基于Codex的修复）对比修复率，并分析多轮对话对修复质量的贡献。

- **关键结论**：通过将测试反馈融入对话迭代，ChatRepair 显著提升了修复效果和效率。在Defects4J上，ChatRepair 成功修复了**162个/337个**缺陷，平均每个缺陷仅花费**$0.42美金**的API调用成本**([2304.00385] Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT)。相比一次性生成，ChatRepair自动对之前错误学习，可避免重复错误并累积成功经验，从而实现更高的修复率。该方法证明了**持续对话+即时反馈**是LLM APR的有效策略。
- **研究贡献**：ChatRepair 是**第一个对话驱动的自动程序修复**方法，将人类调试中“边运行测试边修改”的过程自动化地迁移给LLM，有效利用测试用例提供的故障信息。它表明结合LLM的对话能力和即时失败反馈，可以在无需人工参与的情况下，大幅提高修复成功率，并降低补丁验证的成本([2304.00385] Keep the Conversation Going: Fixing 162 out of 337 bugs for ParseError: KaTeX parse error: Expected 'EOF', got '#' at position 58: …/abs/2304.00385#:~:text=,For%20…)([2304.00385] Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT)。此工作推动了LLM在软件调试领域的应用，并为后续研究提供了新思路。
- **开放资源**：相关代码与运行脚本在论文附录中提供，依赖OpenAI的ChatGPT API执行补丁生成（由于使用商用API，作者未公开专用仓库，但提供了实验细节）。

FuseFL：融合多源信息的可解释故障定位 (Widyasari et al., SANER 2024) ([

"Demystifying faulty code: Step-by-step reasoning for explainable fault" by Ratnadira WIDYASARI, Jia Wei ANG et al.
](https://ink.library.smu.edu.sg/sis_research/9257/#:~:text=explainable fault localization,To evaluate the)) (
"Demystifying faulty code: Step-by-step reasoning for explainable fault" by Ratnadira WIDYASARI, Jia Wei ANG et al.
)

- **背景**：自动故障定位工具通常只输出可疑代码的位置排名，但缺乏解释说明，开发者仍需花时间理解原因 (
  "Demystifying faulty code: Step-by-step reasoning for explainable fault" by Ratnadira WIDYASARI, Jia Wei ANG et al.
)。为提升定位结果的可解释性，作者探索让LLM“像人一样”对代码进行推理解释。FuseFL由此诞生，通过结合多种信息，引导LLM逐步分析故障原因并给出解释。
- **方法**：提出 **FuseFL**，使用ChatGPT辅助进行逐步推理的故障定位。FuseFL 将**三类信息**融合进Prompt： (1) **谱信息**：传统光谱法的怀疑行排名结果； (2) **测试执行结果**：失败的断言或错误输出； (3) **代码意图描述**：对相关代码片段功能的解释（人工生成，用于提示正确行为） (
  "Demystifying faulty code: Step-by-step reasoning for explainable fault" by Ratnadira WIDYASARI, Jia Wei ANG et al.

)。LLM据此进行链式思考，确定最可能的故障位置，并生成**逐步推理解释**。作者还构建了包含 324 个带有人类解释的故障样本的数据集，用于评估LLM解释质量 ("Demystifying faulty code: Step-by-step reasoning for explainable fault" by Ratnadira WIDYASARI, Jia Wei ANG et al.)。

- **数据集**：采用 **Refactory** 基准中的有缺陷代码（包含 Java 项目的真实Bug），扩充了每个bug的人类解释，以评估FuseFL生成解释的准确性 ("Demystifying faulty code: Step-by-step reasoning for explainable fault" by Ratnadira WIDYASARI, Jia Wei ANG et al.)。总计收集了324个有错误的代码文件和600条逐行的故障解释。
- **实验设计**：首先比较FuseFL的自动故障定位性能，与不含这些附加信息的LLM定位结果作对比，以Top-1命中率衡量改进幅度。其次，将FuseFL生成的解释与人类参考解释比对，并通过人工用户研究评估解释的正确性和有用性（随机抽样30个案例请开发者判断）。
- **关键结论**：通过融合谱信息、测试输出和代码描述，FuseFL将ChatGPT的故障定位Top-1成功数提高了**32.3%**，证明多源信息能显著增强LLM定位准确性 ("Demystifying faulty code: Step-by-step reasoning for explainable fault" by Ratnadira WIDYASARI, Jia Wei ANG et al.)。在人类评估中，随机30个案例中有22个FuseFL生成的解释是正确的，表明其解释质量较高 ("Demystifying faulty code: Step-by-step reasoning for explainable fault" by Ratnadira WIDYASARI, Jia Wei ANG et al.)。这说明LLM不仅能指出bug位置，还能给出有价值的**原因分析**，有助于开发者理解。
- **研究贡献**：FuseFL开创性地将**故障定位与解释生成**相结合，引入链式推理（CoT）让LLM输出带原因说明的定位结果。在技术上，它证明了将**传统FL工具输出+自然语言描述**融合进LLM提示可以提升定位性能并生成解释，为可解释APR提供了新途径。作者还发布了带有人类书写解释的故障定位数据集，为后续研究提供了评测基准。
- **开放资源**：FuseFL相关数据（代码片段、失败测试、人类解释）已随论文公开，代码实现细节在论文附录给出。

AutoCodeRover：LLM+代码搜索的自主软件改进 (Zhang et al., ISSTA 2024) ([2404.05427] AutoCodeRover: Autonomous Program Improvement) ([2404.05427] AutoCodeRover: Autonomous Program Improvement)

- **背景**：AI社区近期出现了一些让LLM充当多步骤编程助手（agent）的尝试，但很多将软件项目简单视为文件集合，缺乏软件工程视角 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。本研究聚焦软件维护中的两个核心任务：**bug修复**和**特性增加**，提出让LLM结合程

序分析技术，自动解决开发者提出的Issue，从而实现自主的程序改进 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。

- **方法**：提出 **AutoCodeRover**，一个利用LLM驱动的自治编程代理，通过**语法结构感知的代码搜索**来增强问题理解和上下文获取，从而进行代码修改 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。具体而言：给定一个真实GitHub Issue（包含bug报告或新功能请求），AutoCodeRover 首先由LLM解析Issue的自然语言描述，提取出潜在相关的**类、方法或变量名**作为关键字 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)；然后使用定制的**代码搜索API**（基于AST分析）迭代检索代码库中相关位置的上下文（如类定义、方法实现等）供LLM阅读 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。此外，如果有测试用例，AutoCodeRover 还结合**谱式故障定位**技术快速锁定可疑区块，加强上下文聚焦 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。LLM在每轮检索后更新对问题的理解，逐步定位根因，最终生成修改方案并应用于代码。整个过程中LLM作为中枢，调用搜索和调试工具协同完成任务。
- **数据集**：主要评估在 SWE-bench 的子集 **SWE-bench-lite (300个GitHub Issue)** 上，其中既包含软件缺陷也包含新特性请求，以验证AutoCodeRover对多种任务的适应性 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。另外在完整SWE-bench全数据（2294个Issue）上也报告结果用于观察大规模表现 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。
- **实验设计**：将AutoCodeRover与近期其它自动代码助手（如AI社区提出的LLM Agent或工业工具）比较。在SWE-bench-lite上计算Issue解决率，并与无需程序分析的纯LLM agent方案对比。在全SWE-bench上，与业界类似系统（如Cognition Labs的“Devin”自动编程助手）比较解决比例和速度 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。同时统计AutoCodeRover在每个Issue上的平均解决时间，与人类开发者解决相同Issue的用时比较。
- **关键结论**：AutoCodeRover显著提高了LLM解决真实Issue的效率和效果。在SWE-bench-lite (300个Issue)上，AutoCodeRover解决了**22-23%的问题，高于之前报道的AI agent方法 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)；其中67个Issue在12分钟内**即被解决，而开发者平均花费2.77天 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。在完整的2294个Issue上，AutoCodeRover解决了约**16%**，超越了业内“Devin”系统的效果，而耗时相当 ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。这表明结合代码结构搜索和LLM代理的方案，可以在更大规模真实项目上取得领先表现。
- **研究贡献**：AutoCodeRover将**软件工程知识（AST、谱定位）**与LLM智能有机结合，开创了一种解决GitHub Issue的自治Agent框架。它证明了LLM在复杂代码库下通过调用分析工具可以克服仅依赖自身上下文长度的限制，实现对多文件、多步骤问题的处理 ([2404.05427] AutoCodeRover: Autonomous Program Improvement) ([2404.05427] AutoCodeRover: Autonomous Program Improvement)。该工作拓展了LLM在自主软件维护领域的应用边界，并通过与开发者实际耗时的对比，展示了潜在的生产力提升。

- **开放资源**：AutoCodeRover 的算法实现和实验脚本已在 GitHub 开源 (AutoCodeRoverSG/auto-code-rover - GitHub)。此外，SWE-bench-lite 数据和问题解析流程也在附录提供，方便复现研究结果。

Toggle：LLM粒度化故障定位与修复 (Hossain et al., FSE 2024) (A Deep Dive into Large Language Models for Automated Bug Localization and Repair) (A Deep Dive into Large Language Models for Automated Bug Localization and Repair)

- **背景**：许多深度学习APR方法假设**缺陷位置已知**或依赖额外的故障定位工具，这割裂了定位和修复过程 (A Deep Dive into Large Language Models for Automated Bug Localization and Repair)。一些工作尝试端到端直接从代码预测修复，但在真实场景中缺乏准确的bug定位会降低修复效果 (A Deep Dive into Large Language Models for Automated Bug Localization and Repair)。为此，作者提出将LLM用于**精细粒度**的缺陷定位，然后再用于修复，分别优化两步并集成。
- **方法**：提出综合框架 **Toggle** (**T**oken-**g**ranulated Bug **Lo**calization and **Re**pair)，分三部分： (1) **Bug 定位模型**：基于LLM，在**token级别**预测bug的位置（精确到具体代码标记）；(2) **调整模型**：解决不同模型词元化不一致的问题，确保定位结果正确对应源码位置；(3) **Bug 修复模型**：使用另一LLM，根据定位的bug位置，对代码进行修改修复 (A Deep Dive into Large Language Models for Automated Bug Localization and Repair) (A Deep Dive into Large Language Models for Automated Bug Localization and Repair)。这种设计将定位和修复解耦，使每步可融入不同上下文信息和诱导偏置（inductive bias）。作者还实验了多种prompt风格，找出对修复模型最有效的提示方式，以及探究提供额外信息（如注释、怀疑行）对定位的影响 (A Deep Dive into Large Language Models for Automated Bug Localization and Repair)。
- **数据集**：使用 **CodeXGLUE Code Refinement** 基准和 **Defects4J** 等多个常用APR数据集评估Toggle性能 (A Deep Dive into Large Language Models for Automated Bug Localization and Repair)。这些数据集涵盖不同语言和类型的缺陷，验证模型的通用性。
- **实验设计**：比较Toggle与现有APR方法在各数据集的修复准确率Top-K指标上的表现，尤其关注Defects4J数据集上Top-10/50/100修复率的提升 (A Deep Dive into Large Language Models for Automated Bug Localization and Repair)。同时进行消融实验：仅使用行级定位 vs token级定位、是否使用调整模型、不同prompt内容（如有无附加代码注释）对性能的影响 (A Deep Dive into Large Language Models for Automated Bug Localization and Repair)。还测试Toggle在未见过的新数据上的泛化能力。
- **关键结论**：Toggle在多个数据集上达成新的**性能最优**。例如，在CodeXGLUE代码完善任务上取得了新的SOTA结果；在Defects4J上，各Top-K指标均优于已有方法，在Top-10、30、50、100修复率上排名第一 (A Deep Dive into Large Language Models for Automated Bug Localization and

Repair)。特别地，精细的token级定位使得修复更精准，与仅行级定位的相比有明显优势。提供额外上下文（如相关注释）对定位有帮助，但增益有限；反之，**引入调整模型**对保证修复效果很重要 (A Deep Dive into Large Language Models for Automated Bug Localization and Repair)。

- **研究贡献**：该研究将LLM用于**细粒度故障定位**这一新颖方向，实现了在**不依赖额外标签**的情况下，通过LLM内部知识来确定bug具体位置，再配合另一个LLM生成修复，从而构建端到端APR系统 (A Deep Dive into Large Language Models for Automated Bug Localization and Repair) (A Deep Dive into Large Language Models for Automated Bug Localization and Repair)。Toggle框架的分离设计使不同阶段各自优化，并成功集成，证明了**逐步定位-修复**优于一体化黑盒生成。作者的分析也为如何设计有效prompt和结合代码注释等上下文提供了经验。
- **开放资源**：Toggle的实现以及用于训练/评估的数据已经在作者主页提供 (Publications - Soneya Hossain)。特别是作者开源了名为 “**BugsAPI**” 的接口，方便在不同LLM之间对接定位结果。

闭源通用LLM的缺陷定位与修复能力评估 (Jiang et al., LLM4Code@ICSE 2024) (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair) (Integrating Various Software Artifacts for Better LLM-based Bug Localization and Program Repair)

- **背景**：大量研究聚焦开源模型或特定LLM，而对商用**通用大型模型**（如ChatGPT、ERNIE Bot等）在软件缺陷定位和修复任务上的实际表现缺乏对比分析。此工作针对这一空白，对比评估了数种流行闭源LLM在这些任务上的能力，以指导业界如何选用通用LLM解决软件问题。
- **方法**：选取三种热门闭源通用LLM（OpenAI ChatGPT-3.5、百度ERNIE Bot 3.5、科大讯飞Spark 2.0），在标准缺陷基准上评测其**故障定位（FL）和自动修复（APR）性能 (Evaluating Fault Localization and Program Repair Capabilities of ...)。评测通过prompt向模型提供有缺陷的方法代码**以及**相关的测试断言和错误栈信息**，让模型输出怀疑的bug行位置或生成修复补丁 (Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs)。另外，将这些表现与一些传统学习式FL技术进行比较。
- **数据集**：使用 **Defects4J (Java)** 中筛选出的120个真实缺陷进行评估 (Evaluating Fault Localization and Program Repair Capabilities of ...)。每个缺陷提供给模型的信息包括失败的测试描述（断言信息）、错误堆栈，以及有缺陷的代码段。
- **实验设计**：分别测试各模型的**故障定位Top-K准确率**（例如Top-1, Top-3命中率）以及**修复成功率**。同时关注不同提示信息对结果的影响，例如有无堆栈、断言等辅助信息时性能差异 (Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose

LLMs)。并与已有研究结果比较，如AgentFL、LLMAO等最新LLM定位方法，以及学习型FL框架DeepFL等。

- **关键结论**：评估显示，不同通用LLM在软件缺陷任务上表现各异：ChatGPT-3.5 整体上在FL和APR上表现较为突出，而中国本土的ERNIE Bot和讯飞Spark在某些案例上也提供了独特正确结果，三者组合投票可提升总体效果 (Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs) (Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs)。例如，通过融合多模型答案，Top-1 定位准确率比单一ChatGPT提高了超过10%，说明各模型存在互补性 (Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs)。此外，引入验证提示让模型相互检查答案，可进一步提升FL性能约16% (Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs)。总体而言，闭源LLM（即使非专门训练代码的）在缺陷定位和简单修复上已经有一定能力，但仍有改进空间。
- **研究贡献**：该研究首次**系统比较**了多种主流闭源LLM在软件质量保障任务（缺陷定位、漏洞检测等）上的表现，发现了一些**意外之处**：有些较小或不知名的LLM在特定bug上能胜过ChatGPT，提示将不同LLM结果组合是提升效果的可行途径 (Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs) (Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs)。此外，作者提出的跨验证策略（LLM相互验证答案）为提高闭源LLM可靠性提供了新方法。此工作对实践中选择和融合通用LLM解决软件问题具有指导意义。
- **开放资源**：由于使用的是商用闭源模型，作者未提供模型源码，但分享了**提示模板**和缺陷样本，供后续研究复现实验过程。

AdaPatcher：带位置感知的自适应程序修复 (Dai et al., arXiv 2025) (Less is More: Adaptive Program Repair with Bug Localization and Preference Learning) (Less is More: Adaptive Program Repair with Bug Localization and Preference Learning)

- **背景**：大多数LLM修复研究追求生成正确补丁，但往往不顾及补丁与原代码的一致性，可能改动过多代码，令开发者难以接受 (Less is More: Adaptive Program Repair with Bug Localization and Preference Learning)。实际场景中，**最小改动地修复**（保持原代码风格和结构）同样重要。为此作者提出新的“自适应程序修复”任务和两阶段方法，既确保修复正确又尽量少改代码 (Less is More: Adaptive Program Repair with Bug Localization and Preference Learning)。
- **方法**：提出 **AdaPatcher** 两阶段框架，实现**自适应的**（尽量少改动的）自动修复 (Less is More: Adaptive Program Repair with Bug Localization and Preference Learning)。第一阶段为 **Bug 定位**

器：通过一种自我调试学习（self-debug learning）策略准确定位bug所在行。第二阶段为 **程序修改器**：以定位的bug行为基础进行小范围修改。修改器训练时引入**位置感知**（location-aware）机制，只根据标记的bug行上下文生成补丁，并采用**混合训练策略**强化对原代码的参考，最后通过**偏好学习**引导模型倾向于产生更小改动的补丁 (Less is More: Adaptive Program Repair with Bug Localization and Preference Learning)。同时Prompt中包含错误信息（如断言失败提示）来指明正确预期输出，确保补丁纠正功能错误 (Less is More: Adaptive Program Repair with Bug Localization and Preference Learning)。

- **数据集**：作者构建并开放了自适应修复评测数据集，包含若干有失败测试的代码缺陷案例，每个案例都要求在保证通过测试的前提下**尽可能少改代码**。另外也在常规的缺陷数据集上测试以比较与传统修复的差异。
- **实验设计**：对比AdaPatcher与多种基线（包括直接让LLM看代码+错误信息生成补丁的方式，以及其他APR工具），评估正确修复率和代码改动规模。采用指标如patch最小编辑距离、保持原代码片段比例等衡量“一致性”。也验证了定位器准确性对最终修复的影响。
- **关键结论**：AdaPatcher在实现高修复率的同时有效控制了补丁大小，显著领先于基线。在实例分析中，直接给LLM代码和错误信息常产生**大幅度改写**（甚至重写整个函数） (Less is More: Adaptive Program Repair with Bug Localization and Preference Learning)，而AdaPatcher生成的补丁更精简，修改更少的行却成功通过测试。实验证明，两阶段框架可以兼顾正确性和最小改动：有了精准的故障定位，引导LLM专注于bug附近修改，结合偏好学习，能生成贴近原代码风格的小补丁。
- **研究贡献**：该工作引入了“**自适应修复**”的新视角，强调补丁与原实现的一致性，为LLM自动修复增加了实用维度。AdaPatcher证明了**故障报告（如错误消息）+ 精准定位**可以让LLM生成让开发者更满意的补丁 (Less is More: Adaptive Program Repair with Bug Localization and Preference Learning)。同时，作者开源了模型和数据，推动社区关注**补丁质量**而非仅正确性。这对实际部署LLM修复工具具有借鉴意义。
- **开放资源**：AdaPatcher的源码和数据集已在GitHub发布 (Less is More: Adaptive Program Repair with Bug Localization and Preference Learning)。其中包括故障定位器、自适应修改器的训练代码，以及用于训练的标注数据（错误信息、偏好反馈等）。开发者可据此复现并改进模型。

参考文献：本文涉及的研究均来自近五年高水平会议或arXiv预印本，对应的完整论文可通过引用【】中的标注链接获取。

我们需要关注的点/借鉴的点

这次我们先不着急动手，先调查好看看现有的模型怎么做的，然后我们再动手。

1. DEVLoRe：
    - 配置上(a) 无任何故障报告信息，仅代码；(b) 仅 issue 描述；(c) 仅错误堆栈；(d) 仅调试信息；以及各种组合（issue+堆栈、堆栈+调试、issue+调试等），我们可以到时候也以这样组合的方

式搞一套横向对比

- 看论文需要捕捉的信息：
  a. 其问题报告是怎么来的，用了什么测试程序生成的，是否可以用在大部分代码上
  b. 其具体如何将问题报告结合的，如果有些困难，我们是否可以采取相对暴力的办法（比如复制黏贴这样的），或者有其他方法实现吗
  c. 关注一下这里面的数据集具体是什么样的，什么形式的
  d. 源码,尽量跑一跑

2. SWE-bench:提出 SWE-bench 基准，用于评价LLM在真实场景下解决GitHub Issue的能力。这个其实和我们做的不是很符合，它只是提出了一个基准，但是我们也可以关注一下，方便后续进行横线的对比
3. ChatRepair：
   - 关注一下它的 测试失败信息 是什么，怎么来的
   - 关注一下它迭代下的修复情况，我们也可以进行一个迭代测试（比如控制迭代次数，看看修补bug的能力如何）
   - 代码估计用处不大，可以选择性看，主要从论文里看看有什么可用的信息
4. FuseFL：自动定位故障，也是使用ChatGPT
   - 这个或许可以帮助我们改进实验方案，因为FuseFL的实验动机就是 让LLM“像人一样”对代码进行推理解释 ，我们可以借鉴它处理故障信息的方式，并且以此方式加入横向对比。
   - FuseFL 将三类信息融合进Prompt，关注一下分别是什么，以什么形式融合进去的，以及如何产生这些信息的
   - 关注数据集的形式，看看代码里面是如何处理数据集的（就是，导入数据集之后，怎么用LLM接上去）
5. AutoCodeRover：价值不大，没必要看感觉（）
6. Toggle：将LLM用于精细力度的缺陷定位，再用于修复
   - 可能是一个可行的点，就是是直接把代码给LLM让他修复，还是分两步，第一步缺陷定位，然后再...比如说给出定位到的代码及其上下文再修复，会不会效果更好。
   - 数据集也是 Defects4J ，可以关注一下他如何处理数据集的（如何导入，并且接到LLM上）
7. AdaPatcher:提供了一个新思路， 实际场景中，最小改动地修复（保持原代码风格和结构）同样重要 。我们可以关注这一点，其他的自适应修复什么的不需要关注。
   - 看看它如何评价“代码风格”的一致性，我们能不能也去对比代码风格的一致性