arXiv:2304.03816v1 [cs.SE] 7 Apr 2023

从自然语言问题描述生成语义正确的代码编辑

Sarah Fakhoury*
微软研究院

Saikat Chakraborty*
微软研究院

Madan Musuvathi
微软研究院

Shuvendu K. Lahiri
微软研究院

摘要

大语言模型（LLM），例如 OpenAI 的 Codex，已展现出根据自然语言描述为各类编程任务生成代码的潜力。近期出现了多项基准测试，用于评估大语言模型根据自然语言意图生成代码的能力，并通过一组隐藏测试用例检验其功能正确性。这使研究界能够识别大语言模型能力方面显著且可复现的进展。然而，目前仍缺少用于评估大语言模型能否根据预期更改的自然语言描述生成在功能上正确的代码编辑的基准数据集。本文旨在通过提出 *nl2fix* 这一问题来填补这一空白：将代码更改的自然语言描述（即代码仓库的问题报告中描述的错误修复）转化为正确的代码修复。为此，我们推出 *Defects4J-Nl2fix*，这是一个包含 283 个 Java 程序的数据集，这些程序来自广受欢迎的 Defects4J 数据集，并附有高层次的错误修复描述；我们还对多个最先进的大语言模型在此任务上的表现进行了实证评估。结果表明，这些大语言模型合计能够为 64.6% 的缺陷生成合理修复，而表现最佳的基于大语言模型的技术在该基准测试上可达到最高 21.20% 的 Top-1 准确率和 35.68% 的 Top-5 准确率。

1 引言

近年来，人们对使用大语言模型（LLM）完成各种软件开发任务的兴趣激增。例如，GitHubCopilot [17], 由OpenAI 的 Codex 模型提供支持， [11], 已展现出令人印象深刻的能力，能够根据自然语言描述和相关代码上下文生成代码。Copilot 的近期更新 [4] 也突出了根据自然语言指令转换代码的能力。除了 Copilot 之外，大语言模型还为其他几款商业化 AI 辅助软件开发产品提供支持，例如 Amazon CodeWhisperer、[1], GhostWriter [3] 和 Tabnine [5]。

对大语言模型代码生成能力的研究，得益于用于评估并常常提升模型在特定任务上表现的基准。例如，用于评估 Codex 模型的 HumanEval [11] 基准，在指导 OpenAI [38] 训练能力大幅提升的后继模型 GPT-4 方面发挥了重要作用。在此类离线基准测试上的提升，往往也意味着大语言模型在现实场景中执行类似任务（例如在集成开发环境中）的表现有所改善。事实上，近年来，自然语言代码生成任务（下文称为 *nl2code*）还出现了若干其他基准，包括 MBPP 等基础的众包编程基准测试（来自 Google） [6], 代码竞赛基准 [31], 以及基于真实世界程序的基准 [62]。

尽管有若干此类基准测试能够评估大语言模型根据自然语言意图生成具有功能正确性的代码的能力，但目前缺乏能够评估大语言模型根据自然语言意图执行 编辑 的基准测试（我们将其称为 *nl2edit*）。

在现实世界的软件开发中，与使用自然语言从头编写一个自包含的程序相比，用户更有可能在现有软件代码库的背景下执行此类软件演进和维护任务。此外，在采用云托管持续集成（CI）流水线的现代软件开发中，成熟的软件程序通常通过响应自然语言问题描述而创建的拉取请求 （PR）进行演进，这些问题描述对应于错误修复、优化、功能添加等。

在本文中，我们迈出了创建 第一步，旨在构建一个用于 *nl2edit* 的基准，并评估当前最先进的大语言模型在该问题上的表现。具体而言，我们聚焦于受限的 *nl2fix* 问题，其任务是修复存在缺陷的程序，其中缺陷在 Issue 描述中以自然语言进行描述。我们利用广受欢迎的 Defects4J [22] 基准（其中包含真实世界的缺陷及其修复），定义了用于 *Defects4J-Nl2fix* 的 *nl2fix* 基准。我们引入了一个扩充数据集，其中包含 283 个 Java 程序及其预期代码更改的高层描述，以及用于确保修复能够解决缺陷并保持原有功能的测试套件。我们还引入了用于在该基准上评估大语言模型解决 *nl2fix* 问题能力的指标。

图1 展示了一个来自*Defects4J-Nl2fix* 数据集的示例。该示例展示了一个*Issue* ，其中包含缺陷 ID，该 ID 引用了 JxPath 中的一个问题[1]，该项目托管于 Jira[2]；还包含一个标题，用于从高层次描述该缺陷；最后是一个描述，用于提供有关该缺陷的更多细节，包括以自然语言描述的缺陷复现场景。原始函数由包含修复内容的方法体构成——我们将数据集限制为修复局限于单个方法的情况。Issue 与存在缺陷的函数共同构成*nl2fix* 问题的输入。生成的补丁分别使用回归测试和触发测试来评估功能正确性和缺陷消除情况。集合中的每个测试 *t* 在触发测试集合中，都会在存在缺陷的程序上失败，并在以修复后的函数形式提供的用户提供的修复上通过。

仅将隐藏测试用于验证的动机来自几个方面。首先，先前的研究发现，大多数真实世界的缺陷修复问题都没有附带失败案例或触发条件

*贡献相同。

[1]https://issues.apache.org/jira//browse/JXPATH-149

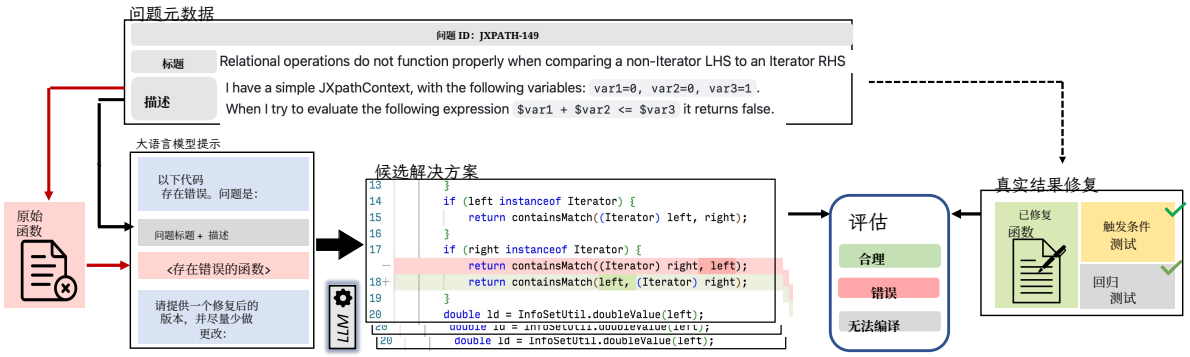


图1：nl2fix 问题设置概览。图中展示了 JXPATH-149 错误的问题标题、描述和合理修复。该图展示了用于生成候选补丁的标准大语言模型提示，以及如何使用真实触发条件和回归测试评估这些补丁。

测试 [23]。其次，尽管程序可能附带回归测试，但在持续集成流水线中运行这些测试可能代价高昂，实际操作中只能运行少数几次（例如 5 次）。正如稍后在相关工作（第 5 节）中所讨论的，隐藏测试使 *nl2fix* 问题有别于自动化程序修复（自动程序修复，APR）问题 [15, 16, 36]。

本文还对当前最先进（SOTA）大语言模型在该数据集上的表现进行了详细的实证评估。我们选择了三种基于 OpenAI 生成式预训练变换器（GPT）神经架构的大语言模型：(a) Codex 代码补全模型 `code-davinci-002`，(b) Codex 代码编辑模型 `code-davinci -edit-001`，以及 (c) ChatGPT 对话模型 `gpt-3.5 -turbo`。我们在不同的采样设置和提示策略下评估这些模型，并对所提出修复的准确性和质量进行了详细的定量与定性分析。

我们的结果表明，这些大语言模型结合起来能够生成合理的补丁（即，满足回归测试和触发测试），在最多采样 100 个候选补丁时，能够为相当大比例的缺陷生成补丁，占缺陷总数的 64.6%。更有意思的是，ChatGPT 模型`gpt-3.5-turbo` 在 pass@1、pass@5 和 pass@100 准确率指标方面均优于其他模型（我们将在第3.4.2节中更正式地介绍 pass@k 指标）。最后，我们描述了一种通用方法，该方法基于大语言模型计算的嵌入相似度对无序的候选补丁集合进行排序；排序后得到的建议具有确定性，其 Top-1 和 Top-5 准确率分别为 21.20% 和 35.68%。

这些发现既凸显了*Defects4J-Nl2fix* 基准的非平凡性，也体现了当前大语言模型构建良好初始基线的能力，并可推动后续研究。

贡献。 总之，本文的工作包括：(*i*) 我们阐明了*nl2fix* 问题，并提出了具有实质意义的增强基准*Defects4J-Nl2fix*及相应指标。

(*ii*) 我们对三种最先进的大语言模型在该基准上的表现进行了广泛的实证评估。

（*iii*）我们介绍一种基于嵌入相似度的排序策略，用于提供经过排序且确定的修复列表。

2 研究问题

本文旨在了解当前最先进的大语言模型在 *NL2Fix* 问题上的性能：即根据自然语言意图修复存在缺陷的程序。为此，我们提出了四个研究问题，以实证评估模型的能力：

研究问题 1：大语言模型能否根据自然语言意图为 NL2Fix 生成修复?

为回答这一研究问题，我们评估了三种最先进的大语言模型利用自然语言问题描述生成正确错误修复的 pass@k 准确率。此外，我们还对生成的修复进行了定量统计，包括重复建议的出现频率、编译比例，以及各方法产生合理补丁所针对的不同缺陷的分布与重叠情况。

研究问题 2：大语言模型会生成什么类型的候选修复?

为了深入了解大语言模型生成的候选补丁的特性，我们研究这些补丁的特点，考察它们与开发者编写的真实修复以及输入的有缺陷代码之间的相似性。

研究问题 3：大语言模型需要哪些信息才能为 NL2Fix 生成修复?

我们探究大语言模型需要何种程度的信息，才能根据自然语言描述正确生成修复。我们通过不同的提示风格，使用经过整理的信息进行实验，包括：高层次的问题摘要、深入的问题描述、零样本和单样本提示设置，以及通过推理提取提示策略生成的缺陷修复推理。

研究问题 4：能否使用大语言模型为 NL2F 的修复排序?ix?

基于研究问题 1、研究问题 2 和研究问题 3 的观察结果，我们探索如何利用大语言模型设计一种简单的排序方法，对无序的候选修复集合中的修复进行排序，从而更好地近似开发确定性且实用的错误修复推荐器所需的 pass@k 指标。

表1：数据集统计信息

项目	# 缺陷数	SH† Bugs	Avg. 代码片段	平均改动量 Line	令牌	问题长度 标题	描述。
图表	6	5	1.17	1.83	9.67	7.17	149.33
Cli	28	16	1.68	4.07	22.07	9.36	206.46
编解码器	11	9	1.27	2.18	10.82	12.73	171.09
集合	1	1	1.00	1.00	1.00	20.00	457.00
压缩	36	19	1.78	5.39	29.28	10.03	320.42
Csv	12	8	1.42	2.50	18.92	10.83	104.25
JacksonCore	13	9	1.38	3.69	20.69	11.38	251.69
JacksonDataBind	67	36	1.87	5.37	33.90	11.90	294.49
JacksonXml	5	1	2.80	6.20	30.80	11.60	126.80
JxPath	10	5	1.60	4.80	22.60	9.70	159.10
Math	73	37	2.11	5.29	35.00	10.07	165.18
Mockito	21	16	1.33	3.29	27.90	9.00	311.67
总体	283	162	1.78	4.65	28.76	10.53	231.92

†SH：单块

3 方法

3.1 数据集

在本文中，我们迈出了创建 第一步 来构建一个用于*nl2edit*的基准，并在该问题上评估当前最先进的大语言模型。我们聚焦于*nl2fix*这一受限问题，其任务是修复一个存在错误的程序，其中错误在问题描述中以自然语言进行描述。

我们选择 Defects4J [22] 基准，其中包含来自真实世界问题的缺陷和测试，我们可以从中提取问题描述。

特别是，我们使用 Defects4J 2.0，这是一个知名基准，包含从 17 个 Java 项目中收集的 835 个经过人工整理的真实世界缺陷及其修复。现有数据集由一组缺陷、用于复现缺陷的测试用例（触发测试）以及加载被测方法所在类的回归测试用例组成。Defects4J 数据集中的每个缺陷都包含一个 修复前_修复_版本 和修复后_修复_版本，分别代表代码的缺陷版本和修复版本。这两个版本反映了发现和修复缺陷时项目的实际状态。

我们使用这些开发者编写的测试来评估生成的补丁，补丁必须同时通过触发条件测试和回归测试，才能被视为合理补丁。虽然这并不能保证生成的补丁与真实结果修复之间具有语义等价性，但我们认为，出于以下两个原因，这可以作为补丁正确性的现实代理指标。首先，借助能够生成数百个候选补丁的大语言模型，手动评估每个生成补丁的语义等价性可能成本极高（每种模型配置约需 28,000）。因此，使用开发者编写的测试评估候选补丁，可以作为保持功能性和无缺陷性的可扩展代理指标。其次，与用户提供的修复具有语义等价性可能并非必要，因为在实践中，开发者可能存在多个彼此不等价但都可接受的修复方案。在不了解项目详细不变量的情况下，很难确定某个特定的真实结果修复是否是唯一可接受的修复。

我们通过三种不同的方式扩充 Defects4J 数据集：

• 我们将 *Defects4J-Nl2fix* 数据集限定为仅包含影响单个方法 体的修复。在 Defects4J 2.0 数据集的 835 个缺陷中，有 283 个属于单方法缺陷，即可以通过修改单个方法来修复的缺陷。修复可能包含多行代码行，但其作用范围仅限于单个函数。表 1 列出了各项目中缺陷数量的明细。
我们说明将重点放在单方法缺陷上的理由：与孤立代码行或整个文件或代码库相比，方法通常定义了一个可独立审查的代码单元。其次，大语言模型的输入提示仅限于几千个令牌，可能不足以涵盖整个文件或代码库层面的信息。使用 Defects4J 数据集的自动程序修复方法通常会将数据集限制为仅包含单个代码块或单行缺陷[59]。我们不作此限制，表 1展示了我们数据集中缺陷的平均代码块数量。

• 其次，为了解决*nl2fix* 问题，我们通过为 Defects4J 数据集中的每个缺陷配对相应的问题元数据（包括问题标题和描述）来扩充该数据集，这些元数据是我们从 GitHub、SVN 和 Jira 抓取的。

• 最后，通过仔细检查 De-fects4J 数据集中的错误方法，我们注意到，基准创建者采用的错误修补流程会产生一个副作用：出现在代码修复后_修复_修订版本 中的注释，也会出现在代码修复前_修复_修订版本 中[2]。这意味着，与开发者实际所做修复相关的注释，可能会出现在我们用作大语言模型输入的修复前_修复_修订版本 中。为避免这些注释向模型提供解决方案提示，我们移除了修复前_修复_修订版本中的所有注释。

3.2 生成式预训练变换器(GPT)

生成式预训练变换器（GPT）是大规模自回归 [21]生成模型，根据自然语言前缀（提示）上下文来预测下一个词元。近年来，参数量达数十亿的超大规模 GPT 模型的发展表明，这些模型展现出了涌现特性，能够在不进行微调的情况下执行任务 [25, 40]。在根据提示生成回答时，GPT 模型会逐个词元地从词元概率分布中采样。为了生成最可能的回答（或多个回答），这些模型会采用不同的采样方法，包括基于温度的采样；该方法会调整词元分布，从而控制回答的多样性。温度较低通常会降低多样性，温度较高则相反。我们在整个实验中使用两种温度值：0.2 和 0.8。

为回答已定义的研究问题，我们选取了三种基于 GPT 的先进大语言模型（LLM），它们在多种代码生成任务中展现出强大的能力[3]。

Codex。 OpenAI 的 Codex，`code-davinci-002` 是一种专为代码补全任务设计的语言模型。它基于 GPT-3 架构，并在来自公开代码仓库的大型代码语料库上进行了微调。Codex 擅长生成语法正确的代码，在涉及代码生成的任务中表现出很高的效能。

Codex 编辑模型。 Codex 编辑模型 `code-davinci-edit-001` 是具备编辑能力的 Codex GPT-3 版本。给定一段代码

[2]https://github.com/rjust/defects4j/issues/477[3]在提交时，作者无法访问最新的最先进模型 GPT-4 [38]

以及用自然语言编写的指令，例如“改进此函数的运行时复杂度”，模型会编辑代码，以可能满足该指令。

ChatGPT。 最近发布的 ChatGPT 模型（`gpt-3.5-turbo`）基于预训练的 GPT-3.5 模型，并进一步通过基于人类反馈的强化学习（RLHF）进行微调 [39]。虽然`gpt-3.5-turbo` 并未针对代码生成任务进行专门微调，但早期评估已证明它在科学和工程的多个领域展现出强大能力 [20, 26, 41, 43]，包括理解和生成代码片段 [33, 49, 50]。ChatGPT 的对话特性使其擅长同时需要代码生成和类人交互的任务，也因此能够使用涉及思维链 [57] 和推理提取[25]的高级提示结构。

嵌入。 OpenAI 的嵌入模型会为输入字符串生成高维向量表示。

研究表明，在这种高维向量空间中的相似性对应着字符串在语义上的相似性。除其他诸多应用外，这种表示方式还可用于相似性分析、搜索等。在这项工作中，我们利用 text-embedding-ada-002 模型生成代码的嵌入，并使用基于嵌入的相似性对补丁进行排序。

3.3 提示框架

大语言模型的能力并非在所有情境下都固定不变，也就是说，如果大语言模型答错了问题，稍微调整提示，修改所提供信息的内容或格式，就可能得到不同的结果。有多种技术可以提高大语言模型输出的准确性和可靠性，这些技术称为提示工程 [45]。本文采用标准提示，以及两种已被证明能够提升大语言模型处理复杂任务表现的不同策略：1）少样本提示；2）推理提取。这些策略旨在提供上下文和指导，帮助有效地解决任务，同时减少模型生成输出可能带来的陷阱。

3.3.1 零样本提示。 零样本提示也常被称为 标准提示，是使用任务提示模型的基本配置。提示中不包含任何可接受解决方案的示例，也不会将问题拆解为更简单的子问题。本文中的零样本提示（或标准提示）如图 1所示。它由问题标题和描述、有缺陷的代码，以及要求提供代码修复方案的指令组成。

3.3.2 少样本提示。 少样本提示是一种通过向模型提供一系列示例或演示来引导其理解当前任务的技术。通过为模型提供少量相似任务的实例及其各自的输入和期望输出，我们引导模型生成期望的输出，确保其功能和格式都符合要求。这种方法使模型能够根据提供的示例调整其响应，从而生成更准确、更连贯的输出。对于给定的问题，我们选择另一个问题及其有缺陷的代码和修复后的代码作为一个样例。我们使用标准的编辑距离度量，选择示例中有缺陷的代码与目标有缺陷的代码最接近的问题作为示例。

3.3.3 推理提取。 推理提取是一种专注于提取特定任务或问题背后潜在依据的策略 [25]。我们应用这一策略来帮助模型理解代码修复任务的目标和解决方案。具体而言，我们使用不同的查询与模型进行三次明确交互。首先，在给定有缺陷的代码和问题报告后，我们要求模型定位错误；然后要求模型解释被定位的代码行为何存在缺陷；最后要求模型修复该错误。Chat- GPT 的对话性质使得自然地使用能够维持对话上下文的高级提示结构成为可能，例如思维链推理和推理提取。因此，我们仅将这一提示策略用于`gpt-3.5-turbo`。

3.4 生成代码的正确性

实验分为两个阶段：修复生成和修复验证。所有验证实验均在 Docker 容器中运行，该容器使用 Ubuntu 20.04.4 和 Java OpenJDK 1.8.0；我们将公开 Docker 镜像。为生成候选修复，我们使用 OpenAI API。本节其余部分将介绍补丁验证的细节和评估指标：

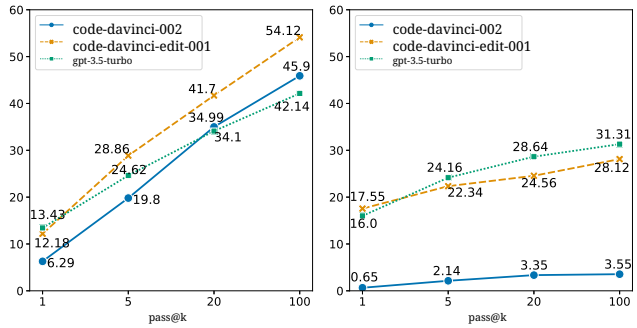
3.4.1 补丁验证。Defects4J 数据集中的每个缺陷都包含一个 `PRE_FIX_REVISION` 版本和一个`POST_FIX_REVISION` 版本，分别代表代码的缺陷版本和修复版本。这两个版本反映了发现和修复缺陷时项目的实际状态。为了确定生成的修复是否正确，我们执行以下步骤：1）检出项目的`PRE_FIX_REVISION` 版本；2）用生成的函数替换原有的缺陷函数；3）运行触发条件和回归测试，以确定包含生成修复的代码是否通过测试。

对于每个修复，验证结果可以是以下三种之一：1) 合理：所有复现错误的测试和回归测试均通过；2) 错误：至少有一个触发条件或回归测试失败；或 3) 无法编译。

3.4.2 评估指标。为了衡量*nl2fix*解决方案的质量，我们使用 pass@k 指标[4] 。该指标最初用于评估大语言模型在*nl2code* 问题上的表现，现已得到广泛应用 [7, 11]。直观而言，给定一组无序的候选修复，pass@k 表示从这组候选修复中抽样 k 次时选出正确修复的可能性。在*nl2fix* 场景中，如果某项修复通过了该缺陷的所有触发测试和回归测试，则该修复正确。给定n 为生成的样本数量， k 为用于估算 pass@k 的样本数量，且 c 为 n中的正确样本数量，我们使用下列公式计算 pass@k，其定义见 [11]:

$$\text{pass@k} := \mathop{\mathbb{E}}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{1}$$

对于本文，我们生成 $n \geq k$ 个样本，其中 $n = 100$。此外，为回答 RQ4，我们还引入了排名版 pass@k，记为 r.pass@k（第 4.4节），用于确定排名列表前 k 条建议中至少有一个代码修复正确的案例数。



(a) 温度=0.8 **(b) 温度=0.2**

图 2：零样本设置下的 pass@k 结果。4 结果

4.1 研究问题 1：大语言模型能否根据自然语言意图为 NL2Fix 生成修复?

对于每个模型，我们在零样本提示词下考虑两种设置：温度为 0.2 和温度为 0.8。对于每种设置，我们为 283 个缺陷分别生成 100 个候选修复，并根据触发条件和相关测试评估每个候选修复的正确性。接下来，我们使用第 3.4.2 节所述的方法计算 pass@k，并在图 2a 中绘制温度为 0.8 时的结果，在图 2b中绘制温度为 0.2 时的结果。

在温度为 0.8 时，我们发现编辑模型 `code-davinci-edit-001` 整体表现最佳，pass@100 达到 54.12%。`gpt-3.5-turbo` 的 pass@1 略高（为 13.93%，而编辑模型为 12.18%）；但从 pass@20 开始，其表现低于补全模型 `code-davinci-002`。补全模型的 pass@100 为 45.9%，位居第二，比编辑模型低 8.22%。

在温度为 0.2 时，我们发现 `code-davinci-edit-001` 和 `gpt-3.5-turbo` 的 pass@1 精确率均有所提升，分别达到 17.55% 和 16.0%。然而，从 pass@5 到 pass@100，所有模型的精确率都持续较低。尤为明显的是，`code-davinci-002` 在温度为 0.2 时的精确率要低得多，pass@1 仅为 0.65%，pass@100 也只有 3.55%。我们分别进行了两次模型精确率测试，发现模型在此设置下的表现始终不佳。

为了更好地了解各模型的 pass@k 准确率，我们提取了关于各模型在两种温度配置下生成的代码的高级统计数据。表 2 列出了每个缺陷对应的重复代码候选项平均占比，以及能够编译、通过回归测试和同时通过回归测试与触发测试（合理补丁）的候选项平均占比。

从表2 可以看出，温度降低时生成的重复项数量会大幅增加，这符合预期，因为较低温度下模型的行为更加确定。例如，专为代码编辑优化的`code-davinci-edit-001`在温度为 0.2 时生成的重复项超过 90%，而在温度为 0.8 时仅为 26%。

不同模型生成的代码中能够编译的比例存在显著差异。我们观察到，`code-davinci-002` 在温度为 0.2 时生成的代码中仅有 4.66% 能够编译，这解释了图 2b 中所示的极低准确率。然而，在温度为

[4]本文中，我们交替使用 pass@k 和 P@k

表2：各缺陷的候选补丁平均统计数据。

	code-davinci-edit-001		code-davinci-002		gpt-3.5-turbo	
温度。	0.2	0.8	0.2	0.8	0.2	0.8
重复	**90.7**%	26%	87.9%	**35.2**%	51.8%	8.41%
编译	**74.3**%	54.8%	4.66%	35.9%	57.2%	**60.4**%
回归	**51.2**%	**41.7**%	3.23%	20.19%	25.1%	33.0%
合理	**20.9**%	12.0%	0.65%	6.0%	16.0%	**13.4**%

0.8 编译率显著提高，介于 35.9% 和74.3%之间。参考相关工作，在 De-fects4J数据集的另一个子集上，最先进的神经自动程序修复技术生成的补丁在前 100 [59][32][19] 中的编译率为 15% 到28%。尽管自动程序修复的设置不同于NL2Fix，但触发条件和相关测试并未从自动程序修复设置中隐藏，我们观察到，将整个方法作为输入提供给大语言模型，在生成更高比例的可编译补丁方面具有优势。

两者 `gpt-3.5-turbo` 和 `code-davinci-edit-001` 在温度 0.8 设置下的 pass@100 精确率高于 0.2 设置（图 2b）。然而，在表 2 中，我们看到 0.2 设置下合理补丁的平均百分比更高。虽然这看起来有些违反直觉，但每个缺陷存在大量重复项，会影响跨缺陷计算出的 pass@k。换句话说，模型可能对少数缺陷具有较高的置信度，并为这些缺陷生成较高比例的合理补丁。

结果 1.1：仅凭对缺陷的自然语言描述，三个大语言模型都能为数据集中的少量缺陷生成合理修复，pass@1 最高为 6.29%–17.55%，pass@100 最高为 42.19%–54.12%。在零样本设置下，与 `code-davinci-002` 和 `gpt-3.5-turbo` 相比，`code-davinci-edit-001` 的总体准确率最高。

我们在表 3 中报告了每个项目中存在合理修复的缺陷数量。总体而言，我们观察到，各个模型生成的合理修复分布于所有项目中。通常，对于每个项目，这三个模型为相近比例的缺陷生成合理修复。也有少数值得注意的例外，例如仅包含 1 个缺陷的 Collections 项目，只有 `code-davinci-edit-001` 成功修复了该缺陷。`gpt-3.5-turbo` 在 Mockito 和 JacksonDatabind 项目上的准确率也较低，分别仅为 1/21 和 7/67 个缺陷生成了合理修复；相比之下，另外两个模型分别为 6/21 和至少 22/67 个缺陷生成了合理修复。不过，在 Codec 项目中，`gpt-3.5-turbo`生成的合理修复比另外两个模型多修复了两个缺陷。总体而言，各模型修复的缺陷总数与图 2a 中展示的 pass@100 指标一致。

虽然两个模型可能为某个项目修复了数量相同的缺陷，但被修复的具体缺陷可能不同。图 3 展示了每个模型能够生成合理补丁的缺陷数量之间的重叠情况。三个模型都能够为相同的 28%（283 个缺陷中的 82 个）生成合理补丁。然而，我们观察到，将三个模型结合起来后，它们能够为数据集中的 64%（283个缺陷中的 183个）生成合理补丁。每个模型都有一组独特的缺陷，另外两个模型无法为其生成合理补丁：`code-davinci-edit-001`能够处理 22 个独特缺陷，`gpt-3.5-turbo`能够处理 18 个，另外 10 个由

表3：各项目中含有合理补丁的缺陷。该项目中含有合理修复的缺陷数 / 缺陷总数。

方法	davinci-002	edit-001	GPT-3.5
图表	6/6	6/6	5/6
Cli	13/28	16/28	13/28
代码c	7/11	7/11	9/11
集合	0/1	1/1	0/1
压缩	15/36	19/36	18/38
Csv	10/12	8/12	9/12
JacksonCore	8/13	9/13	5/13
JacksonDatabind	22/67	27/67	7/67
JacksonXml	1/5	2/5	2/5
JxPath	3/10	5/10	4/10
Math	38/73	45/73	45/73
Mockito	6/21	6/21	1/21
总计	**129/283**	**151/283**	**118/283**

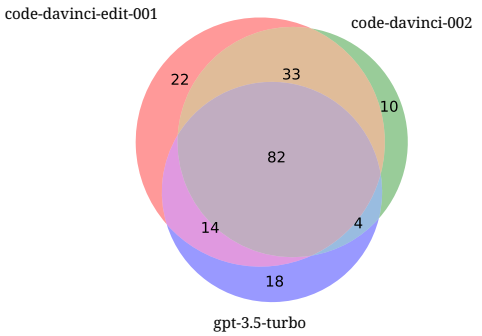


图 3：在零样本、温度为 0.8 的设置下，各大语言模型生成的合理补丁所针对缺陷的重叠情况。

code-davinci-002 尽管某些模型可能更容易修复某些缺陷，但从表3中可以看出，这似乎并非由缺陷所属的项目造成的一致性现象。

结果 1.2：数据集中 28%（82/283）的缺陷可由全部三个大语言模型修复。结合起来，三个大语言模型能够为数据集中 64%（183/283）的缺陷生成合理补丁。

4.2 研究问题 2：大语言模型会生成什么样的候选修复?

为回答此研究问题，我们从研究问题 1 中选取表现最佳的配置（温度 0.8），以更好地理解大语言模型生成代码的性质。我们研究不同模型生成的补丁的特征。为了解不同补丁的特征，我们研究这些补丁与有缺陷的代码（作为输入的一部分）以及实际修复代码之间的相似度。我们使用 CodeBLEU [44] 作为代表性相似度度量。给定两段代码 c_1 和 c_2，CodeBLEU 定义为$\alpha * B + \beta * W + \gamma * S + \delta * D$，其中 B、W、S、D分别是 c_1和 c_2之间的 BLEU分数、关键词BLEU分数、语法匹配分数和数据流匹配分数，而 α、β、γ、δ是权重。

表 4：模型生成的补丁与真实修复完全匹配时，缺陷和补丁的统计数据

模型	# 已修复 Bugs	# 精确匹配* Bugs	# 合理补丁	# EM 补丁
code-davinci-edit-001	151	32	1724	146
code-davinci-002	129	24	1182	120
GPT-3.5-turbo	118	11	2356	73

* EM：忽略空白字符的精确匹配

常数通常均设为 0.25 [5]。我们在本研究问题中选择 CodeBLEU，因为除了词汇匹配之外，它还考虑了代码之间的语法和语义匹配。

图 4a 展示了补丁与实际修复代码之间的相似性。在本文研究的全部三个模型中，通过回归测试和缺陷触发测试（即，合理补丁）的补丁，相较于不合理的补丁，与实际修复代码具有更高的 CodeBLEU。由于合理补丁通过了整个测试套件，这一结果是预期之中的；从理论上讲，合理补丁应当是实际修复的语义等价物，并表现出更高的 CodeBLEU。有趣的是，由 `gpt-3.5-turbo`模型生成的合理补丁，与实际补丁在 CodeBLEU 相似性方面表现出更高的变异性。对于 `code-davinci-edit-001` 和 `code-davinci-002`，合理补丁与实际修复代码之间的 CodeBLEU 四分位距（IQR）分别为 0.18 和 0.19。相比之下，对于 `gpt-3.5-turbo`，IQR 为 0.26。此外，对于`code-davinci-edit-001` 和 `code-davinci-002` 模型，峰度分别为 2.62 和 1.12，表明其分布更加集中；而 `gpt-3.5-turbo` 模型的峰度为 0.41，表明其生成能力更加多样，这一点从图 4a中可以明显看出。

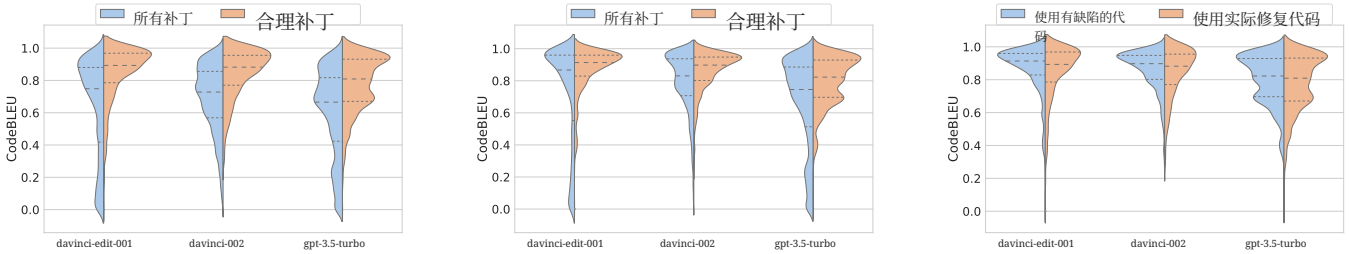
√ `gpt-3.5-turbo` 相比其他模型能生成更多样化的补丁。

此外，我们分析了生成的补丁与相对于有缺陷的代码的相似程度。图4b 展示了不同模型生成的各种类型补丁的 CodeBLEU 分布。在这三个模型中，有趣的是，我们观察到，合理补丁与有缺陷的代码之间的 CodeBLEU 高于非合理补丁。我们推测，当模型对输入的有缺陷的代码进行大幅修改时，生成的代码会产生各种问题，导致其无法通过编译、回归测试和触发测试（见表2）。我们推测，如果能够控制生成代码相对于输入有缺陷代码的偏离程度，大语言模型将对*NL2Fix* 问题产生更显著的影响。尽管如此，合理补丁表现出与有缺陷的代码更高相似度这一观察结果，为根据大语言模型生成的代码与有缺陷代码之间的相似度对其进行排序提供了新的可能性，我们将在接下来的研究问题中对此进行详细探讨。

√ 合理补丁与有缺陷的代码的相似度高于不合理的补丁。

学术界和业界普遍认为，davinci 或 gpt-3.5模型等大语言模型几乎都在所有开源软件上进行了预训练。因此，这些

[5]微软的 CodeBLEU 实现



(a) 使用实际修复代码。 **(b) 使用有缺陷的代码。** **(c) 输入有缺陷的代码时生成的合理补丁代码和实际修复代码。**

图 4：不同模型生成的补丁的代码相似性分析。我们在此分析 CodeBLEU 相似度。

```
1. 私有 布尔型 计算(对象 左, 对象 右){
   ...
9.    如果 (左 instanceof 迭代器 && 右 instanceof 迭代器){
10.       返回 findMatch((迭代器) left, (迭代器) right);
11. }
12.   如果 (left 属于 迭代器){
13.       返回 containsMatch((迭代器) left, right);
14. }
15.   如果 (right 属于 迭代器){
16.       返回 containsMatch((Iterator) right, left);
17. }
   ...
27. }
```

(a) 有缺陷的代码

```
1. private 布尔型 compute(Object left, Object right) {
   ...
9.    如果 (left instanceof 迭代器 && right instanceof 迭代器) {
10.       返回 findMatch((迭代器) left, (迭代器) right);
11. }
12.   如果 (left 是 迭代器){
13.       返回 containsMatch((迭代器) left, right);
14. }
15.   如果 (right 是 迭代器){
16.       返回 containsMatch(左侧, (迭代器) 右侧);
17. }
   ...
27. }
```

(b) 开发者编写的固定代码

```
1. 私有 布尔型 containsMatch(最终 迭代器 it, 最终 对象 value) {
2.     当 (it.hasNext()) {
3.         最终 对象 元素 = it.下一个();
4.         如果 (compute(element, value)) {返回 真;}
5.     }
6.     返回 假;
7. }
```

(c) **containsMatch** 函数的实现

```
1. private 布尔型 compute ( Object left , Object right ) {
   ...
9.    布尔型 isLeftIterator = (left instanceof 迭代器);
10.   布尔型 isRightIterator = (right instanceof 迭代器);
11.   如果 (isLeftIterator && isRightIterator) {
12.       返回 findMatch((Iterator) left, (Iterator) right);
13. }
   ...
22.   否则 如果 (isRightIterator) {
23.       迭代器 iter = (Iterator) right;
24.       当 (iter.hasNext()) {
25.           对象 下一个 = 迭代器。下一个();
26.           如果 (计算(左侧, 下一个)){返回 真;}
27. }
28.       返回 假;
29. }
   ...
39. }
```

(d) code-davinci-001 生成的修复后的代码（内联 **containsMatch** 函数）

图 5：展示 JxPath 项目 Bug ID 20 的实际修复代码与模型生成的合理补丁之间差异的示例。尽管生成的补丁未能与真实结果中的修复完全匹配，但它通过了所有回归测试和触发测试，因此与实际修复在语义上等价。

大语言模型在各自的预训练过程中已经观察过我们数据集中的方法。这引发了一个问题：大语言模型从其预训练中记忆了多少内容 [8, 34, 52]？遗憾的是，除非我们知道大语言模型的预训练数据是什么，否则没有很好的方法来衡量这一点。无论如何，为了定性地理解生成的补丁，我们研究了补丁与有缺陷的代码（该代码已作为输入提供给大语言模型）以及实际修复代码之间的 CodeBLEU 相似度。在全部三个模型中，生成的补丁与有缺陷的代码的相似度略高于与实际修复代码的相似度（见图 4c）。根据单侧 Wilcoxon 符号秩检验，这种差异具有统计显著性；对于 code-davinci-edit-001、code-davinci-002 和gpt-3.5-turbo，p 值分别为 $1.6 * 10^{-9}$、$9.9 * 10^{-5}$、$2.7 * 10^{-14}$。

表4 展示了与数据集真实结果修复完全匹配的补丁的汇总统计数据。code-davinci-edit-001 模型为 151 个缺陷正确生成了至少一个补丁，其中针对 32 个缺陷的补丁与真实结果完全匹配。对于code-davinci-002 模型，这一数字为 129 个中的 24 个；而对于gpt-3.5-turbo，则为 118 个中的 11 个。在合理补丁中，code-davinci-edit-001、code-davinci-002 和 gpt-3.5-turbo 的补丁与真实结果完全匹配的比例分别仅为 8.5%（146/1724）、10.1%（120/1182）和 3%（73/2356）。这些结果表明，大多数合理补丁在语法上都与真实结果修复不同。

在图5中，我们展示了gpt-3.5-turbo 为图1中的示例生成的一个合理补丁。有缺陷的代码将containsMatch 函数的参数放错了位置（图5（a）中第 16 行），而开发者编写的补丁则通过将参数放在正确的位置修复了错误（图5（b）中的第 16 行）。图5（d）展示了gpt-3.5-turbo 模型生成的修复后代码，它与开发者编写的代码在语义上等价。事实上，大语言模型生成的修复实际上将containsMatch （其实现如图5（c）所示）函数内联到了上下文中（图5（d）中的第 23 至 28 行）。此外，大语言模型生成的补丁通过提取两个变量重构了代码，这两个变量分别对应于

表 5：不同提示配置的准确率。

模型	提示	P@1	P@5	P@100
code-davinci-edit-001	问题标题	4.73	15.39	44.12
	零样本	12.18	28.86	54.12
	单样本	4.73	12.75	30.24
gpt-3.5-turbo	问题标题	8.06	17.69	37.72
	零样本	13.43	24.62	42.14
	单样本	**16.22**	**31.93**	**56.93**
	*R.E.	15.44	25.72	47.68
code-davinci-002	问题标题	0.25	1.22	14.94
	零样本	6.29	19.8	45.91
	单样本	3.92	14.34	40.09

*RE：推理提取

原始代码中使用的两个布尔型表达式，使得生成的代码更具可读性。我们观察到，尽管可以说大语言模型已经见识过所有开源软件，但在应用于 NL2Fix 问题时，它们仍会探索代码的新变体。

√ 大语言模型不一定总能记住预训练中的代码。它们会在尝试生成补丁时探索新的代码。

结果 2： 在我们研究的全部三个模型中，与非合理补丁相比，合理补丁与输入的有缺陷的代码表现出更高的相似度，中位相似度的差异为 ~0.15。合理补丁与有缺陷的代码的相似度也高于与实际修复后的代码的相似度，且具有统计显著性（p 值 << 0.05）。在最多 10% 的情况下，合理补丁与开发者编写的补丁完全匹配。

4.3 研究问题 3：大语言模型需要哪些信息来源来为 NL2Fix 生成修复？

为了确定生成合理修复需要哪些信息，我们采用不同的提示技术，为每个大语言模型提供不同层次的信息，并评估每种方法的 pass@k 指标。在研究问题 1 和研究问题 2 中，我们使用了基本的零样本提示，其中包含问题标题和描述，如第 3.3 节所述。

表5 展示了对这一基础提示词的修改如何影响以下提示词的 pass@k：

（1）零样本是研究问题 1-2 中使用的标准提示结构，如图所示1。（2）“问题标题”表示移除问题描述，同时保留提示结构的其余部分。（3）单样本会添加一个问题示例、对应的含缺陷方法以及正确的修复方案，供模型学习。提示的其余部分则保留原有格式。所选示例基于其与待修复代码的相似度。关于此选择的详细信息，请参见第 3.3 节。（4）对于推理提取，我们修改了提示，要求 `gpt-3.5-turbo` 将问题分解为两个步骤：1）定位原始方法中的含缺陷代码行；2）解释这些代码行为何存在缺陷，然后再请求修复方案。示例如图 6

在所有三种模型中，我们观察到，与使用问题标题和描述（零样本）相比，仅使用问题标题并不足以获得相当的 P@k，pass@1 会下降约 5%–8%，pass@100 会下降约 5%–30%。这表明，问题描述包含有助于模型生成合理修复的宝贵信息。

√ 问题描述为大语言模型解决 NL2Fix 问题提供了有用的上下文。

对于 `code-davinci-edit-001`和 `code-davinci-002`，我们观察到，在提示中为单样本方法添加示例并不能提升性能；事实上，与零样本尝试相比，所有 pass@k 都有所下降。在 1-shot 设置下，`code-davinci-edit-001` 生成的补丁中可编译的平均比例从零样本设置下的 54.8%（表 2）降至 24.7%。仔细查看 `code-davinci-edit-001`在单样本尝试中生成的补丁后，我们发现其中有几个包含了提示中单样本示例的代码。例如，问题 ID`MATH-58` 对应的原始错误方法具有如下方法签名： `public double[] fit(){...}`。单样本设置中使用的示例具有如下函数签名：`private boolean isShortOption (Stringtoken){..}`，而生成的六个补丁都修改了 `isShortOption()` 函数，而不是有缺陷的`fit()` 函数。

这表明模型尝试使用示例中的代码编辑目标函数。`code-davinci-edit-001` 模型需要输入代码和一组编辑指令。虽然 OpenAI 没有公开`code-davinci-edit-001` 的创建技术细节，但观察到的行为可能表明，该模型并未使用 k-shot 指令进行微调，这或许可以解释模型性能为何下降。此外，即提示中可使用的最大令牌数——上下文窗口——对于 `code-davinci-edit-001`要小得多，约为 3000 个令牌；相比之下，`code-davinci-002` 约为 8000 个令牌，`gpt-3.5-turbo`约为 4000 个令牌。因此，为确保提示中仍包含目标问题，会截断若干示例以适应上下文窗口，进而导致方法体格式错误。

在单样本设置下，`code-davinci-002`生成的可编译补丁平均比例从零样本设置下的 35.9%（表 2）降至 20.5%。查看生成的补丁后，我们注意到有大量补丁生成不完整，即，代码无法编译，因为生成的函数在语法上不正确。例如，缺少右花括号（}）、缺少 return 语句，以及生成内容在目标函数尚未完成时便中止。请注意，有两种已知方式可以停止补全任务——(a) 设置最大长度；(b) 设置特定的 STOP 词。我们将最大长度设为 750 个词元。此外，我们在提示末尾附上了修复后方法的签名，因此补全模型只需补全该方法的函数体。

√ 在单样本提示中使用的示例信息并不能提高 `code-davinci-edit-001` 和 `code-davinci-002` 在 nl2fix 问题上的 pass@k 准确率。

另一方面，与零样本尝试相比，`gpt-3.5-turbo` 在单样本和推理提取提示词下的 pass@k 均有所提升。例如，在 1-shot 设置下，pass@1从 13.43%提升至 16.22%；使用推理提取时，则提升至15.44%。图 6展示了推理提取提示词配置中的交互示例。该示例来自 JacksonDatabind

项目[6]。当被要求识别存在 bug 的代码行时，gpt-3.5-turbo 会在回复 1 中返回正确的缺陷区域。然后，提示 2 将原始提示 1 和回复 1 追加为提示 2 的上下文，并添加指令，要求解释为什么已识别的代码行包含 bug。在回复 3 中，gpt-3.5-turbo 解释了这个问题，并从问题描述中提取了一个会导致代码运行失败的示例输入及相应错误。在最终提示中，我们追加所有输入和回复，以请求提供 bug 函数的最终修复版本。这是一个示例：gpt-3.5-turbo 在推理提取设置下能够为其生成合理补丁，但在零样本设置下则不能。

√ 推理提取和单样本提示方法都有助于 gpt-3.5-turbo 生成更多合理补丁，并提高 pass@k。

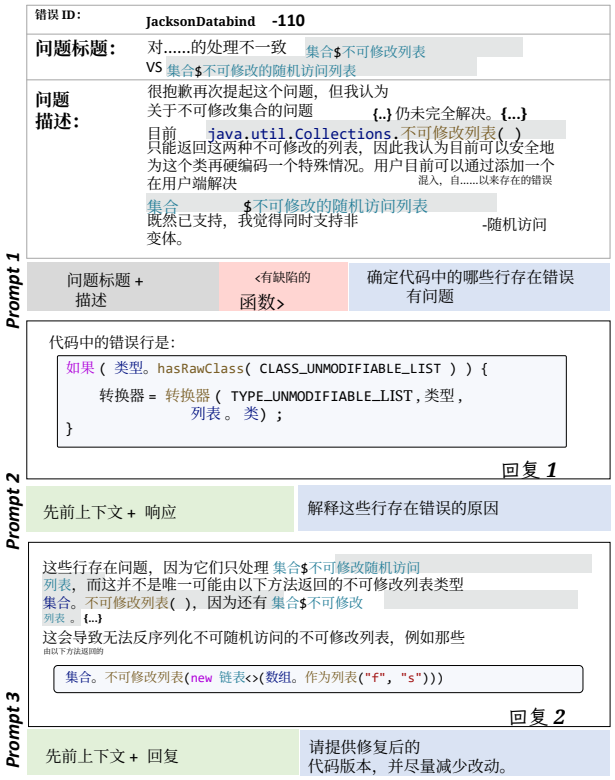


图 6：一个通过推理提取正确修复的错误。

然而，在 1-shot 设置下，gpt-3.5-turbo 能够为 56 个新缺陷生成正确的补丁，但对于零样本设置中的 14 个缺陷，它失去了修复能力。使用推理提取时，gpt-3.5-turbo 能够为 34 个新缺陷生成正确的补丁，但对于零样本设置中的 18 个缺陷，它失去了修复能力。与汇总所有方法相比，gpt-3.5-turbo 在推理提取设置下只能独自修复 4 个缺陷。查看每个示例的问题描述中包含的信息后，我们发现gpt-3.5-turbo 能够正确定位含缺陷代码行并推理其存在缺陷的原因，但这只能借助问题描述中的上下文。示例见图6 。尽管这些提示技术确实提升了总体性能指标，但也可能降低数据集中部分缺陷的修复效果。

结果 3： 问题描述为解决 nl2fix 问题提供了有用的上下文。能够提供示例（即，少样本提示）并分解任务（即，推理提取）的提示技术，显著提高了gpt-3.5-turbo在pass@k 等综合指标上的准确率；然而，在数据集的某些子集上，性能可能会下降，并且相较于标准提示，这并不能保证获得解决方案。

4.4 研究问题4：能否使用大语言模型为 NL2Fix 的修复排序?

回顾一下，对于给定问题（在我们的案例中是一个 bug），给定一个包含 n 个候选解且其中有 c 个正确解的无序集合，pass@k 指标表示在 k 次尝试内至少选出一个正确解的可能性。这一统计量对于评估语言模型很有用，但无法直接提供一个实用的真实世界推荐系统，以确定性地向用户提出少量候选修复（例如最多 $k = 5$ 个）。例如，对于 $n = 100$ 和 $k = 5$，从 100 个样本中选择 5 个解有 $\binom{n}{k} > 75$ 百万种方式。对于*NL2Fix* 这一实用工具，我们希望开发一种（i）确定性地排序 这些建议并向用户呈现排名前 k 的建议的方法，同时（ii）保持较高的准确率，使其接近平均 pass@k 指标。

在本节中，我们利用大语言模型提出一种简单且通用的排序策略，以帮助实现上述两个目标（i）和（ii）。具体而言，受研究问题 2 的研究发现启发，我们探究能否利用输入的有缺陷函数与生成的补丁之间的嵌入相似度来识别合理补丁。我们使用 OpenAI 的 text-embedding-ada-002 嵌入模型为所有有缺陷函数及其对应补丁生成嵌入，详情见第 3 节。我们计算每一对有缺陷代码及其对应候选补丁嵌入之间的余弦相似度，并用该分数剔除相似度低于中位数（0.95）的补丁。为保持一致，我们对所有模型固定使用这一数值。接着，为避免对相似度极高的补丁（例如，1.0）进行排序，我们按余弦相似度从低到高（从 0.95 开始）排列补丁。根据研究问题 2 的观察，分数较低的补丁更可能属于错误或无法编译的补丁分布。补丁应与有缺陷的输入程序足够接近，但又不能过于接近，以免改动不够显著。

对于每个模型，我们在两组补丁上应用排名方案：(*a*)所有生成的补丁，以及(*b*)通过编译器检查的生成补丁子集。

为了评估排序策略，我们为每个模型选择了在研究问题 3 中 pass@1 与 pass@100 差异最大的 LLM 配置（见表 5）；这也恰好是每个模型表现最佳的配置。

表 6 展示了两组补丁的排名 pass@1和排名 pass@5 准确率（记作 r.P@k）：1）之前（记作

[6]https://github.com/fasterxml/jackson-databind/issues/2265

表 6：采用和不采用编译器剪枝时的排序准确率。

方法	指标	GPT-3.5 单样本	davinci-edit 零样本	davinci-002 零样本
All	排名精确率@1（精确率@1）	16.96 (16.22)	13.78 (12.18)	11.30 (6.29)
	r.P@5 (P@5)	31.8 (31.93)	30.74 (28.86)	17.66 (19.8)
剪枝	排名精确率@1（精确率@1）	21.20 (22.22)	19.78 (18.99)	17.66 (16.89)
	r.P@5（P@5）	34.98 (39.66)	35.68 (36.73)	34.2 (33.97)

作为 All）以及 2）在剪枝（记作剪枝）编译器错误之后。作为参考，我们还在括号中报告了P@k 指标。我们观察到，剪枝编译器错误确实改善了指标 P@k，尤其是 `davinci-002` 模型的 P@5 提高了超过 14%（从19.8提高到33.97）。其次，在提供确定性的同时，对于大多数配置，r.P@1都优于 P@1，除了 `gpt-3.5-turbo` 在剪枝情况下略有下降。对于 `davinci-002`，排序使 r.P@1提高了超过5个百分点。最后，对于大多数配置，r.P@5仍接近 P@5，只有 `gpt-3.5` 例外，其 r.P@5 比 P@5 低 4.68%。

结果 4： 通过剪除编译失败项，大语言模型的 pass@5 最高可达 39.66%。通过计算候选补丁与有缺陷输入代码嵌入向量之间的余弦相似度，我们可以采用确定性排序策略，在保持较高准确率的同时实现 34.2% - 35.68% 的 r.pass@5。

5 相关工作

我们的工作与两个广泛的研究方向最为相关：（a）自动代码编辑；（b）自动程序修复（APR）。

自动化代码编辑。 早期有关学习代码编辑的研究包括学习如何编辑代码以进行重构 [35, 46], 学习针对代码分析器发现的缺陷进行语义代码更改 [47]。近期方法利用深度学习技术，从 GitHub 上挖掘的代码更改中学习常见的代码编辑模式 [9, 13, 53]。

除了从历史更改中学习外，近期一些方法 [10, 63] 提出利用提交消息等辅助输入来指导代码编辑。我们认为，提交消息是代码更改的事后描述，并未捕捉更改的意图，而只是对更改的总结。相比之下，我们在*nl2Fix* 问题中考虑的问题报告，是更改的事前描述，可以说更能贴近地捕捉其意图。Tufano等人 [55] 提出根据审查者的评论编辑代码，以实现代码审查活动自动化。尽管这些方法与我们的工作最为接近，但两者有所不同：我们使用测试和 pass@k 等语义指标，来验证所提出补丁的功能正确性并确认其无缺陷。

自动程序修复。

自动程序修复（APR）的方法大致分为基于搜索的技术和基于机器学习的方法。有关自动化程序修复技术的全面概述，请参阅近期对该领域进行综述的研究 [15, 16, 18, 36]。

基于搜索的技术采用生成并验证 方法：生成原始代码的多个变体，然后使用未通过的测试进行评估[24, 42, 48, 51, 58]。这些方法通过不同的变换来处理有缺陷的代码，包括随机变换 [30, 42, 58], 人工设计的变换 [48], 以及从以往代码语料库中学习到的变换 [12, 24, 29, 54]。

近期，研究人员也将大语言模型用于自动程序修复。Alpharepair [59]采用完形填空式自动程序修复方法，由大语言模型根据含缺陷程序的上下文直接填入正确代码。Xia 等人 [60] 使用 ChatGPT 构建对话式自动程序修复问题，利用失败补丁的反馈来补充后续给大语言模型的提示词。此外，还有一些方法利用大语言模型根据问题描述生成自动程序修复所需的触发测试 [23]，以及辅助自动程序修复中的根因分析 [37]。最后，Fan 等人 [14] 利用自动程序修复方法（包括基于大语言模型的方法）修复根据自然语言意图生成的代码。

尽管与自动程序修复密切相关，我们的方法仍有细微差异：隐藏测试仅用于评估，绝不会作为修复算法的输入。这使得我们的方法更适用于现实世界中的错误修复，因为在推理时，可能没有失败测试可用于帮助分析含缺陷代码行的根因并验证修复，或者利用这些测试的成本高得难以承受。

6 局限性与威胁

模型输出的稳定性。 由于我们使用 OpenAI Web API 访问不同模型，因此无法控制模型输出的随机性。模型本身也经常更新。这对本研究的可复现性构成威胁。为减轻这一威胁，我们公开了模型生成的所有输出。

关于补丁正确性的假设。在本文中，我们利用测试来判断修复是否合理。然而，这样的合理补丁可能无法彻底修复错误。这可能需要人工分析，或与用户提供的修复进行语义等价性检查。然而，对于*Defects4J-Nl2fix*这样的大规模基准测试，前者并不可行，而语义等价性检查技术也无法扩展到适用于大多数真实程序。鉴于测试套件永远无法穷尽所有情况，我们可以借鉴近期关于补丁正确性 [28, 56, 61] 的研究，以增强对这些补丁的信心。

研究发现的普适性。 鉴于*Defects4J-Nl2fix* 基准所考察的缺陷数量相对较少（283 个），我们的发现可能无法推广到不同编程语言和软件仓库中的任意缺陷。为减轻这一威胁，我们使用了来自开源项目的真实缺陷。

7 结论

在本文中，我们阐述了*nl2fix* 问题，定义了首个基准*Defects4J-Nl2fix*，并对该问题上的各种最先进的大语言模型进行了详细的实证评估。

我们相信，*nl2fix* 这一任务以及诸如 *Defects4J-Nl2fix* 这样的具有挑战性的基准测试，将作为评估未来生成此类大语言模型（例如 GPT-4）的重要现实世界基准，同时利用此类大语言模型的新涌现行为（例如这些模型预测正确性的能力），以提升在此类基准测试上的性能。在未来的工作中，我们计划结合问题驱动的触发测试生成 [23] 和用户参与式 [27] 来增强对生成的修复的信任，并将我们的框架扩展到更一般的 *nl2edit* 问题，以

涵盖程序演进的其他形式，包括功能添加、重构以及优化。

参考文献

[1] [无日期]。Amazon CodeWhisperer。https://aws.amazon.com/codewhisperer/。访问日期：2023年3月29日。[2] [无日期]。Atlassian - JIRA。https://www.atlassian.com/software/jira。访问日期：2023年3月29日。[3] [无日期]。Ghostwriter。https://replit.com/ai。访问日期：2023年3月29日。[4] [无日期]。GitHub Copilot × AI 驱动的软件开发的未来。h t t p s:// w ww.linkedin.co m/pulse/github-copilot-x-future-ai-powered-software-development-github/。访问日期：2023年3月28日。[5] [无日期]。Tabnine。https://www.tabnine.com/。访问日期：2023年3月29日。[6]Jacob Austin、AugustusOdena、Maxwell Nye、Maarten Bosma、Henryk Michalewski、David Dohan、Ellen Jiang、CarrieCai、Michael Terry、QuocLe 等。2021年。使用大语言模型进行程序合成。arXiv 预印本 arXiv:2108.07732（2021年）。[7]Jacob Austin、AugustusOdena、Maxwell Nye、Maarten Bosma、Henryk Michalewski、David Dohan、Ellen Jiang、Carrie Cai、Michael Terry、Quoc Le 和 Charles Sutton。2021年。使用大语言模型进行程序合成。https://doi.org/10.48550/ARXIV.2108.07732[8] Nicholas Carlini、Daphne Ippolito、Matthew Jagielski、Katherine Lee、FlorianTramer 和ChiyuanZhang。2022年。量化神经语言模型中的记忆。arXiv 预印本 arXiv:2202.07646（2022年）。 [9] Saikat Chakraborty、Yangruibo Ding、MiltiadisAllamanis和Baishakhi Ray。2020年。Codit：使用基于树的神经模型进行代码编辑。IEEE软件工程汇刊 48，4（2020年），1385–1399。 [10] SaikatChakraborty 和 BaishakhiRay。2021年。源代码编辑的多模态学习。在2021年第36届IEEE/ACM自动化软件工程国际会议（ASE）中。IEEE，443–455。[11] Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、JaredKaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman、Alex Ray、Raul Puri、Gretchen Krueger、Michael Petrov、Heidy Khlaaf、Girish Sastry、Pamela Mishkin、Brooke Chan、ScottGray、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、AlexNichol、Alex Paino、Nikolas Tezak、Jie Tang、Igor Babuschkin、Suchir Balaji、Shantanu Jain、William Saunders、Christopher Hesse、Andrew N. Carr、JanLeike、Josh Achiam、Vedant Misra、Evan Morikawa、Alec Radford、Matthew Knight、Miles Brundage、Mira Murati、KatieMayer、PeterWelinder、Bob McGrew、Dario Amodei、Sam McCandlish、Ilya Sutskever 和 Wojciech Zaremba。2021年。在代码上训练的大语言模型的评估。https://doi.org/ 10 48550 2107 03374 。/ARXIV. 。[12] Zimin Chen、Steve Kommrusch、Michele Tufano、Louis-Noël Pouchet、Denys Poshyvanyk 和 Martin Monperrus。2019年。Sequencer：用于端到端程序修复的序列到序列学习。IEEE软件工程汇刊 47，9（2019年），1943–1959。 [13] Elizabeth Dinella、Hanjun Dai、ZiyangLi、Mayur Naik、LeSong 和Ke Wang。2020年。Hoppity：学习图变换以检测和修复程序中的缺陷。在国际学习表征会议（ICLR）中。[14] Zhiyu Fan、Xiang Gao、Martin Mirchev、Abhik Roychoudhury 和 Shin Hwei Tan。2023年。基于大语言模型的程序自动修复。arXiv: 2205 10583 [ ]。cs.SE Xiang Gao、YannicNoller和 AbhikRoychoudhury。2022年。[15] 程序修复。arXiv: 2211 12787 [ ]。cs.SE Luca Gazzola、DanielaMicucci和[16] Leonardo Mariani。2019年。软件自动修复：综述。IEEE软件工程汇刊 45，1（2019年），34–67。https://doi.org/ 10 1109 2017 2755013 。/TSE. 。GitHub。 。GitHub Copilot。访问日期：2022年8月5日。[17] 2022 https://github.com/features/copilot/。 [18] Claire Le Goues、MichaelPradel 和Abhik Roychoudhury。2019年。自动化程序修复。Commun. ACM 62，12（2019年），56–65。 [19] Nan Jiang、Thibaud Lutellier 和Lin Tan。2021年。Cure：用于自动化程序修复的代码感知神经机器翻译。在2021年第43届IEEE/ACM软件工程国际会议（ICSE）中。IEEE，1161–1173。[20] WenxiangJiao、Wenxuan Wang、Jen-tse Huang、Xing Wang 和ZhaopengTu。2023年。ChatGPT 是一名优秀的翻译员吗？一项初步研究。arXiv 预印本 arXiv:2301.08745（2023年）。[21] SørenJohansen。1995年。协整向量自回归模型中基于似然的推断。牛津大学出版社。 [22] René Just、Darioush Jalali和Michael D Ernst。2014年。Defects4J：现有故障数据库，以支持针对Java 程序的受控测试研究。在2014年国际软件测试与分析研讨会论文集中。437–440。 [23] Sungmin Kang、Juyeon Yoon和Shin Yoo。 [无日期]。大语言模型是少样本测试器：探索基于大语言模型的通用错误复现。在IEEE/ACM软件工程国际会议（ICSE）中。IEEE，（即将出版）。 [24] Dongsun Kim、Jaechang Nam、JaewooSong 和Sunghun Kim。2013年。从人工编写的补丁中学习自动补丁生成。载于 2013年第35届软件工程国际会议（ICSE）。IEEE，802–811。[25] Takeshi Kojima、Shixiang Shane Gu、Machel Reid、Yutaka Matsuo和Yusuke Iwasawa。2022年。大语言模型是零样本推理器。arXiv预印本 arXiv:2205.11916（2022年）。[26] Tiffany H Kung、Morgan Cheatham、Arielle Medenilla、Czarina Sillos、Lorie De Leon、Camille Elepaño、Maria Madriaga、Rimel Aggabao、Giezel Diaz-Candido、James Maningo等。2023年。ChatGPT 在美国医师执照考试（USMLE）中的表现：使用大语言模型辅助医学教育的潜力。PLoSdigitalhealth 2，2 2023年 0000198（），e。[27] Shuvendu K. Lahiri、Aaditya Naik、Georgios Sakkas、Piali Choudhury、Curtis von Veh、Madanlal Musuvathi、Jeevana Priya Inala、ChenglongWang和 Jianfeng Gao。2022年。通过测试驱动的用户意图形式化实现交互式代码生成。arXiv预印本 arXiv:2208.05950（2022年）。[28] Xuan-Bach D Le、Lingfeng Bao、David Lo、Xin Xia、ShanpingLi和CorinaPasareanu。2019年。关于补丁正确性评估的可靠性。载于 2019年 IEEE/ACM第41届软件工程国际会议（ICSE）。IEEE，524–535。[29] Xuan BachD Le、DavidLo和ClaireLe Goues。2016年。历史驱动的程序修复。载于 2016年 IEEE 第 23 届软件分析、演化与再工程国际会议（ SANER ），第 1 卷。IEEE，213–224。[30] Claire Le Goues、ThanhVu Nguyen、Stephanie Forrest 和 WestleyWeimer。2012年。GenProg：一种通用的软件自动修复方法。IEEE软件工程汇刊 38，1（2012年），54–72。https://doi.org/10.1109/TSE.2011.104[31] Zhiyu Li、Shuai Lu、Daya Guo、NanDuan、ShaileshJannu、GrantJenks、Deep Majumder、Jared Green、AlexeySvyatkovskiy、ShengyuFu等。2022年。通过大规模预训练自动化代码审查活动。载于第30届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集。1035–1047。[32] Thibaud Lutellier、Hung Viet Pham、Lawrence Pang、Yitong Li、MoshiWei 和 LinTan。2020年。Coconut：使用集成方法结合上下文感知神经翻译模型进行程序修复。载于第29届ACMSIGSOFT 软件测试与分析国际研讨会论文集。101–114。[33] Paula Maddigan 和 Teo Susnjak。 [n. d.]。Chat2vis：使用 ChatGPT、Codex 和 GPT-3 大语言模型通过自然语言生成数据可视化。载于IEEE/ACM软件工程国际会议（ICSE）。IEEE，（即将发表）。[34] R ThomasMcCoy、Paul Smolensky、Tal Linzen、Jianfeng Gao 和 Asli Celikyil maz。2021年。语言模型会从训练数据中复制多少内容？使用 raven 评估文本生成中的语言新颖性。arXiv 预印本 arXiv:2111.09509（2021年）。[35] Na Meng、Lisa Hua、Miryung Kim和 Kathryn SMcKinley。2015年。自动化重构是否可以避免系统化编辑？载于 2015年IEEE/ACM第37届IEEE 软件工程国际会议，第1卷。IEEE，392–402。[36] Martin Monperrus。2018年。软件自动修复：一份书目。ACM Comput. Surv. 51，1，文章17（2018年1月），24页。https://doi.org/10.1145/ 3105906[37] Manish Motwani 和 YuriyBrun。 [n.d.]。通过结合使用错误报告和测试改进自动程序修复。[38] OpenAI。2023年。GPT-4 技术报告。arXiv:2303。08774 [cs.CL][39] Long Ouyang、Jeffrey Wu、Xu Jiang、Diogo Almeida、Carroll Wainwright、Pamela Mishkin、Chong Zhang、Sandhini Agarwal、KatarinaSlama、AlexRay 等。2022年。训练语言模型以遵循人类反馈中的指令。神经信息处理系统进展 35（2022年），27730–27744。[40] Ajay Patel、Bryan Li、Mohammad Sadegh Rasooli、Noah Constant、Colin Raffel 和 ChrisCallison-Burch。2022年。双向语言模型同样是少样本学习器。arXiv 预印本 arXiv:2209.14500（2022年）。[41] Junaid Qadir。2022年。ChatGPT时代的工程教育：生成式人工智能用于教育的前景与陷阱。（2022年）。[42] Yuhua Qi、Xiaoguang Mao、YanLei、ZiyingDai 和Chengsong Wang。2014年。随机搜索在自动化程序修复中的作用。载于第36届软件工程国际会议论文集。254–265。[43] ChengweiQin、Aston Zhang、Zhuosheng Zhang、Jiaao Chen、Michihiro Yasunaga 和 DiyiYang。2023年。ChatGPT 是通用的自然语言处理任务求解器吗？arXiv 预印本 arXiv:2302.06476（2023年）。[44] ShuoRen、Daya Guo、Shuai Lu、Long Zhou、ShujieLiu、DuyuTang、Neel Sundare-san、MingZhou、AmbrosioBlanco 和Shuai Ma。2020年。CodeBLEU：一种用于代码合成自动评估的方法。arXiv 预印本 arXiv:2009.10297（2020年）。[45] LariaReynolds 和 KyleMcDonell。2021年。面向大语言模型的提示编程：超越少样本范式。载于 2021年 CHI 人机交互领域因素会议扩展摘要。1–7。[46] Reudismam Rolim 、 Gustavo Soares、Loris D'Antoni、Oleksandr Polozov、Sumit Gulwani、Rohit Gheyi、RyoSuzuki 和Björn Hartmann。2017年。从示例中学习句法程序变换。载于 2017年 IEEE/ACM第39届软件工程国际会议（ICSE）。IEEE，404–415。[47] Reudismam Rolim、Gustavo Soares、Rohit Gheyi、Titus Barik 和Loris D'Antoni。2018年。从代码仓库中学习快速修复。arXiv 预印本 arXiv:1803.03806（2018年）。

[48] Ripon K Saha、Yingjun Lyu、Hiroaki Yoshida 和 Mukul R Prasad。2017年。Elixir：有效的面向对象程序修复。载于*2017*年 第*32*届*IEEE/ACM*自动化软件工程国际会议（*ASE*）。IEEE，第648–659页。[49] Dominik Sobania、Martin Briesch、Carol Hanna 和 Justyna Petke。2023年。ChatGPT自动错误修复性能分析。*arXiv*预印本 *arXiv:2301.08653*（2023年）。[50] Nigar M Shafiq Surameery 和 Mohammed Y Shakor。2023年。使用ChatGPT解决编程错误。《国际信息技术与计算机工程杂志》（*IJITC*）*ISSN*：*2455-5290* 3，01（2023年），17–22。[51] Shin Hwei Tan 和 Abhik Roychoudhury。2015年。relifix：软件回归的自动修复。载于*2015*年 *IEEE/ACM* 第*37*届*IEEE*软件工程国际会议，第1卷。IEEE，第471–482页。[52] Kushal Tirumala、Aram Markosyan、Luke Zettlemoyer 和 Armen Aghajanyan。2022年。无需过拟合的记忆：分析大语言模型的训练动态。神经信息处理系统进展 35（2022年），38274–38290。[53] Michele Tufano、Jevgenija Pantiuchina、Cody Watson、Gabriele Bavota 和 Denys Poshyvanyk。2019年。通过神经机器翻译学习有意义的代码变更。载于*2019*年 *IEEE/ACM* 第*41*届软件工程国际会议（*ICSE*）。IEEE，第25–36页。[54] Michele Tufano、Cody Watson、Gabriele Bavota、Massimiliano Di Penta、Martin White 和 Denys Poshyvanyk。2018年。通过神经机器翻译实证研究真实环境中的错误修复补丁学习。载于第*33*届*ACM/IEEE*自动化软件工程国际会议论文集。第832–837页。[55] Rosalia Tufano、Luca Pascarella、Michele Tufano、Denys Poshyvanyk 和 Gabriele Bavota。2021年。迈向代码审查活动自动化。载于*2021*年 第*43*届*IEEE/ACM*软件工程国际会议（*ICSE*）。IEEE，第163–174页。

[56] ShangwenWang、MingWen、Bo Lin、Hongjun Wu、Yihao Qin、DeqingZou、Xiaoguang Mao和Hai Jin。2020年。自动化补丁正确性评估：我们走了多远？载于第*35*届*IEEE/ACM*自动化软件工程国际会议论文集。968–980。[57] JasonWei、Xuezhi Wang、DaleSchuurmans、Maarten Bosma、EdChi、Quoc Le 和Denny Zhou。2022年。思维链提示能够在大语言模型中激发推理能力。*arXiv*预印本 *arXiv:2201.11903*（2022年）。[58] Westley Weimer、ZacharyP Fry和 Stephanie Forrest。2013年。利用程序等价性进行自适应程序修复：模型与初步结果。载于*2013*年第*28*届*IEEE/ACM*自动化软件工程国际会议（*ASE*）。IEEE，356–366。[59] ChunqiuSteven Xia 和Lingming Zhang。2022年。少训练，多修复：通过零样本学习重新审视自动化程序修复。载于第*30*届*ACM*欧洲软件工程会议与软件工程基础研讨会论文集（新加坡，新加坡）（*ESEC/FSE2022*）。计算机协会，纽约，NY，美国，959 971 10 1145 3540250 3549101 – 。https://doi.org/。/ 。[60] ChunqiuStevenXia和 Lingming Zhang。 。对话式自动程序修复。arXiv：2023 2301 13246 [ ] 。cs.SE [61] Yingfei Xiong、Xinyuan Liu、MuhanZeng、Lu Zhang和GangHuang。2018年。在基于测试的程序修复中识别补丁正确性。载于第*40*届软件工程国际会议论文集。789–799。[62] Hao Yu、Bo Shen、Dezhi Ran、Jiaxin Zhang、Qi Zhang、Yuchi Ma、Guangtai Liang、YingLi、Tao Xie 和 Qianxiang Wang。2023年。CoderEval：使用生成式预训练模型进行实用代码生成的基准。*arXiv*预印本 *arXiv:2302.00288*（2023年）。[63] Jiyang Zhang、Sheena Panthaplackel、PengyuNie、JunyiJessy Li和Milos Glig- oric。2022年。CoditT5：源代码与自然语言编辑的预训练。载于第*37*届*IEEE/ACM*自动化软件工程国际会议。1–12。