软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007613] [CSTR: 32375.14.jos.007613]

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

基于大语言模型的受约束代码生成研究进展*

李宇轩[1,2], 邹艳珍[1,2], 王 玥[1,2], 谢 冰[1,2]

[1](高可信软件技术教育部重点实验室 (北京大学), 北京 100871)

[2](北京大学 计算机学院, 北京 100871)

通信作者: 谢冰, E-mail: xiebing@pku.edu.cn

摘 要: 大语言模型在自动代码生成领域已展现出巨大的潜力, 但在实际应用中, 生成的代码常存在语法、语义、安全性、运行效率和可维护性等多方面的问题. 为解决这些挑战, 受约束代码生成技术应运而生. 此技术借鉴了受约束文本生成的方法, 通过在代码生成的各个阶段引入严格的约束, 确保生成代码能够满足预期的要求. 首先回顾大语言模型在代码生成过程中所暴露的主要问题, 详细分析代码正确性与代码质量方面的缺陷. 接着, 总结当前受约束代码生成技术的研究进展, 深入探讨不同方法的优势与局限. 之后, 讨论评估方法, 包括基准数据集的构建和评价指标的设计, 为后续研究的实验方案提供有价值的参考. 最后, 展望受约束代码生成技术面临的研究挑战和未来发展趋势.

关键词: 大语言模型; 代码生成; 代码质量; 受约束生成; 评估方法

中图法分类号: TP311

中文引用格式: 李宇轩, 邹艳珍, 王玥, 谢冰. 基于大语言模型的受约束代码生成研究进展. 软件学报. http://www.jos.org.cn/1000-9825/7613.htm

英文引用格式: Li YX, Zou YZ, Wang Y, Xie B. Research Progress on Constrained Code Generation in Large Language Models. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7613.htm

Research Progress on Constrained Code Generation in Large Language Models

LI Yu-Xuan[1,2], ZOU Yan-Zhen[1,2], WANG Yue[1,2], XIE Bing[1,2]

[1](Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)

[2](School of Computer Science, Peking University, Beijing 100871, China)

Abstract: Large language models (LLMs) demonstrate significant potential in automatic code generation. However, in practical applications, the generated code often suffers from multiple issues, including syntax errors, semantics inconsistencies, security inconsistencies, inefficient runtime performance, and poor maintainability. To address these challenges, constrained code generation techniques are introduced. Drawing inspiration from constrained text generation, these techniques impose explicit constraints at various stages of the code generation process to ensure that the generated code satisfies predefined requirements. This study first reviews the major issues exposed by LLMs in code generation, with a detailed analysis of deficiencies related to code correctness and quality. Subsequently, recent research progress in constrained code generation is summarized, and the strengths and limitations of existing approaches are systematically examined. Furthermore, evaluation methods are discussed, including the construction of benchmark datasets and the design of evaluation metrics, providing valuable references for experimental settings in future research. Finally, the research challenges faced by constrained code generation and its future development trends are outlined.

Key words: large language model (LLM); code generation; code quality; constrained generation; evaluation method

---

* 基金项目: 国家重点研发计划 (2023YFB4503803)
收稿时间: 2025-05-30; 修改时间: 2025-08-13; 采用时间: 2025-12-09; jos 在线出版时间: 2026-04-01

1 研究背景

1.1 大语言模型代码生成

随着大语言模型 (large language model, LLM) 在自然语言处理中取得突破性成果, 其在软件开发和代码生成任务中的应用也日益广泛. 基于大语言模型的代码生成指的是在给定不完整、不完美或不同语言的源代码的情况下, 使用大语言模型来生成指定目标语言的代码, 或根据自然语言描述生成目标代码的问题. 研究表明, 在 HumanEval[1]等一些公开的数据集上, 先进的大语言模型在一些简单代码生成任务上的准确率已经达到较高水平[2,3]. 除了基础的代码生成任务, 大语言模型还在代码补全[4,5]、代码翻译[6]、代码修复[7,8]和自动化测试用例生成[9]等衍生任务中展现出显著优势. 软件行业中已经有许多基于大语言模型技术的智能化工具 (如 GitHub Copilot[10]) 被广泛应用于协助开发者来完成实际开发任务.

然而, 目前大语言模型生成的代码往往存在语法、语义和安全性等问题. Wang 等人[11]对 6 种具有代表性的大语言模型进行了实证分析, 发现不同规模的大语言模型在生成代码时普遍存在语法和语义错误, 且生成的错误代码与正确代码之间存在明显偏差. Zhong 等人[12]的研究发现, 即使是能力较强的 GPT-4[13], 也有 62% 的生成代码存在 API (application program interface) 误用问题. Liu 等人[14]通过扩充 HumanEval 数据集的测试用例, 构建了更严格的代码生成基准. 在该基准上对 26 个模型进行实验的结果显示, 大语言模型生成的代码存在很多不易察觉的潜在缺陷, 如边界条件处理不当和复杂功能逻辑错误. 此外, Licorish 等人[15]的研究通过将模型生成代码和人类编写代码进行对比, 发现大语言模型生成的代码容易继承训练数据中存在的安全漏洞, 且在代码风格和编码规范上也常出现问题. Fu 等人[16]则通过分析基于软件常见弱点枚举 (common weakness enumeration, CWE)[17]的数据集, 揭示了现有大语言模型在生成代码安全性方面的缺陷.

1.2 受约束代码生成

为了解决大语言模型代码生成中存在的问题, 许多研究选择在代码生成过程中引入多种约束条件, 以提升生成代码的正确性和质量. 本文将这些工作总结为受约束代码生成技术 (constrained code generation). 这一术语借鉴了受约束文本生成技术 (constrained text generation)[18,19]和受控文本生成技术 (controlled text generation)[20,21], 这两类研究探讨了在生成自然语言文本时, 通过引入特定约束让模型生成具有预期属性的文本. 这类技术早期主要应用于机器翻译[22,23]等任务, 其代表性方法是约束解码 (constrained decoding)[23].

在本文中, 受约束代码生成的概念是在受约束文本生成的基础上衍生而来, 专注于代码生成领域. 与自然语言文本生成不同, 代码生成不仅需要满足语义上的合理性, 还必须严格遵循编程语言的语法规范, 以及由程序运行时行为衍生出的其他要求. 现有的许多代码生成研究已经注意到这些问题, 并应用了受约束文本生成的思路进行解决, 然而这些研究在命名上往往基于具体的约束类型, 如“安全代码生成”或“可靠代码生成”等. 本文则将其统称为“受约束的代码生成”. 此外, 本文还将一些使用提示词工程和后处理等技术优化大语言模型代码生成的研究纳入受约束代码生成的范畴. 原因在于这些技术本质上通过在代码生成的不同阶段施加约束, 确保生成结果符合特定的要求, 与狭义上的受约束代码生成方法具有相似的目标和作用.

我们对收集到的受约束代码生成相关论文进行阅读, 建立了一个考察受约束代码生成的整体性视角. 我们把相关研究分为 3 个主要方向: 代码生成中存在的问题和约束需求、受约束代码生成的实现方法和受约束代码生成的评估方法. 第 1 个研究方向重点讨论大语言模型代码生成中存在的问题, 并根据这些问题引出相应的约束需求. 后两个研究方向则聚焦于具体的约束生成方法和评估工具.

本文第 2 节介绍文献收集流程并分析文献情况. 第 3 节介绍代码生成中存在的问题与相应的约束需求. 第 4 节和第 5 节分别综述受约束代码生成的实现方法和评估方法, 并对其展开分析对比. 第 6 节在以上调研的基础上总结受约束代码生成的研究挑战并展望未来研究方向. 第 7 节对全文进行总结.

2 文献收集与分析

为了从整体性视角构建受约束代码生成的研究框架, 我们基于一组人工挑选的关键词对文献数据库进行了检

索, 并对检索得到的论文进行了分析. 关注的研究主要包括以下 3 个方向.

(1) 大语言模型生成代码中存在问题的分析. 这一方向的研究揭示了生成过程中的问题和缺陷, 并引出了对代码生成过程施加约束的需求, 提供了受约束代码生成的研究背景.

(2) 受约束代码生成技术. 这一方向涵盖了通过对模型生成过程施加约束来优化生成结果的各种方法.

(3) 受约束代码生成技术的评估方法. 这一方向主要研究如何评估并判断受约束代码生成的效果, 判断其是否成功达到了约束目标并提升了生成代码在某些方面的表现.

在文献收集过程中, 首先使用“大语言模型 (large language model, LLM)”和“代码生成 (code generation)”等关键词, 检索得到一组初始论文, 作为后续检索的基础. 然后, 根据不同的研究方向进一步筛选相关文献.

(1) 对于第 1 个研究方向, 使用“错误 (error)”“漏洞 (vulnerability)”“幻觉 (hallucination)”等关键词, 筛选出 19 篇相关性较高的论文.

(2) 对于第 2 个研究方向, 使用“受约束 (constrained)”“受控 (controlled)”“安全 (secure)”“可靠 (reliable)”等关键词, 筛选出 42 篇相关性较高的论文.

(3) 对于第 3 个研究方向, 使用“基准测试 (benchmark)”“数据集 (dataset)”“评估 (evaluating)”等关键词, 筛选出 18 篇相关性较高的论文.

由于第 1 个研究方向与第 3 个研究方向存在一定程度的重叠, 即部分研究在分析生成代码缺陷的同时设计了相应的数据集和评价指标, 因此这两个方向筛选出的论文存在重复. 去除重复后, 在 3 个方向上总计选取了 75 篇论文进行综述. 文献搜集的截止时间是 2025 年 8 月.

我们在保证主题相关性的前提下优先选择了发表年份较晚的新论文, 图 1 展示了这 75 篇论文的发表年份分布. 除了一篇在 2017 年发表的早期代表性研究外, 所有关注的论文均为最近 4 年内的文献, 且大多数发表于 2024 年之后, 具有较好的时效性. 从文献数量的变化趋势来看, 受约束生成代码相关研究的发表量在近几年逐渐增加, 研究热度在 2024 年迅速增长, 当前仍受到持续关注.

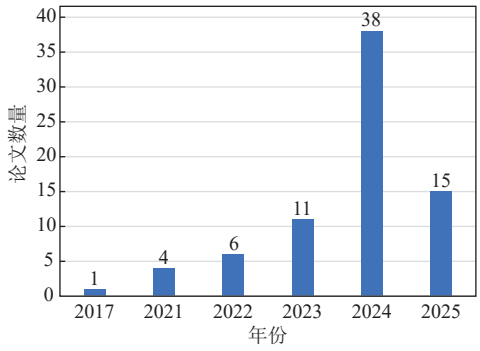

图 1 受约束代码生成相关研究论文的发表年份分布

基于我们选取的 3 个研究方向, 进一步总结了如后文图 2 所示的受约束代码生成研究框架. 对于受约束代码生成的研究工作, 首先需要明确其引入约束是为了解决代码生成中的哪些问题, 即确定约束需求; 其次, 根据约束施加的具体阶段, 将受约束代码生成技术分为 3 类: 预生成约束、生成中约束和后生成约束, 并在此基础上进一步确定其主要技术路径和具体研究内容; 最后, 从基准数据集和评价指标两个方面明确其使用的评估方法.

3 问题与约束需求

研究受约束代码生成技术的首要步骤在于明确当前代码生成中存在的问题, 并深入分析解决这些问题所需的约束需求. 本节将从代码正确性和代码质量两个层面, 系统梳理现阶段代码生成中存在的问题, 并探讨相应的约束需求, 为后续两节的综述提供必要的背景和理论基础.

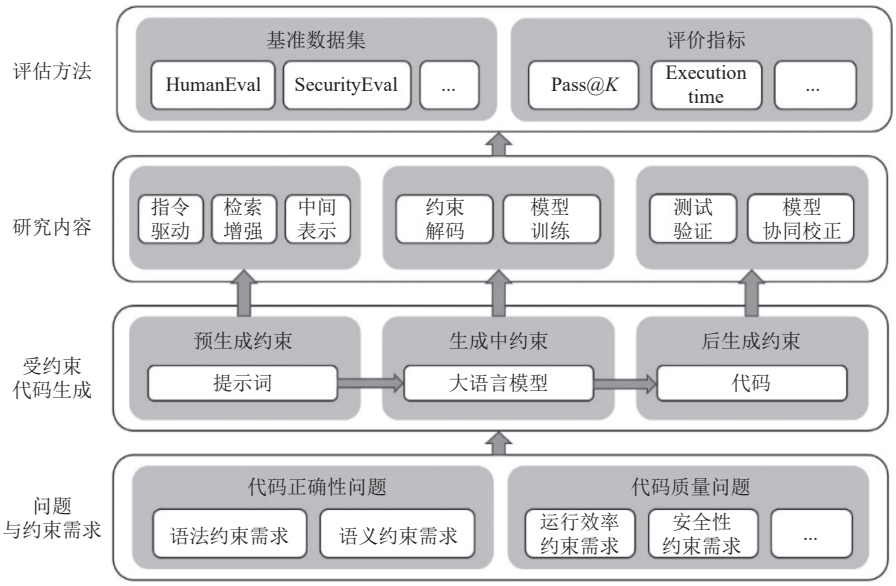

图 2 受约束的代码生成研究框架

3.1 关注代码正确性问题的约束需求

受约束代码生成的基础是通用的代码生成技术, 因此在约束生成过程之前首先需要生成功能正确的可运行代码. 大语言模型在代码生成任务上已取得显著进展, 但在许多实际应用场景中, 生成的代码仍存在正确性问题, 其中不仅包括表面的语法错误, 还涉及更深层的功能实现和逻辑流程中的语义偏差. 解决这些错误是代码生成研究普遍关注的问题, 但在此基础上, 受约束代码生成主要关注一些对代码正确性要求较高的领域和任务, 因为它们可能需要引入对代码语法或语义的严格约束. 接下来, 我们将从代码语法正确性和代码语义正确性两个角度, 详细探讨大语言模型代码生成中存在的正确性问题.

3.1.1 代码语法正确性

代码语法正确性主要衡量生成的代码是否在形式上符合编程语言的语法规则. 这一指标主要关注代码结构、关键词使用、标点符号及其他语言规范是否正确, 从而确保代码能够顺利通过编译或解释执行. 大语言模型生成的代码有时会因为存在语法错误而无法运行, 或是因为缺少特定的语法结构而无法应用到上下文环境中. Yetiştiren 等人[24]对 GitHub Copilot 等智能化代码生成工具进行的实证研究发现, 这些工具生成的代码中有大约 10% 是无法运行的无效代码, 而语法错误是导致无效代码产生的主要原因之一. Liu 等人[25]在研究代码幻觉 (code hallucination) 的实验中统计了无幻觉但有错误的代码的比例, 发现近 1/5 出错代码中的错误不是代码幻觉引起的, 而是由包括简单语法错误在内的其他因素引起的.

几乎所有代码生成研究都会关注到语法问题, 但在某些情况下语法正确性会显得格外重要, 一个代表性场景是针对复杂编程语言的代码生成. 具体而言, 使用以 Python 为代表的高级语言生成程序时, 由于其语法简单且存在诸多语法糖 (syntactic sugar), 保证程序的语法正确性对模型来说并不算很大的挑战. 相比之下, 使用中低级语言生成程序往往更容易出现语法错误. Kharma 等人[26]在一个包含 200 项任务的数据集上对大语言模型生成的代码进行了多语言、多模型的分析, 发现模型的生成效果因编程语言而异, 在 C++语言的任务中多数模型都经常产生不可编译的代码, 这是由于 C++语言语法复杂且存在多种不同标准, 影响了模型对正确语法的理解. 另一个对语法正确性要求高的常见场景是生成完整程序, 因为相比代码片段或独立函数, 完整程序对语法结构的要求更严格. Tosi[27]在 3 个复杂的编程考试题目上测试了 3 个大语言模型 (ChatGPT[28,29]、GPT-4 和 Google Bard[29]) 的代码生成效果, 发现 3 个模型均存在生成无法运行代码的情况, 其主要原因为缺少必要的语法构件, 如 main 方法. 其中

Google Bard 模型在给出进一步提示的情况下仍然无法生成包含正确语法结构的代码, 体现出模型生成的代码在语法完整性上存在不足.

现有代码生成研究中对语法正确性的优化往往不足以支持上述场景的需求, 因此一些工作考虑在模型生成过程中引入严格语法约束, 包括但不限于以下几种.

• 完善语法信息: 在输入中提供完整的语法结构或使用结构化模板, 引导模型生成符合语言规范的代码.

• 语法规则约束: 设定严格的语法规则来约束模型生成过程, 例如强制检查关键字、标点和代码块边界, 确保生成代码能顺利通过编译器或解释器的检查.

• 后置语法校验: 在生成后进行自动化语法检测, 并反馈给模型或开发者, 确保输出代码的形式完全符合编程语言标准.

3.1.2 代码语义正确性

代码语义正确性主要衡量生成的代码在逻辑和功能实现上是否与设计者的意图一致. 该指标不仅关注代码能否顺利运行, 更强调代码在实际执行过程中是否实现了预期的功能、正确处理数据以及合理控制流程, 是评价一个程序是否正确的关键标准. 大语言模型生成代码时, 可能因未能正确理解需求而生成功能错误的代码, 或是产生潜在的逻辑漏洞、算法缺陷和边界条件处理不当等问题, 这些问题往往不会在语法检查中暴露. Wang 等人[11]从错误原因和错误位置两个维度出发, 在 HumanEval 数据集上对 6 种代表性大语言模型生成的代码中存在的问题进行了深入分析, 结果显示 6 种模型都会出现语义正确性问题, 主要表现为两类: 代码明显偏离预期的任务逻辑或预期结果, 以及在条件判断语句和循环语句中使用错误的条件. 该研究进一步分析发现这些语义问题往往涉及多行代码, 需要付出大量努力来修复. Troshin 等人[30]采用一系列探测任务评估预训练模型对代码各类属性的理解能力, 包括对代码抽象语法树深度、变量声明情况和变量误用情况等属性的预测. 实验结果表明, 模型在捕捉基础语法信息方面表现良好, 但在处理代码语义 (如变量作用域、数据流和算法等价性) 任务上与简单基线相比存在不足, 这可能影响模型在代码生成任务中的表现. Liu 等人[25]的研究系统性地探讨了大语言模型在代码生成过程中产生的代码幻觉 (code hallucination), 并将其分为 5 个大类. 统计分析表明, 最常见也最重要的幻觉类型是意图冲突 (intent conflicting), 即生成代码与用户意图之间的语义相关性较小. 意图冲突还可以进一步分为两类: 整体语义冲突, 即代码片段的整体功能与任务描述有很大不同或没有明确的整体功能; 以及局部语义冲突, 即代码中部分语句的功能与用户要求相矛盾. 总体而言, 大多数意图冲突幻觉都会导致语义错误并产生错误的输出.

语义正确性是代码生成研究通用的核心指标, 但一些开发场景不仅要求模型生成功能正确的程序, 还会对功能语义提出更高的要求. 对于领域特定代码生成任务, 通常要求模型必须使用该领域的方法和工具完成功能. Gu 等人[31]对大语言模型在领域特定代码生成任务中的表现进行了评估, 结果表明当任务依赖于特定领域库 (如 Web 开发或游戏开发中的第三方库) 时, 生成代码的语义正确性会显著下降, 主要表现为 API 误用和 API 遗漏问题. Lian 等人[32]在多个广泛使用的数据集上对 3 种先进模型进行了细致分析, 归纳出大语言模型代码生成的 8 种不同类型的弱点, 其中多数弱点 (如不准确的提示、缺失关键语义信息、错误的 API 调用和领域知识不足) 都可能导致生成代码与设计意图严重偏离, 产生语义正确性问题. 另一种常见情况是需要生成具有复杂结构代码的任务, 这类任务要求模型在实现正确功能的基础上, 确保代码的各个部分之间具有正确的依赖关系. Du 等人[33]研究了大语言模型在较为复杂的类级代码生成任务上的效果, 实验结果显示模型在生成具有真实类结构和内在依赖关系的代码时语义正确性显著降低, 其生成的代码往往包含大量属性错误和类型错误.

现有研究往往基于测试用例评价和优化代码语义正确性, 这一方法难以应对上述额外语义要求. 因此, 需要考虑对模型施加严格语义约束, 包括但不限于以下几种.

• 上下文与意图一致性: 强化模型对任务描述和上下文的理解, 确保生成代码的逻辑流和功能实现与设计意图保持高度的一致性.

• 语义校验机制: 引入机制对关键逻辑、条件判断和循环语句进行检查, 验证代码在处理数据和控制流程时是否满足预期要求.

• 领域知识融入: 通过提供领域相关的提示或示例, 确保生成代码能够正确调用 API 和使用特定库, 避免因理

解不当而引发语义错误.

3.2 关注代码质量问题的约束需求

在实际应用中, 除了保证代码的正确性, 代码质量也是衡量生成代码是否适用于生产环境的重要标准. 代码质量不仅涉及代码本身的高效运行, 还包括可维护性、可读性和安全性等多个维度. 大语言模型生成的代码往往在这些方面存在不足, 因此在受约束代码生成中引入相应的质量约束是十分必要的. 接下来, 我们将从代码可维护性、可读性、安全性和运行效率这 4 个维度, 详细阐述代码质量问题及其约束需求.

3.2.1 代码可维护性

代码可维护性是指生成的代码在后期调试、扩展和修改时的难易程度. 可维护性高的代码应具备良好的模块化结构、清晰的函数划分以及充分的注释和文档说明. 然而, 当前大语言模型生成的代码往往缺乏统一的编码规范, 容易导致逻辑耦合度高、代码冗长或难以理解, 从而增加后续维护成本. Tosi[27]在编程考试题目上的实验使用 SonarQube[34]等工具测量了生成代码的圈复杂度 (cyclomatic complexity)[35]和认知复杂度 (cognitive complexity)[36]指标, 发现生成的代码虽然能够通过测试用例, 但复杂度指标较高, 这意味着在维护和扩展时可能需要更多人工干预. Yetiştiren 等人[24]对 GitHub Copilot 等智能化代码生成工具进行的实证研究发现, 大量生成结果中都带有一些代码异味 (code smell), 严重程度各不相同. 3 个最常见的问题是: 函数命名不当、变量命名不当以及认知复杂度高. 这些代码异味会降低代码可维护性, 带来技术债务 (technical debt)[37].

现有研究显示大语言模型生成代码存在可能影响后期调试和扩展的可扩展性缺陷, 为此需要施加可维护性约束, 包括但不限于如下几种.

• 模块化设计: 要求生成代码具备清晰的模块划分和层次结构, 降低各部分之间的耦合度, 便于局部修改和整体重构.

• 一致性规范: 要求生成代码遵循统一的编码规范 (如函数划分、变量作用域和异常处理), 确保代码设计具有良好的结构性.

• 系统集成友好: 生成的代码应易于与现有系统整合, 支持持续演进, 减少因接口或结构不统一带来的维护成本.

3.2.2 代码可读性

代码可读性是指代码易于被人类理解的程度, 是协同开发和代码审查的重要基础. 大语言模型生成的代码有时会存在缩进不规范、格式不一致或缺乏必要注释等问题, 降低了代码的直观性和可理解性. Troshin 等人[30]通过构建探测任务对大语言模型生成代码的可读性进行了评估, 其数据基础是 Scalabrino 等人[38]建立的人工标注可读性分数的代码数据集. 实验结果显示, 模型在代码可读性任务上的表现非常接近基于词袋模型的简单基线, 且基于代码数据预训练的模型表现不如基于文本数据预训练的模型. Siddiq 等人[39]的研究关注了模型训练集中的代码异味泄漏到模型输出中的情况, 通过使用 Pylint[40]对数据集代码和模型生成代码进行静态分析, 发现其中大量存在行长度过长、重复代码与未使用变量等非安全性代码异味, 这些代码异味不会对安全性造成直接影响, 但会降低代码的可读性.

现有研究显示大语言模型生成代码存在可读性问题, 其生成结果对开发者而言往往难以理解和审查, 因此需要引入可读性约束, 包括但不限于如下几种.

• 格式规范化: 要求统一的缩进、空格和换行格式, 使代码视觉上整洁, 易于阅读.

• 清晰命名: 制定严格的命名规则, 确保变量、函数和类名能直观表达其含义.

• 必要注释补充: 在关键代码段添加清晰注释和文档说明, 使代码逻辑易于跟踪和理解, 方便代码审查和协同开发.

3.2.3 代码安全性

代码安全性直接关系到软件系统的整体可靠性和数据安全. 大语言模型在生成代码时, 可能会忽视安全最佳实践, 从而引入各类安全漏洞. Kharma 等人[26]对大语言模型代码生成的多语言、多模型分析涵盖了安全性指标, 通过使用静态安全分析工具检测代码中的漏洞, 发现模型在生成某些编程语言的代码时存在普遍性的安全漏洞,

例如在生成 Java 代码时多数模型未能利用最新的安全特性. Khoury 等人[41]基于 ChatGPT 进行了跨多种编程语言的代码生成实验, 其结果显示尽管 ChatGPT 能够在用户提示下识别并解释代码中的安全漏洞, 但其生成的初始代码普遍存在安全隐患, 而且在尝试生成更安全的代码时经常无法提出足够的改进措施. Zhong 等人[12]从 StackOverflow[42]收集了 1 208 个围绕 Java API 的编程问题, 并通过静态分析方法检测大语言模型为这些问题生成代码时的 API 误用情况. 结果显示, 即便是先进的模型在代码生成时也存在高比例的 API 误用问题, 生成的代码即使在功能上通过了测试, 仍可能存在潜在的安全隐患和鲁棒性缺陷, 最终导致内存泄漏、程序崩溃和资源泄露等严重问题. Siddiq 等人[39]关于训练集中代码异味泄漏到模型输出的研究使用 Bandit[43]分别检测了一个开源大语言模型和一个闭源大语言模型的代码生成结果, 发现这些模型生成的代码都包含多种可能导致安全漏洞的代码模式, 如使用断言和弱哈希函数. Dai 等人[44]的研究使用了 3 种流行的静态分析器和 2 种大语言模型来识别生成代码中的潜在漏洞, 实验结果显示大语言模型输出的代码不仅存在多种安全漏洞, 其中许多漏洞还无法被常用的静态分析工具 (如 CodeQL) 检测到. 进一步的研究发现, 现有用于增强代码安全性的方法往往会牺牲代码的功能性, 意味着目前的技术尚无法完全解决安全性问题. Schaad 等人[45]在 MITRE CWE 上对主流大语言模型进行了实验, 发现模型最初生成的代码中有 65% 会被安全工程师识别为不安全; 但在工程师的引导下进行多轮迭代后, 模型生成代码的安全性可以达到 94% 以上, 说明大语言模型在生成安全代码方面仍有较大的改进空间.

现有研究表明大语言模型生成代码时常忽视安全最佳实践, 导致 API 误用、异常处理不当等问题, 进而引发内存泄漏、程序崩溃等安全隐患. 为此, 需要对生成代码施加安全性约束, 包括但不限于如下几种.

• 安全编码规范: 要求生成代码遵循安全最佳实践, 如正确使用安全 API、避免硬编码敏感信息, 并确保异常处理完善.

• 静态安全检测: 在生成后采用静态安全分析工具检测潜在漏洞和 API 误用, 确保输出代码在安全性上达到生产标准.

• 风险防范机制: 引入防护措施, 要求生成代码对异常情况进行全面检查, 如防止资源泄露、内存泄漏等安全隐患.

3.2.4 代码运行效率

代码运行效率主要反映生成代码在实际执行过程中的资源消耗和响应速度. 在生成代码时, 大语言模型可能会因缺乏针对性优化而产生冗余计算、不必要的循环或低效的数据处理方式, 进而导致程序运行缓慢或资源占用过高. 这些效率问题在生产环境中可能造成性能瓶颈, 影响系统的整体运行效果. Huang 等人[46]的研究基于效率关键编码问题构建了用于评估代码运行效率的基准, 在该基准上对 42 个大语言模型生成代码能力的实证检验显示, 当前最强大的大语言模型 (如 GPT-4), 其生成代码的平均运行时间是人类编写的规范解决方案的 3.12 倍, 且在最极端的情况下的运行时间和总内存使用量可达规范解决方案的 13.89 倍和 43.92 倍, 表明大语言模型生成的代码往往效率较低. Sikand 等人[47]从软件可持续性视角出发, 调研了当前主流的智能化代码生成工具在生成代码时是否默认遵循绿色编码实践. 研究发现, 多数工具在默认情况下生成的代码存在明显的能耗高、资源利用率低等情况, 未能体现出节能降耗的设计模式, 存在运行效率方面的问题.

模型能生成正确的代码并不意味着模型也能生成高效的代码. Niu 等人[48]在 HumanEval、MBPP[49]和原创的 LeetCodeEval 这 3 个数据集上对一组大语言模型进行了代码生成实验, 使用 gem5 CPU 模拟器[50]计算其生成代码的运行时间以评估运行效率, 结果显示模型生成正确代码的能力与生成高效代码的能力不呈正相关, 且在生成代码的运行效率方面更大的参数量并不能保证更高的性能, 而是模型的训练策略和训练数据影响更大. Du 等人[51]使用以参考解决方案作为效率基线的数据集评估代码大语言模型的代码生成效率, 该研究引入了一个能同时反映代码功能正确性和运行效率的新度量 Beyond, 其实验结果显示模型在测试用例通过率上的表现显著高于在 Beyond 度量上的表现, 表明虽然模型在生成功能正确的代码方面表现出了令人印象深刻的能力, 但它们生成代码的运行效率仍然存在不足. Solovyeva 等人[52]在 Mac 和 PC 两个平台上针对 3 种编程语言 (Python、Java 和 C++) 分析了大语言模型生成代码的能效和性能, 使用 3 种前沿模型处理了 LeetCode 中的高难度编程问题. 实验结果显示, 大语言模型在生成 Java 和 C++代码时经常会输出低效的程序, 而在生成 Python 代码时则可能输出比人类解决方案

更高效的程序, 尽管从实践角度看 Python 代码对运行效率的要求通常较低.

现有研究指出大语言模型生成的代码在执行效率上存在冗余计算、低效循环和资源浪费等问题. 因此, 需要为模型生成的代码施加运行效率约束, 包括但不限于如下几种.

• 优化生成逻辑: 要求生成的代码在实现功能的同时, 尽可能减少冗余计算和不必要的循环, 采用更高效的数据处理方式.

• 性能评估机制: 在生成后对代码的运行时间和资源占用进行评估, 并与标准基线对比, 确保生成代码在实际运行中具备较高的效率.

• 资源使用限制: 设定明确的资源消耗指标 (如内存占用), 促使模型生成代码时兼顾功能与执行效率.

3.3 小 结

本节分别从代码正确性和代码质量两个角度, 详细梳理了现有大语言模型生成代码中存在的问题, 并分析了针对不同问题的约束需求. 相关研究的总结见表 1.

表 1 大语言模型代码生成问题总结

问题类型	实例	数据集	研究重点	约束需求
语法正确性	Kharma等人[26]	200 programming tasks manually defined	编程语言差异	严格语法约束
	Tosi[27]	three complex coding problems	语法完整性	
	Liu等人[25]	HALLUCODE, HumanEval	代码幻觉	
	Yetiştiren等人[24]	HumanEval	智能化代码生成工具	
语义正确性	Gu等人[31]	web and game development repositories	领域特定代码生成	严格语义约束
	Liu等人[25]	HALLUCODE, HumanEval	代码幻觉	
	Lian等人[32]	CoNaLa[53], HumanEval, DS-1000	代码生成弱点	
	Wang等人[11]	HumanEval	错误原因与错误位置分析	
	Troshin等人[30]	Ahmad等人[54]	模型诊断探测	
	Du等人[33]	ClassEval	类级代码生成	
运行效率	Niu等人[48]	HumanEval, MBPP, LeetCodeEval	代码运行效率	运行效率约束
	Du等人[51]	Mercury	现实效率基线	
	Solovyeva等人[52]	LeetCode	硬件效率指标综合对比	
	Huang等人[46]	EffiBench	效率关键编码问题	
	Sikand等人[47]	nano-dataset of prompts	绿色编程	
可维护性	Tosi[27]	three complex coding problems	圈复杂度和认知复杂度	可维护性约束
	Yetiştiren等人[24]	HumanEval	代码异味	
可读性	Troshin等人[30]	Scalabrino等人[38]	人工标注可读性分数	可读性约束
	Siddiq等人[39]	HumanEval	静态分析	
安全性	Kharma等人[26]	200 tasks manually defined	编程语言差异	安全性约束
	Khoury等人[41]	21 programs in 5 languages	模型自我审查	
	Zhong等人[12]	RobustAPI	现实编码问题	
	Siddiq等人[39]	HumanEval	代码安全气味	
	Dai等人[44]	BigCodeBench, SecCodePLT	安全性审查与功能审查	
	Schaad等人[45]	MITRE CWE	常见安全性漏洞	

从代码正确性角度来看, 问题主要体现在语法与语义两个方面. 首先, 生成代码因语法结构不全、关键词或标点使用不当等原因, 可能导致无法编译运行; 其次, 代码的语义可能存在偏差, 表现为代码逻辑与预期任务的偏离, 影响功能和数据处理的准确性. 针对这些问题, 约束措施包括完善语法信息、引入严格的语法规则以及通过上下文一致性与语义校验机制保障代码逻辑正确性等.

从代码质量角度出发, 现有问题涉及运行效率、可维护性、可读性以及安全性多个方面. 大语言模型生成的

代码在实现功能时, 常因冗余计算、缺乏模块化设计、格式不规范或安全编码不足, 影响其在实际生产环境中的适用性和持续演进能力. 因此, 需要在生成过程中引入优化生成逻辑、性能评估机制、统一编码规范及静态安全检测等多重约束手段, 以提升代码的整体质量.

总体而言, 尽管大语言模型在代码生成任务上已展现出较大潜力, 但生成代码在正确性和质量上仍面临诸多挑战. 通过对问题及约束需求的深入分析, 可以为后续约束代码生成技术的研究提供理论基础和实践指引.

4 受约束代码生成方法

本节将探讨如何在大语言模型的代码生成过程中引入约束机制, 以提升生成代码的正确性与质量. 根据约束措施在生成流程中的作用时机, 我们将受约束代码生成方法分为 3 类: 预生成约束方法、生成中约束方法和后生成约束方法, 即分别将约束机制施加于代码生成前、代码生成过程中和代码生成后的方法. 这 3 类方法相辅相成, 共同构建起完善的受约束代码生成体系. 接下来, 我们将依次介绍 3 类方法的相关研究.

4.1 预生成约束方法

预生成约束方法主要侧重于在代码生成任务开始前就对模型输入及预期输出设定约束. 这一阶段的关键目标是通过对输入数据、任务描述以及相关上下文的预处理, 明确代码结构与语义要求, 为后续生成过程提供坚实的基础. 预生成方法的优势之一在于通常不需要获取模型的内部表征或修改模型内部的生成机制, 因此可以应用在不开放内部细节的闭源大语言模型上. 根据具体技术路径, 现有的预生成约束方法可分为 3 类: 指令驱动方法、检索增强方法和中间表示方法.

4.1.1 指令驱动

指令驱动方法通过为大语言模型提供明确、详细的提示指令和任务描述, 使模型能够准确理解预期目标和结构要求. 该方法通常采用预定义的格式和模板对输入任务进行标准化描述, 或是基于分析工具对输入进行预处理, 从而在生成代码前就设定好明确的约束条件, 降低语法和语义偏差的可能性.

Pearce 等人[55]针对安全漏洞修复问题提出了一种零样本漏洞修复方法. 该方法通过设计详细的提示指令, 将漏洞信息 (如错误报告) 嵌入到提示中, 从而引导大语言模型生成既安全又功能正确的代码修复方案. Res 等人[56]进一步研究了提高生成代码安全性的提示词优化策略, 具体包括 4 种优化方法: (1) 提供场景特定信息: 基于专家知识提供对安全相关特性的要求和警告; (2) 迭代优化: 基于一个规则集迭代地修改提示; (3) 一般性对齐变换: 使用一般性语句在与模型对话开始时对齐设置, 例如在文件头部添加一个精心制作的注释; (4) 前 3 种方法的组合应用. 在 GitHub Copilot 上应用上述策略的实验证明, 该方法使安全代码的比例最多提高了 8%, 同时将不安全代码的比例降低了 16%, 为受约束代码生成提供了一种低成本、高效率的预生成约束手段.

Wu 等人[57]提出了一种针对少样本代码生成的提示选择和增强算法, 通过引入高质量的示例, 约束代码生成过程并提高结果的正确性. 具体来说, 该研究设计了一套指标体系, 在选定一组初始示例增强提示之后, 通过衡量这些示例的复杂性、语义相似度以及概念重叠度, 对示例集合进行筛选, 保留最具代表性的示例, 最终构造出更加精炼且准确的提示. 在数学推理和机器人控制任务中的实验结果显示, 该方法不仅显著提升了代码生成表现, 还大幅减少了提示词所需示例的数量.

Nazzal 等人[58]提出了 PromSec 方法, 该方法结合图生成对抗网络 (graph generative adversarial network, gGAN)[59]和对比学习策略对大语言模型生成代码的提示进行自动优化, 以减少生成代码中的安全漏洞. 其主要流程包括: 如果初始提示词中包含代码, 则通过静态安全分析工具检测代码中的常见弱点枚举 (CWE); 若检测出安全漏洞, 则将代码转换为图表示, 再由 gGAN 对图进行修复生成安全性更高的版本, 随后通过逆向操作生成改进后的提示, 并将其反馈给 LLM 进行代码生成. 如果初始提示词中不包含代码, 或是改进后的提示仍然未能生成足够安全的代码, 方法还可以通过一个交互式循环系统, 不断重复上述优化过程直至代码中的漏洞数量降至预设阈值. 实验结果表明, PromSec 通过优化后的提示词对大语言模型代码生成过程施加了有效的安全性约束, 显著减少了后续对代码进行安全分析的成本.

Tihanyi 等人[60]提出了 ESBMC-AI 框架, 该方法利用形式化验证技术为大语言模型提供增强的指示, 约束模型为软件漏洞修复任务生成的代码. 其核心思想是: 首先利用有界模型检测 (bounded model checking, BMC)[61]来检测输入源代码中的漏洞并生成精确的反例 (包括堆栈跟踪、错误行号及错误类型), 然后将这些漏洞信息与原始代码一同作为提示输入给大语言模型, 指导模型生成修复代码. 方法还提供了迭代校正, 即修复后的代码再次通过 BMC 进行验证, 确保修复效果. 该方法在缓冲区溢出、算术溢出及空指针解引用等常见漏洞的修复上取得了较高准确率.

Shi 等人[62]提出的 Progent 方法通过限制大语言模型的工具调用能力实现了一种特殊的指令驱动约束. Progent 包含一种专用的权限控制语言, 能够细粒度地描述对模型工具调用的控制策略. 方法基于该控制策略在预生成阶段限制模型的行为, 减少错误的工具调用带来的负面影响, 提升模型最终输出结果的安全性和可用性, 同时还避免了可能在工具调用操作中存在的安全隐患. Progent 使用的控制策略可以动态更新, 并能够基于大语言模型自动生成. 实验结果表明, 使用了 Progent 的大语言模型在多种外部攻击场景下能够保持较高的任务完成率, 有效提升了复杂环境中模型生成代码的安全性.

4.1.2 检索增强

检索增强方法利用现有的代码库、文档资源或示例代码, 在生成前向模型提供领域内的背景知识和相关参考. 通过检索相关信息并将其融入输入中, 模型可以借鉴这些优质资源, 从而更好地理解任务需求和编程语言规范, 提高生成代码的准确性和实用性.

Parvez 等人[63]的研究是将检索增强技术用于约束大语言模型代码生成的代表性工作. 该研究提出了一种检索增强框架 REDCODER, 主要面向代码生成和代码摘要任务. 该框架利用检索器模块从大规模代码或文档数据库中获取与输入相关的候选项, 然后将这些候选项作为辅助信息补充给生成器. REDCODER 的检索器基于扩展的密集检索技术, 并且可以处理由单峰实例 (仅代码或自然语言描述) 和双峰实例 (代码描述对) 组成的检索数据库. 实验结果表明, REDCODER 在 Java 和 Python 代码生成以及代码摘要任务上均取得了明显优于传统生成模型的效果. Chen 等人[64]提出的 CodeRetriever 调整了检索方法, 对大规模代码库建立向量索引, 使得模型能够高效检索到与当前上下文相关的代码片段, 并能将模型的概率分布与检索到的候选代码进行融合. CodeRetriever 使用代码上下文与目标 API 的配对数据对模型进行了训练, 使其能够学习到如何在不同情况下权衡模型预测结果和检索增强内容. Lu 等人[65]提出了一种检索增强的代码补全框架 ReACC. 该方法使用一个双编码器架构的检索器, 为代码补全任务中的输入代码执行检索, 从大规模代码库中寻找与之语义和词汇相似的代码片段. ReACC 采用混合检索方法 (结合密集检索和稀疏检索), 并针对从部分代码到完整代码的搜索场景进行了专门设计, 有效提升代码补全任务中生成代码的准确性和健壮性. 不同于 ReACC 主要检索相似代码, Zhou 等人[66]提出的 DocPrompting 方法受启发于人类程序员使用未见过的函数和库时参考代码手册和文档的行为, 通过检索相关的代码文档来补充自然语言提示中的信息. 该方法从文档库中获取与输入意图匹配的文档片段, 再将这些文档与原始提示结合, 提高了多种基础模型在处理新函数和新库调用时的生成代码的准确性.

检索增强技术已经较为成熟, 目前被应用在工业控制等领域协助大语言模型开展工作. 例如, Koziolek 等人[67]提出了一种针对工业自动化中控制代码的代码生成方法. 该方法首先利用文本嵌入技术将专有功能块的规格文档存储到向量库中, 然后在生成控制逻辑代码时, 通过相似度搜索将相关的功能块信息融入生成提示中, 从而指导大语言模型输出符合编码标准的结构化文本代码. 实验结果表明, 该方法能够有效集成预先验证的功能块, 约束生成代码的语法结构并提高其语义正确性.

4.1.3 中间表示

中间表示方法旨在将复杂的代码生成任务转化为一种抽象的、中间层次的结构化表达. 先生成中间表示, 再由此导出最终代码, 该方法有助于将任务需求分解为若干较简单的子任务, 从而在生成初期捕捉整体逻辑和结构, 减少语义偏差和生成错误.

Chen 等人[68]提出了一种 Sketch then generate 方法, 该方法通过从用户提供的自然语言提示词中抽取语言结

构信息, 自动生成代码草图, 从而提供即时且增量的反馈, 帮助用户完善提示词并指导大语言模型生成代码. 其主要流程包括: (1) 映射: 利用依存句法分析和词性标注, 将提示词中的代码相关短语映射到预定义的代码元素; (2) 组装: 基于规则匹配和抽象语法树 (abstract syntax tree, AST) 构建, 尝试对映射得到的代码元素进行合理组装, 可能的组装方案将作为建议被呈现给用户; (3) 保留: 当用户接受建议后, 系统完成组装并保存结果. 用户最终将自然语言提示词和组装完成的代码草图一并提供给大语言模型作为输入, 代码草图的存在可以约束模型生成代码的语法结构并降低理解用户意图的难度, 方法以这种方式施加预生成约束, 提高生成代码的正确性.

Ma 等人[69]提出了一种基于语义思维链 (SeCoT) 提示的代码生成方法. 该方法为输入的提示词生成数据流和控制流等关键语义信息, 通过这些中间表示约束生成过程, 使得大语言模型能够根据原始输入和关键语义信息生成更准确的代码. 具体而言, SeCoT 利用少量示例进行上下文学习, 使模型在生成过程中自动生成包含数据流和控制流分析的中间推理步骤, 进而生成中间表示. 实验结果表明, 该方法不仅显著提升了闭源模型和开源模型的代码生成准确率, 而且在 HumanEval 和 MBPP 等多个基准上均取得了优于传统提示技术的表现.

4.2 生成中约束方法

生成中约束方法是指在大语言模型实际生成代码时, 通过约束机制对输出进行调控. 这一阶段关注的是即时纠正生成过程中的错误和偏差, 确保代码在逐步生成的过程中逐步满足预定约束. 根据具体技术路径, 现有的预约束生成方法可分为 2 类: 约束解码方法和模型训练方法.

4.2.1 约束解码

约束解码方法在代码生成的解码阶段嵌入实时校验机制, 对生成的每一步输出进行检查和纠正. 通过检测语法结构、标点和逻辑条件, 约束解码可以在生成过程中及时抑制不符合规则的输出, 从而使最终生成的代码更加准确和连贯. 图 3 展示了约束解码方法的基本流程示意图. 可以看到, 约束解码方法通常使用一个补全引擎 (completion engine, CE) 约束模型的解码过程, 该引擎可以根据预设的上下文无关语法 (通常为具体的约束内容) 筛选模型每一轮解码的候选令牌, 保留符合要求的令牌组成输出序列, 使得最终输出也满足上下文无关语法所描述的约束.

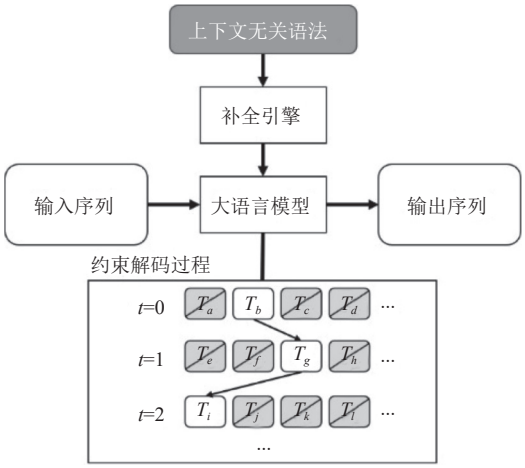

图 3 约束解码方法基本流程示意图

约束解码技术早期主要被应用在以神经机器翻译 (neural machine translation, NMT)[70]为代表的自然语言处理任务上. Hokamp 等人[23]提出了一种名为网格束搜索 (grid beam search, GBS) 的算法, 通过扩展传统的束搜索 (beam search)[71], 使其能够在解码过程中加入预设的词汇约束. 具体来说, GBS 方法能够在不修改模型参数或训练数据的前提下, 强制模型在生成序列时包含特定的词汇或短语, 从而将外部知识 (如用户修正或领域术语) 整合到序列生成模型中. 在神经机器翻译实验中, GBS 显著提升了交互式翻译和领域自适应任务的性能, 尤其是在使用专有术语和术语映射时, 表现出比传统方法更高的翻译质量. 研究探讨了该方法对于许多序列生成任务都具有广泛的适用性.

Scholak 等人[72]提出了 PICARD 方法, 该方法通过增量解析技术对大语言模型的自回归解码过程进行约束, 是大语言模型约束解码的早期代表性工作之一. PICARD 通过在每一步解码中对模型输出进行有效性检查, 逐步排除不符合约束条件的候选令牌, 从而避免生成无效的输出. 具体约束条件包括基础的排除低概率令牌, 以及 3 种可选的进阶约束模式: 词法检查、无保卫的句法解析和有保卫的句法解析. 该方法的特点在于对模型侵入性较小, 能兼容多数通用模型. 应用了 PICARD 的大语言模型在文本到 SQL 的生成任务中确保了无效的输出几乎不会出现. Ugare 等人[73]提出的 SynCode 框架则在 PICARD 方法思路的基础上使用一种新的数据结构, 确定有限自动机掩码存储 (DFA mask store), 实现约束解码技术. SynCode 在解码过程中解析部分输出, 获得可接受的终结符序列和余部, 再利用掩码存储中离线预计算的语法掩码对每一步生成的令牌进行约束, 过滤掉不符合语法规则的候选令牌. 实验结果表明, 该方法显著减少了多种代码生成任务中的语法错误.

Poesia 等人[74]提出的 Synchromesh 框架设计了更复杂的约束机制, 其核心是一个约束语义解码模块 (constrained semantic decoding, CSD). 该模块使用的补全引擎能够基于一个不完整的程序片段返回所有可以将该片段扩展为一个完整正确程序的令牌. 补全引擎的具体设计包含两层: 检验语法有效性的上下文无关层, 以及基于语言语义和用户上下文内容编码语义约束的上下文敏感层. Synchromesh 框架提供了一种直接从目标语言语法构建补全引擎上下文无关层的自动化方法. 此外, Synchromesh 还结合了目标相似度调整技术 (target similarity tuning, TST), 从一个数据集中检索与当前任务具有相似意图的示例用于提示词增强. 该研究在基于 3 种现实语言 (SQL、Vega-Lite 和 SMCalFlow) 的代码生成任务上进行了实验, 结果显示 Synchromesh 能够提升模型的代码生成准确率并有效防止代码中运行时错误的产生.

一些研究者通过将约束解码方法与其他技术结合来提升其效果. Zhang 等人[75]提出了一种融合检索增强技术的特殊约束解码方法 kNN-TRANX, 能够基于 Seq2Tree 模型[76]将编码器得到的内部语义表征转换为一棵抽象语法树的构建操作序列. 由于构建抽象语法树的每一步操作都受到语法规则约束 (如 ASDL 规则), 可以确保最终生成的抽象语法树以及由其转换而来的输出代码符合语法规范. 在此基础上, kNN-TRANX 构建了包含语法和语义信息的外部数据库, 能够在上述生成过程中基于最邻近检索机制寻找与当前上下文相关的令牌, 并最终通过置信度网络将检索得到的外部信息融入模型的生成过程中. Park 等人[77]提出了语法对齐解码方法, 旨在解决现有语法约束解码技术在强制模型生成满足给定语法的代码时所带来的问题. 作者提出了自适应采样与近似预期未来 (ASAp) 算法, 通过迭代地改进生成的序列的未来语法性预测, 从而在保证语法约束的同时, 使生成结果更符合模型的原始概率分布. 实验证明, ASAp 在代码生成任务中有效缓解了约束解码技术对模型生成效果的负面影响.

随着约束解码方法的发展, 相关技术被应用于各种下游任务上. Ilager 等人[78]提出了 GREEN-CODE 框架, 实现了一种特殊的关注模型能耗的约束解码, 通过引入动态提前退出机制约束大语言模型的解码过程. 在代码生成任务中, GREEN-CODE 将提前退出机制构建为强化学习决策问题并训练代理, 在保证生成代码的准确性和生成过程低延迟的同时, 显著降低计算和能源消耗. 实验结果表明, 结合 GREEN-CODE 的代码生成模型平均降低了 23%–50% 的能耗, 同时几乎没有影响生成准确性. Storhaug 等人[79]提出了一种漏洞约束解码方法, 旨在减少大语言模型生成代码中出现的漏洞. 该方法使用一个由带有安全漏洞的代码组成的数据集微调模型, 使其在生成代码时能够为低安全性代码附带漏洞标签, 充当嵌入式分类器. 在解码过程中, 方法基于约束解码技术设计掩码以禁止生成包含漏洞标签的代码, 从而提升生成结果的安全性. 实验结果显示, 采用漏洞约束解码后模型能够有效识别并避免生成代码中的多数安全漏洞. Pimparkhede 等人[80]提出了 DocCGen 框架, 通过文档引导的约束解码来改进结构化领域特定语言 (domain-specific language, DSL)[81]中的代码生成. 该框架将自然语言到代码的生成任务分为两步: 首先, 通过信息检索识别与任务相关的代码库; 其次, 使用从库文档中提取的模式规则约束解码过程, 确保生成的代码符合目标库的语法要求. 实验表明, DocCGen 框架在处理两种复杂结构化语言 (Ansible YAML 和 Bash 命令) 的任务上能够有效减少生成代码中的语法和语义错误, 且在面对未见过的代码库时效果尤其明显. Li 等人[82]的研究则尝试使用约束解码提升代码健壮性, 他们统计了大语言模型生成代码在健壮性方面的缺陷来源, 并设计了 RobGen 方法. 该方法通过在解码过程中动态调整 token 概率, 鼓励模型添加控制结构以修复健壮性缺陷. 实验结果显示, RobGen 将模型生成结果中健壮性较差的代码减少了约 20%.

最新的研究尝试扩展了约束解码的能力. Mündler 等人[83]提出了一种类型感知的受约束解码方法, 旨在提高大语言模型生成代码的类型安全性. 该方法设计了一种基于前缀自动机形式化方法的算法, 能够确定模型的部分输出是否可以补全为类型良好的程序, 以此约束生成代码符合预定的类型要求. 算法的核心是基于输入逐步构建一棵抽象语法树, 使用类型相关上下文对其进行注释, 并维护前缀属性以确保只有有效的前缀才能产生非空的自动机状态集. 实验结果表明, 这种方法能够在包括代码生成在内的多种代码相关任务中显著减少编译错误, 并提升代码的功能正确性. Fu 等人[16]提出了 MUCOLA, 一种基于梯度的非自回归约束解码方法, 该解码方法会一次性生成所有的输出令牌, 能够同时考虑代码整体的安全性和正确性. MUCOLA 包含一个能量函数, 能够基于语言模型损失和约束惩罚项度量生成序列与目标约束条件的匹配程度. MUCOLA 通过梯度下降方法迭代地更新生成代码的嵌入向量以最小化能量函数, 引导模型生成符合约束的代码. 实验表明 MUCOLA 相比传统约束解码方法在约束代码安全性方面有更优的性能, 但在代码正确性方面则有所牺牲. Banerjee 等人[84]提出的 CRANE 方法则使用精心设计的附加规则来增强输出语法, 在确保模型输出正确语法的同时, 尽可能保留其推理能力. 实验表明 CRANE 在具有挑战性的符号推理基准上相较其他约束解码方法表现更好, 有效地平衡了受约束生成的正确性与不受约束生成的灵活性.

4.2.2 模型训练

模型训练方法则侧重于在预训练或微调阶段引入约束机制, 通过设计特定的训练任务和损失函数, 使模型在学习过程中自动掌握生成约束代码的能力. 这种方法通常依赖于高质量的训练数据和约束标注, 旨在根本上改善模型的生成性能, 使其在实际应用中更好地遵循既定规则. 虽然模型训练是在代码生成之前完成的, 但它的核心目标是通过训练引入约束信息和约束目标, 在生成阶段内化并体现这些约束, 其约束效果是在代码生成过程中即时生效的.

Tipirneni 等人[85]提出的 StructCoder 模型基于 Transformer[86]的编码器-解码器架构, 在模型训练阶段通过引入结构感知机制来约束代码生成. 该模型主要包含结构感知编码器和结构感知解码器两个部分, 其中结构感知编码器基于源代码的抽象语法树和数据流图信息生成结构化表示, 扩展编码器的输入序列, 并设计了结构感知自注意力机制为输入的不同部分计算相应的注意力; 在结构感知解码器端, 该研究在基本生成流程的基础上设计了辅助任务, 包括抽象语法树路径预测和数据流预测, 并设计了联合训练目标, 以促使生成过程遵循目标代码的语法和语义约束. 实验结果显示, StructCoder 在 CodeXGLUE[87]基准测试中的代码翻译和文本到代码生成任务中均取得了性能提升.

He 等人[88]提出了 SVEN 方法, 该方法通过训练特定连续向量 (称为 prefix) 并作为额外输入注入模型的生成过程, 引导模型生成符合特定安全属性的代码. SVEN 使用真实 GitHub 提交记录中的安全修复对作为训练数据, 其训练过程保持大语言模型的权重不变, 而是通过关注代码数据中的安全性敏感区域, 结合多种损失函数优化 prefix 参数, 使其能够在不干扰模型功能正确性的前提下, 指导生成代码满足安全要求. SVEN 还可以通过对抗性测试来降低模型安全性, 检验模型在生成不安全代码时的表现. 在一组主流大语言模型 (主要关注 CodeGen) 上的实验显示, SVEN 能够显著提高模型生成代码的安全性, 并且在生成符合功能要求的代码时与原模型表现相似. He 等人[89]的后续研究提出了 SafeCoder 方法, 旨在通过在指令调优阶段加入安全性约束, 提升大语言模型生成代码的安全性. SafeCoder 通过引入安全性微调, 在标准指令调优的基础上进行联合优化, 以同时提升模型的实用性和安全性. 该方法通过对安全代码和不安全代码的对比训练, 采用掩码的负对数似然损失和不可信度损失, 专注于生成符合安全要求的代码. 实验证明, SafeCoder 能够使主流大语言模型生成代码的安全性提高约 30%, 且对模型在功能正确性上的表现几乎没有负面影响.

Jain 等人[90]提出了一种代码数据清洗方法, 旨在通过改善训练数据中程序的可读性并使其更加结构化, 约束大语言模型生成正确性更高的代码. 该方法通过 3 步转换流程优化程序: (1) 变量重命名, 使变量名称更加描述性和一致; (2) 复杂代码模块化, 将程序分解为规模较小的子函数; (3) 插入自然语言注释, 简要描述程序的结构和功能. 实验结果表明, 在 APPS[91]和 CODE-CONTESTS[92]这两个挑战性算法代码生成基准上, 使用该数据清洗方法生成的数据集能显著提高模型的表现. 研究还发现使用高质量数据能提高数据效率, 在数据总量更少的情况下, 使用

清洗后数据集的模型表现超过了使用完整原始数据集的模型.

Sun 等人[93]的研究提出了 AI 导向语法 (AI-oriented grammar) 的概念, 旨在通过设计一种适合大语言模型使用的编程语言来约束大语言模型的代码生成过程, 在保持代码正确性的同时精简代码, 提高模型的生成效率. 具体而言, 该研究设计实现了一种简化的 Python 语言 SimPy, 在保持抽象语法树结构不变的前提下去除了 Python 代码中冗余的语法符号. 研究通过训练来让模型理解并有效使用 SimPy, 并探索了两种训练策略, 包括直接在 SimPy 数据集上训练以及先在 Python 数据集上训练再在 SimPy 数据集上微调. 相比直接在 SimPy 数据集上训练, 先训练后微调的策略被证明能够有效减少语法简化对模型表达能力的限制. 在 HumanEval 数据集上的评估表明, 受训后的模型在使用 SimPy 语法进行代码生成时, 相比使用 Python 语法有效减少了生成代码的令牌总量, 并在部分情况下提高了生成代码的正确性.

Islam 等人[94]针对代码修复任务设计了 SecRepair 系统, 该系统使用强化学习技术与语义奖励机制训练大语言模型, 约束模型生成修复代码的过程, 在确保修复具有较高正确性的同时要求模型附带生成简洁且语义准确的代码注释以提升代码可读性. 具体的训练过程中, 系统使用近端策略优化算法 (proximal policy optimization, PPO)[95]指导模型的参数优化, 目标是生成与标准答案接近的代码和注释. SecRepair 训练所使用的数据集包含与代码漏洞修复相关的多种任务. 研究应用 SecRepair 系统对多个开源物联网操作系统进行了漏洞分析, 其结果表明 SecRepair 在代码漏洞检测和代码修复任务上表现出色. Liu 等人[96]设计的框架 RLTF 同样基于强化学习技术, 但同时结合了软件测试的思想. RLTF 框架能够从单元测试结果中提取多种粒度的反馈信息, 并引入了针对未通过全部测试用例程序的自适应反馈机制, 能够根据通过测试用例的比例分配适当的惩罚. 在两个代码生成基准 APPS 和 MBPP 上的实验表明, RLTF 框架有效提升了模型代码生成的正确性, 并能适用于多种不同基础模型.

He 等人[97]的研究结合了模型训练与约束解码的思路, 采用监督式协同解码策略, 使用一个在漏洞修复数据集上微调过的小规模安全模型监督大语言模型的生成过程, 在解码时过滤掉可能带来安全隐患的 token 以提升代码安全性. He 等人[97]在上述方法的基础上设计了 CoSEFA 工具, 能够嵌入到 VSCode 等 IDE 中, 协助约束大语言模型的代码生成过程, 同时提供了后置的程序修复, 测试用例生成和代码解释等功能.

4.3 后生成约束方法

后生成约束方法主要针对生成完成后的代码进行全面评估与优化, 通过后处理机制进一步提升代码质量. 尽管这类方法通常并不直接干涉模型的代码生成过程, 但它们可以通过筛选或修改代码生成结果来约束整个系统最终的输出, 以达到等效的约束生成. 根据具体技术路径, 现有的预约束生成方法可分为 2 类: 测试验证方法和模型协同校正方法.

4.3.1 测试验证

测试验证方法利用自动化测试、静态分析工具以及其他评估手段, 对生成的代码进行全面检测. 该方法能够识别出语法错误、逻辑漏洞及性能问题, 并通过从多个生成样本中筛选或反馈给模型进行迭代优化等机制确保生成代码在交付前达到预定标准.

Hu 等人[98]提出了一种基于打印调试的代码生成改进方法, 旨在确保大语言模型在处理复杂数据结构和算法时的正确性. 对于代码生成任务, 该方法首先通过修改提示词引导模型将打印语句添加到生成的代码中. 其次, 执行代码并获取打印语句的输出. 之后, 方法基于打印语句输出和相关测试用例构建反馈信息交还给生成模型, 令其根据调试信息和测试用例内容的不一致性检测和修改代码中存在的错误, 对生成结果进行校正. 上述过程将会迭代进行直到模型生成的代码能够通过全部测试用例或达到终止条件. 该方法通过向代码中添加打印调试来追踪程序的执行流和变量值, 对这些信息的检查可以有效筛选不符合功能预期或存在逻辑漏洞的代码, 达到约束输出代码内部逻辑的效果. 实验结果表明, 该方法在解决中等难度的 LeetCode 问题时显著优于传统的模型自调试方法, 且在处理复杂的数据结构和算法时具有更高的调试效率.

Tian 等人[99]提出了μFiX (misunderstanding FiX) 方法, 通过改进大语言模型的规约理解来约束代码生成, 提升生成代码准确性. 该方法结合了两种提示技术: 思维启发提示和基于反馈的提示, 探索了它们的协同作用. 思维启

发提示主要通过分析测试用例来引导模型获取针对当前任务的规约理解, 在此基础上通过一个自我改进的过程辨别和修正规约理解中可能存在的误解. 在生成代码后, μFiX 运行测试并获取测试结果, 进入基于反馈的提示阶段. 如果代码未通过测试, μFiX 会分析先前获取的规约理解与实际生成代码的差异, 并由此构建调整指令来引导模型调整规约理解并生成正确的代码. 具体来说, μFiX 使用代码总结技术从生成代码中逆向推断出模型的实际规约理解并和思维启发提示中的规约理解进行比较. 实验结果表明, μFiX 在多个基准测试中的表现都优于其他提示技术.

Chen 等人[100]提出了 CODET 方法, 该方法主要关注从大语言模型生成的多个代码样本中选择最合适解决方案的挑战, 要求大语言模型同时生成代码候选和对应的测试用例, 并通过双重执行一致性策略在生成后对代码进行验证和筛选. 具体来说, 方法首先通过提供精心设计的提示使模型生成大量测试用例, 然后将所有生成的代码候选在这些测试用例上执行, 选出最可能正确的代码. 实验结果表明, CODET 显著提高了模型选择生成代码候选的性能, 并在多个基准数据集上提高了代码生成正确性指标. Jin 等人[101]的工作使用类似的思路, 提出了一种自进化的代码生成框架, 旨在通过迭代反馈机制持续提升大语言模型在代码生成任务中的性能. 该方法的核心思想是让模型不仅负责生成代码, 还能够自主生成测试用例并执行这些测试, 从而获取细粒度的执行反馈. 基于测试结果, 模型在多轮交互中对生成代码进行分析、修复和优化, 实现无需人工标注的自我改进过程.

4.3.2 模型协同校正

模型协同校正方法通过引入多模型协同工作的机制, 利用多个大语言模型间的互补优势对生成代码进行筛选或多轮校正. 此方法可以弥补单一模型在某些场景下存在的不足, 通过模型协作实现更为精细的代码评估和错误修正, 提升最终输出的准确性和鲁棒性. 相较于测试验证方法, 模型协同校正方法的特点在于依赖大语言模型的能力对生成代码进行后处理, 这种技术路径一方面可以利用模型在代码理解和错误检测方面的强大能力, 具有较好的泛用性和可扩展性, 但另一方面也需要承担额外风险, 即模型可能在后处理阶段引入新的错误.

Dong 等人[102]提出了一种自协作代码生成框架, 旨在基于软件开发中的团队合作实践提高大语言模型在复杂代码生成任务中的表现. 具体而言, 该框架通过角色指令 (role instruction) 为多个大语言模型代理赋予不同的角色 (分析师、编码员和测试员), 分别负责软件开发过程中的对应阶段并相互配合. 该方法利用其他模型检验生成模型的输出, 并反馈给生成模型指导其迭代改进, 达到了对最终输出代码施加正确性和质量约束的效果. 实验结果表明, 这种自协作框架相比单独的模型能够生成更高质量的代码, 特别是在复杂的编码任务 (如库级别编码任务) 中. 基于类似的思路, Zhang 等人[103]提出了一种协同模型的重排序方法 Coder-Reviewer reranking. 该方法借鉴软件开发中代码审查的思想, 引入两个角色: 编码模型负责根据自然语言指令生成代码, 评审模型则评估已生成代码是否合理地满足原始指令. 该方法本质上是最大互信息目标函数 (maximum mutual information, MMI)[104]的一种实例, 基于互信息建模约束代码生成结果, 降低生成退化代码 (如极短代码、重复代码等) 的风险. 实验结果表明, Coder-Reviewer reranking 在多个数据集与模型家族中均显著优于传统的基于生成概率的排序方法, 具有良好的通用性与易用性.

Nunez 等人[105]提出的 AutoSafeCoder 方法则实现了更复杂的模型协作过程, 以增强系统的整体能力. 该方法构建了一个多代理框架, 将代码生成、静态漏洞检测和基于模糊测试的动态分析有机结合. 在该框架中, 首先由编码代理生成初始代码; 随后, 静态分析代理利用常见弱点枚举 (CWE) 等标准对代码进行漏洞检测, 并提供具体的修正建议; 最后, 模糊测试代理通过变异生成输入, 检测代码在运行过程中的异常或崩溃情况. 静态分析代理和模糊测试代理的反馈信息被合并提交给编码代理, 对生成结果进行迭代修正, 以约束最终输出代码在安全性和正确性方面满足要求. 实验结果表明, AutoSafeCoder 相比单独的基准大语言模型, 能在不损失功能性的前提下将生成代码中的漏洞比例降低约 13%. Huang 等人[106]提出的 AgentCoder 框架同样采用 3 代理协同工作模式, 但具体分工有所差异, 包括程序员代理、测试设计代理与测试执行代理. 程序员代理生成代码后, 测试设计代理分析任务要求并生成基本、边界和大规模测试用例, 测试执行代理则将生成的代码和测试用例在本地环境中执行, 收集错误信息或测试结果. 测试执行代理的输出被实时反馈给程序员代理, 对代码进行迭代优化直到满足测试设计代理提出的所有约束要求. 对比其他代码生成与自我优化方法的实验显示出 AgentCoder 在测试生成准确性、代码覆盖率以及总体性能提升等多方面的优势, 消融研究则验证了多智能体协作设计在代码生成任务中的有效性.

模型协作方法已经被应用于安全攸关软件生成等工业级场景. 例如, Bayat 等人[107]提出了一种多代理的框架, 能够使用大语言模型将自然语言的规范合成为基于抽象的控制器设计 (abstraction-based controller design, ABCD). 该框架包括一个代码生成器代理和一个检查器代理, 其中代码生成器代理负责理解输入的自然语言描述并将其转换为可由符号控制软件解释的形式化语言; 检查器代理则负责验证生成语言的正确性, 并通过识别规范不匹配的情况来增强安全性. 在人工搭建的 20 个模拟环境中的实验证明该框架有效提升了大语言模型求解控制问题的能力, 同时使得生成的控制器设计更具健壮性.

4.4 小 结

本节从约束机制作用于代码生成过程的时机出发, 梳理了现有约束代码生成工作并将其分为预生成约束、生成中约束和后生成约束这 3 类方法, 对不同方法的原理进行了介绍和分析, 相关的研究总结见表 2.

表 2 受约束代码生成的实现方法总结

大类	子类	实例	主要约束类型
预生成约束	指令驱动	Pearce等人[55]	安全性
		Res等人[56]	安全性
		Nazzal等人[58]	安全性
		Tihanyi等人[60]	安全性
		Shi等人[62]	安全性
		Wu等人[57]	语法正确性和语义正确性
	检索增强	Parvez等人[63]	语法正确性和语义正确性
		Chen等人[64]	语法正确性和语义正确性
		Lu等人[65]	语法正确性和语义正确性
		Zhou等人[66]	语法正确性和语义正确性
		Koziolek等人[67]	语法正确性和语义正确性
	中间表示	Chen等人[68]	语法正确性
		Ma等人[69]	语义正确性
生成中约束	约束解码	Scholak等人[72]	语法正确性
		Ugare等人[73]	语法正确性
		Poesia等人[74]	语法正确性和语义正确性
		Zhang等人[75]	语法正确性和语义正确性
		Park等人[77]	语法正确性
		Ilager等人[78]	模型能耗
		Storhaug等人[79]	安全性
		Pimparkhede等人[80]	语法正确性
		Li等人[82]	健壮性
		Mündler等人[83]	语法正确性和语义正确性
		Fu等人[16]	安全性
		Banerjee等人[84]	语法正确性和语义正确性
	模型训练	Tipirneni等人[85]	语法正确性
		He等人[88]	安全性
		He等人[89]	安全性
		Jain等人[90]	语法正确性和语义正确性
		Sun等人[93]	语义正确性
		Islam等人[94]	语义正确性和可读性
		Liu等人[96]	语义正确性
		He等人[97]	安全性

表 2　受约束代码生成的实现方法总结 (续)

大类	子类	实例	主要约束类型
后生成约束	测试验证	Hu等人[98]	语义正确性
		Tian等人[99]	语义正确性
		Chen等人[100]	语义正确性
		Jin等[101]	语义正确性
	模型协同校正	Dong等人[102]	复合
		Zhang等人[103]	语法正确性和语义正确性
		Nunez等人[105]	语义正确性和安全性
		Huang等人[106]	语义正确性
		Bayat等人[107]	语义正确性和健壮性

对于预生成约束方法, 多数工作使用指令驱动和检索增强技术, 前者通常用于向模型提出具体的约束要求, 如强调代码安全性; 后者则通常用于整体生成效果的提升, 涉及语法正确性和语义正确性等多个方面. 预生成约束方法既不调整模型也不直接修改输出结果, 因此对模型生成过程的约束能力相对较差. 这类方法的特点在于低成本和易用性, 但适用场景相对受限, 且效果高度依赖于模型本身的能力.

对于生成中约束方法, 现有工作主要基于约束解码和模型训练两条技术路径. 约束解码技术强调对代码生成过程直接进行检查和约束, 通常用于需要严格语法约束的场景, 其优势在于高度的确定性和可靠性, 但也存在过度约束导致模型生成能力严重下降的弊端. 模型训练则通过预训练或微调修改模型参数, 使其能够更好地处理目标任务. 通过对不同约束需求设计不同的训练数据集, 模型训练的适用场景较为广泛, 但效果高度依赖于训练数据集的质量, 可能出现过拟合情况, 且规模较大的模型训练成本很高. 生成中约束方法的整体优势在于其强大的优化能力, 能够针对不同规模不同能力的模型在多种约束需求上提升代码生成的效果, 且针对模型代码生成过程的约束能力是 3 类方法中最高的. 作为代价, 这类方法的使用成本和难度也是 3 类约束方法中最高的, 通常需要根据具体领域和具体问题建立对应的约束工具或高质量数据集.

对于后生成约束方法, 测试验证和模型协同校正都是目前很常见的思路. 测试验证通过测试用例和静态分析工具检测生成代码中的语义和语法错误, 模型协同校正则利用大语言模型的能力检测代码漏洞, 二者的反馈通常都会被重新提交给生成模型, 基于迭代式生成提升最终输出代码的质量. 测试验证的优势在于轻量化和可解释性, 对于其能力之内的代码错误都能高效处理且有详细的过程信息, 但对于结构复杂或规模较大的代码则往往不能达到很高的准确性. 相比之下, 模型协同校正方法理解和处理复杂代码的能力更强, 但引入了新的大语言模型导致整个过程更加不可控, 模型可能因为幻觉检测虚构的漏洞, 或是在多轮迭代修复过程中顾此失彼. 尽管后处理可以检测多种类型的约束, 但现有工作多数仍基于代码测试构建方法, 因此主要关注代码语义正确性. 另外, 虽然后处理可以基于对代码生成结果的直接修改提供较强的约束能力, 但由于现有工作多数会将检测结果反馈给模型进行代码优化或重新生成, 存在模型在后续生成中引入新错误的风险, 因此对代码生成过程的整体约束能力处于 3 类方法中的中游水平.

总结而言, 3 类受约束代码生成方法基于其不同的技术特点和约束能力, 适用于不同的约束需求和问题场景. 预生成约束方法最易于使用且通常能够整体性地提升代码质量, 但需要对某类代码问题进行高度约束的场景往往需要使用生成中约束方法或后生成约束方法来提供更加可信的约束水平. 对于基础的代码语法和语义正确性问题, 3 类方法都有相关研究并实现了一定程度的改善, 但对于进阶的代码质量问题, 如可维护性和可读性, 针对性的研究很少.

5　受约束代码生成的评估方法

在受约束代码生成的研究中, 评估方法是至关重要的一部分. 由于代码生成的任务涉及多个维度的质量要求

(如正确性、安全性、可维护性等), 因此不同类型的约束需求和约束生成方法通常需要不同的基准数据集和评价指标, 以准确衡量生成代码在不同维度上的表现. 本节将介绍评估受约束代码生成技术时常用的基准数据集和评价指标, 为后续研究提供参考.

5.1 基准数据集

评估受约束代码生成的效果通常依赖于标准化的基准数据集, 这些基准数据集包含一组标准化的问题或任务, 且通常具有测试用例以及标准答案. 不同的约束需求及对应的约束生成任务通常需要不同的数据集. 例如对于代码安全性约束生成任务, 可能需要包含安全漏洞数据的数据集; 而对于代码运行效率的约束生成场景, 可能需要测试生成代码在高性能计算场景下的表现. 基准数据集的数据多样性和覆盖面将影响评估结果的代表性和可靠性, 而对于特定约束生成方法的评估, 选择合适的数据集也尤为关键.

5.1.1 代码正确性基准数据集

这类数据集主要用于评估模型生成的代码是否能够正确地完成给定任务. 衡量指标往往是通过运行生成代码的测试用例来判定其正确率.

最具代表性的工作是 Chen 等人[1]构建的 HumanEval 数据集, 该数据集由 164 个人工编写的编程问题组成, 主要用于评估模型生成代码的功能正确性. 每个问题包含函数签名、自然语言描述、函数体以及一组单元测试, 平均每个问题包含大约 7.7 个测试用例. 尽管在规模和题目多样性上存在不足, HumanEval 仍是评估代码生成技术效果的重要工具, 并为后续构建更大规模、多样化的数据集提供了有价值的借鉴.

Austin 等人[49]构建了 MBPP 数据集, 该数据集包含 974 个短小的 Python 程序. 每个问题包括问题描述、解决该问题的 Python 函数及 3 个用于验证函数语义正确性的测试用例. MBPP 的设计目的是评估模型生成代码的能力, 问题范围包括简单的数学计算、列表处理、字符串处理等基础任务. 该研究同时还构建了 MathQA-Python 数据集, 这是一个从现有 MathQA[108]数据集修改而来的数据集, 旨在测试模型生成程序解决数学问题时的表现. 在这些数据集上对不同规模大语言模型的实验表明, 较大的模型能够更有效地解决问题, 微调模型能够进一步提升生成代码的质量.

Dai 等人[109]构建了 MHPP 数据集, 包含 210 个独一无二的、由专家人工设计的 Python 问题, 其设计初衷是弥补现有数据集在题目质量、难度分布和细粒度区分能力等方面存在的不足, 通过设置多种挑战来更全面地考察模型的代码生成能力. MHPP 数据集中的每个编程问题不仅提供了更详细的自然语言描述, 还配有更多的测试用例和一个高质量的标准答案. 研究者采用了严格的措施防止数据污染, 确保所有问题均为原创. 在 MHPP 数据集上对 26 个大语言模型进行的评估显示, MHPP 能更细致地揭示出模型在处理复杂问题时的局限性.

Cao 等人[110]构建了 JavaBench 数据集, 用于评估大语言模型在面向对象代码生成方面的能力. JavaBench 主要关注项目级 Java 编程任务, 包含 4 个来源于本科入门级课程编程作业的 Java 项目. 每个项目都提供了详细的项目描述、代码骨架 (包括类结构、方法签名和 TODO 标记的待补全部分) 以及完整的测试套件, 测试覆盖率高达 92%. 项目中的各个类和方法均经过经验丰富的 Java 程序员验证. 此外, 数据集还包含了本科生对这些项目的评分记录, 用以反映项目的难度和质量.

Jimenez 等人[111]构建了 SWE-bench 数据集, 该数据集来自多个开源 GitHub 代码库, 主要用于评估大语言模型在解决实际软件工程问题中的表现. 每个任务实例包括一个具体的问题描述 (通常是一个 bug 报告或功能请求) 和相关的代码库, 模型需要生成一个补丁来修复问题并通过相关的测试. 该数据集的主要特征包括: 问题描述通常较长, 代码库庞大, 且解决方案涉及多文件、多函数的编辑. 数据集的任务实例经过多阶段筛选, 确保了高质量的任务实例, 最终提供了 2 294 个有效任务实例. 在 SWE-bench 上的实验表明, 尽管许多模型能够在某些简单任务中表现较好, 但它们在复杂任务上普遍未能有效解决问题.

Yu 等人[112]构造了 CoderEval 数据集, 也关注到了大语言模型在实际代码生成场景中表现的评估. CoderEval 从多个开源项目中收集并筛选得到了 230 个 Python 问题和 230 个 Java 问题, 其中不仅包括独立函数, 还涉及很多依赖于外部库、类或文件等上下文信息才能正常工作的非独立函数. 基于该数据集的实验显示出现有的生成模型

在生成独立函数时的效果要优于非独立函数.

Vijayaraghavan 等人[113]提出了 VHDL-Eval 框架, 该框架为评估硬件设计中 VHDL 代码生成的效果而构建了一个标准化的基准数据集. 该数据集包含 202 个问题, 其中既包括由 Verilog-Eval[114]数据集问题转化而来的 VHDL 问题, 也包含从公开教程中收集的问题. 在这些问题中, 主要任务是生成符合 VHDL 语法和功能要求的代码, 用于描述硬件逻辑和电路设计. 在框架中, 首先通过将原始问题进行语言转换和后处理, 形成一个多样化的测试集; 随后, 集成了自验证测试平台, 利用预定义的测试用例自动验证生成代码的语法正确性和功能实现. 基于该框架的实验表明, 现有的大语言模型在 VHDL 代码生成任务中表现仍有较大提升空间.

Yan 等人[115]构建了一个综合评测数据集 CodeScope, 包含了两个大类的 8 个任务 (代码理解: 代码摘要、代码异味检测、代码审查、自动化测试; 代码生成: 程序综合、代码翻译、代码修复、代码优化), 并覆盖了 43 种编程语言. 其数据充分体现了现实软件开发中多语言、多范式的特点, 能够模拟真实开发场景并全面地评估大语言模型在处理复杂代码任务时的通用性和健壮性. 实验结果显示 CodeScope 相比传统基准数据集更能反映真实编程任务中模型的优劣.

5.1.2 代码质量基准数据集

这类数据集则侧重于评估生成代码的整体质量. 除了正确性测试外, 还可能考虑代码的运行效率、可读性、可维护性和安全性等方面. 除了自动化测试用例, 一些数据集还会基于人工评审等方式对生成代码进行评分.

Niu 等人[48]构建了 LeetCodeEval 数据集, 该数据集包含 LeetCode 在线评测平台中不同难度的编程问题, 主要用来评估大语言模型生成代码的正确性和运行效率. 相比 HumanEval 和 MBPP 等常用数据集, LeetCodeEval 的特点在于其任务普遍具有更长的提示词和更多的测试用例数量, 且可以获取由 LeetCode 平台提供的部分运行时指标, 如运行时间测量和代码效率排名. 在 LeetCodeEval 上的实验表明生成代码的正确性与运行效率不一定正相关, 且不同规模的模型在生成代码的运行效率上没有显著差异.

Du 等人[51]构建的 Mercury 同样基于 LeetCode 平台并关注到代码的运行时性能评估问题, 其数据通过一系列过滤条件 (如每个任务至少需有两份历史提交、限制数据结构只包含内置类型和两个自定义结构——二叉树和链表, 以及确保输出唯一性) 来保证各个任务的质量和统一性.

Huang 等人[46]构建的 EffiBench 数据集也基于 LeetCode 平台收集数据, 根据编程题目在面试中的出现频率和涉及的算法 (如贪心、动态规划、回溯等) 筛选出效率至关重要的题目, 并开发了一个测试用例生成器, 能够为每个题目自动生成大量具有不同输入规模、数据分布以及边界情况的测试用例.

Liu 等人[116]提出了 DPE 框架, 旨在通过设计高计算量的任务来更可靠地区分不同模型生成代码的运行效率. 该框架首先从已有的数据集中选择一组编码任务; 之后, 基于合成合成器 (synthesizing a synthesizer, SAS) 方法, 引导大语言模型为每个任务生成一个可控规模的测试输入采样器, 并通过指数输入采样生成大量具有挑战性但可计算的测试输入; 最后, 对每个任务的各个正确解进行多次性能采样, 使用过滤策略筛选出既有足够计算量又容易在代码性能上有明显差异的任务, 并利用自适应聚类方法将解决方案按照效率分成多个层次. 该研究最终挑选出 121 个满足要求的性能挑战性编程任务构成 EVALPERF 数据集. 在 EVALPERF 上的实验显示现有的为代码性能优化设计的提示并没有始终显著提高生成代码的效率.

Wang 等人[117]构建了 CodeSecEval 数据集, 主要用于评估代码生成和代码修复任务中的安全性要素. 数据集共包含 180 个样例, 覆盖 44 种关键漏洞类型, 能够较全面地反映各类安全漏洞在代码中的表现. 每个实例都提供了问题描述、原始的不安全代码和参考的修复后安全代码, 以及测试用例与入口函数等信息. CodeSecEval 数据集可以分为两个子集: SecEvalBase 基于现有的 SecurityEval[118]数据集构建, 人工补充了部分关键信息; SecEvalPlus 则针对部分高风险漏洞进行重点构建, 样本分布更均衡. 所有数据都经过了严格的人工校验与清洗. 在该数据集上的实验结果表明目前的模型在生成代码或修复不安全代码时往往会忽略很多安全问题, 且不同漏洞类型下模型的表现存在较大差异.

Fu 等人[16]构建的 CODEGUARD+数据集关注代码大语言模型生成代码时的安全性与正确性评估. 该数据集基于已有的安全提示数据集 (包括 Pearce 等人[119]的数据集和 SecurityEval) 进行修改, 使其中的任务更适合自动

化测试, 具体措施包括添加明确指令、更换适合测试的库以及更新废弃函数. 在 CODEGUARD+数据集上对 8 种先进模型的实验表明, 不同的解码方法对生成代码的安全性和正确性有显著影响.

Zheng 等人[120]提出了一种多维度评估框架 RACE, 涵盖了代码的正确性、可读性、可维护性和运行效率这 4 个维度. 该框架整合了 4 个现有数据集: HumanEval+[14]、MBPP+[14]、ClassEval[33]和 LeetCode[121], 并针对不同的评估维度选择相应的数据集与评价指标.

5.2 评价指标

评价指标用于量化生成代码的质量, 并且根据不同约束任务的需求, 涵盖多个维度的评估标准. 评价指标不仅帮助衡量生成代码在多大程度上满足正确语法和预期功能, 还可以评估代码在安全性、可维护性、可读性和运行效率等方面的表现. 不同类型的代码生成任务往往会重点关注某些特定的评价维度.

5.2.1 代码正确性评价指标

代码正确性评价指标主要用于衡量代码是否按照预期功能正确运行, 关注代码能否在特定输入下产生正确的输出, 是否能通过预定的测试用例, 是否没有逻辑错误和语法错误.

Chen 等人[1]构建 HumanEval 数据集的研究中使用的 Pass@K 指标是目前衡量模型生成代码正确性的常用指标. 该指标衡量在模型生成的 K 个样本中至少有一个样本能够通过所有单元测试的概率, 能更好地反映模型在生成多样化代码样本时的实用性. 该研究对 Pass@K 的计算方法做了改进优化, 以确保无偏估计并提升数值稳定性. Dai 等人[109]的研究在沿用 Pass@K 的基础上利用该指标进一步分析了模型能力, 通过将代码生成中存在的问题按照难度和成因分成多个类别 (如干扰、重新定义、捷径、常识、边界情况等), 并在每个类别上分别计算 Pass@K, 能够清楚地看出模型在哪些方面 (如处理复杂逻辑、多步推理或应对干扰信息) 表现较弱.

Yu 等人[112]基于 Pass@K 的思路设计了 Acc@K 指标, 主要关注生成代码中是否包含了所有必要的上下文信息. Pass@K 仅评估代码是否能通过测试用例, 但实际生成代码往往依赖于大量的外部上下文元素, 仅依赖于测试用例的正确性无法全面反映模型在实际编程场景中的表现. Acc@K 指标首先需要标识出目标函数 (即应当实现的函数) 的上下文依赖, 例如外部变量或第三方库; 之后, Acc@K 计算生成的前 K 个候选解中, 能够包含正确上下文信息的代码片段所占的比例.

Jimenez 等人[111]在对 SWE-bench 数据集的评估中同样以测试用例为基础, 使用了 fail-to-pass 和 pass-to-pass 两种指标评价模型生成修复代码的质量. 这两个指标主要关注代码修改前后通过测试用例情况的变化, 其中 fail-to-pass 计算从失败转为通过的测试数量, 评估代码修复的有效性; pass-to-pass 计算修改前后保持通过的测试数量, 评估代码修复是否破坏了已有的功能, 引入了新的问题.

Austin 等人[49]在其构建的 MBPP 数据集上对比了多种评估指标, 发现传统的 BLEU 分数[122]等基于文本内容的指标可以反映代码与参考答案在词汇层面的重叠情况, 但与功能正确性并无明显相关性; 相比之下, 基于执行结果的指标能够直接衡量生成代码的功能正确性, 更能反映模型的实际能力. Evtikhiev 等人[123]的研究则对 6 种现有的代码正确性评估指标进行了系统性的对比和评估, 在 CoNaLa 和 Hearthstone[124]两个数据集上分析了不同指标之间的差异性以及它们与人工评估结果的差异性. 结果显示, 当分数差异较小时, 自动评价指标往往不能可靠地区分代码质量. 6 种指标中, ChrF[125]和 ROUGE-L[126]表现较为稳定, 与人工评分的相关性较高, 能够更好地区分代码生成质量的细微差异; 而 BLEU、CodeBLEU[127]和 RUBY[128]在某些情况下存在较大偏差. 该研究认为基于单独的指标分数并不足以准确反映模型代码生成结果的真实差异, 在使用自动评价指标时需要结合统计显著性检验.

Cao 等人[110]针对项目级 Java 编码任务设计了一套包含 3 个层次度量的系统评估方法, 通过计算生成代码中待补全部分被正确完成的比例、生成代码成功通过编译的比例和成功通过测试用例的比例, 从不同粒度对模型生成效果进行综合评估.

5.2.2 代码质量评价指标

代码质量评价指标主要用于衡量代码的可读性、可维护性、性能、结构和安全性等方面.

Niu 等人[48]为了有效评估代码运行效率, 选择通过 gem5 CPU 模拟器计算生成代码的运行时间, 以减少真实

硬件测试的噪声并提升评测的可重复性. 文章对不同问题的运行时间进行了归一化处理, 以削弱编程问题本身的复杂度和输入规模带来的差异性, 更好地计算模型在生成代码运行效率方面的综合性能. 对于数据集中来自 LeetCode 平台的编程问题, 该研究直接通过向平台提交代码获取代码正确性评价和运行时间, 并基于平台提供的击败其他用户代码的百分比的数据评估代码的运行效率综合表现. Huang 等人[46]同样使用归一化处理调整了评估代码运行效率的指标, 包括归一化运行时间、归一化最大内存使用和归一化总内存使用, 其中最大内存使用指代码运行过程中内存的峰值, 总内存使用则分析整个运行过程的内存使用曲线.

Du 等人[51]针对生成代码的运行时性能提出了一个新的评价指标 Beyond, 该指标基于每个编码任务的运行时分布来评估代码效率. 具体而言, 首先从历史解决方案中收集每个任务的运行时间, 形成运行时分布; 然后对于模型生成的代码, 通过计算其在这个分布中所处的百分位数, 得到 Beyond 分数. Beyond 指标能够同时反映代码的功能正确性 (若代码运行出错则得 0 分) 以及效率, 理想情况下其值应当与功能正确性指标相当, 从而体现出在正确执行的前提下代码的高效程度. 实验还表明 Beyond 指标具有硬件无关性, 即在不同硬件配置下评估结果较为一致. Liu 等人[116]基于类似的思想设计了差异表现分数 (differential performance score, DPS) 指标, 通过比较生成代码的性能与一组参考解, 得到一个介于 0–100 之间的得分, 直观表示新代码相对于已有解的效率排名. 该研究还引入了标准化版本的 DPSnorm 指标, 用于消除各性能层次中样本数量不同带来的影响.

Vartziotis 等人[129]提出了一个新的指标, 绿色容量 (green capacity, GC), 用来综合衡量代码在能耗、运行时间、内存使用和浮点运算次数 (FLOPs) 方面的可持续性表现. 绿色容量的计算依赖于一个滤波性能增量 (filtered performance delta, PD) 函数, 该函数用于比较初始代码与优化后代码在各项指标上的改进程度. 使用绿色容量指标进行评估需要将代码生成划分为两个阶段: 第 1 个阶段生成保证正确性的初始代码, 第 2 个阶段则在初始代码基础上生成优化版本的代码. 另外, 也可以对比模型生成代码与人类编写的高质量代码并计算绿色容量, 从而评估模型在绿色编码上的效果.

Fu 等人[16]针对代码安全性评估, 分析了原有的 SVEN-SR[88]指标的局限性, 并提出了新的指标. 该研究在 Pass@K 的基础上设计了 secure-pass@K 和 secure@Kpass, 前者计算生成的 K 个候选代码中, 有多少候选代码能同时通过安全检查和单元测试; 后者计算在所有通过单元测试 (即功能正确) 的代码中, 有多少比例是安全的. 新指标弥补了传统 SVEN-SR 指标只关注安全性、忽略正确性的问题, 使得评价结果更符合开发者对可用代码的实际需求. Peng 等人[130]同样关注到大语言模型生成代码安全性的评估问题, 设计了 CWEval 评估框架和相应的 CWEval-bench 基准数据集, 能够在评估代码功能性的同时, 基于高质量的任务规范和测试结果参照物评估代码的安全性. CWEval 使用的指标是由 Pass@K 改造而来的 func@K 和 func-sec@K, 对应功能性测试用例和安全性测试用例, 兼顾了安全性评估和正确性评估.

Yan 等人[115]在构建 CodeScope 数据集的同时, 以 Pass@K 为基础, 针对数据集中特定类型的任务设计了相应的改进评价指标. 对于代码修复任务, 提出了 DSR@K (debugging success rate@K) 指标, 即在最多 K 轮调试后代码能够正确执行的比例. 对于代码优化任务, 设计了新的指标 Opt@K, 即在最多 K 次优化中优化代码能够在效率方面超过原始代码的比例.

5.3 小 结

本节介绍了受约束代码生成的评估方法, 主要包括基准数据集和评价指标两个方面, 相关的研究总结见表 3.

现有的受约束代码生成研究大多使用通用代码生成基准数据集, 通过修改、补充数据或设计特殊的评价指标来适应受约束场景的评估需求. 目前最具影响力的基准数据集包括 HumanEval 和 MBPP. 我们从各个约束分类中选取了使用上述两个数据集的方法并比较了它们的效果 (通过 Pass@1 评估的功能正确性), 以提供一种对约束方法能力的直观评估, 结果见表 4. 其中加粗数值表示对应评测设置下的最优结果 (即该列中的 Pass@1 指标最高值). 通过对最优结果进行加粗标注, 直观突出各受约束代码生成方法所用基线以及约束后结果在常用评测数据集上所达到的代码生成最优效果. 需要注意的是, 该比较结果仅能作为一种片面的参考, 因为不同的约束方法关注的代码问题不同, 技术的侧重点不同, 使用的基座模型也有差异. 尽管它们都使用了 HumanEval 或 MBPP 数据集, 且都计

算了 Pass@1 指标, 但其实验重点可能并不在该指标上, 例如 Ugare 等人[73]的 SynCode 在 Pass@1 上与基线相比无改善, 但在单独的实验中证明了其生成代码中包含的语法错误减少了 90% 以上. 另外, 即便是使用相同基座模型的方法, 也会因为模型参数、版本或随机性问题导致在同一个数据集上出现不同的结果, 例如 Ma 等人[69]、Tian 等人[99]和 Dong 等人[102]都使用了 ChatGPT 作为基座模型, 但在 HumanEval 上的结果有所差异.

表 3 受约束代码生成的评估方法总结

分类	数据集	实例	评价指标	主要约束类型
代码正确性	HumanEval	Chen等人[1]	Pass@K	语法正确性和语义正确性
	MBPP	Austin等人[49]	任务通过率, 测试用例通过率	语法正确性和语义正确性
	MHPP	Dai等人[109]	Pass@K	语法正确性和语义正确性
	JavaBench	Cao等人[110]	Pass@K, Completion@K, Compilation@K	语法正确性和语义正确性
	SWE-bench	Jimenez等人[111]	fail-to-pass, pass-to-pass	语法正确性和语义正确性
	CoderEval	Yu等人[112]	Pass@K, Acc@K	语法正确性和语义正确性
	VHDL-Eval	Vijayaraghavan等人[113]	Pass@K	语法正确性和语义正确性
	CodeScope	Yan等人[115]	Pass@K, DSR@K, Opt@K	语法正确性、语义正确性和运行效率
代码质量	LeetCodeEval	Niu等人[48]	Pass@K, 运行时间, %Beats	运行效率
	Mercury	Du等人[51]	Beyond	运行效率
	EffiBench	Huang等人[46]	运行时间, 最大内存使用, 总内存使用	运行效率
	EVALPERF	Liu等人[116]	DPS, DPSnorm	运行效率
	—	Vartziotis等人[129]	绿色容量	运行效率
	CodeSecEval	Wang等人[117]	Pass@K	安全性
	CWEval-bench	Peng等人[130]	func@K, func-sec@K	安全性
	CODEGUARD+	Fu等人[16]	SVEN-SR, secure-pass@K, secure@Kpass	安全性
	HumanEval+, MBPP+, ClassEval, LeetCode	Zheng等人[120]	代码长度, 可维护性指数, 时间复杂度, 空间复杂度等	复合

表 4 部分受约束代码生成方法在 HumanEval 和 MBPP 上的 Pass@1 结果对比 (%)

实例	分类	工具	基座模型	HumanEval		MBPP	
				基线	使用工具	基线	使用工具
Ma等人[69]	中间表示	SeCoT	ChatGPT	64.15	68.90	61.45	65.85
			WizardCoder-13B	53.66	55.48	46.42	46.89
Ugare等人[73]	约束解码	SynCode	WizardCoder-1B	20.00	20.00	—	—
Mündler等人[83]	约束解码	Types	Qwen2.5-32B	**79.60**	81.80	**76.30**	**76.60**
He等人[88]	模型训练	SVEN	CodeGen-350M	6.70	6.80	—	—
			CodeGen-6.1B	18.60	17.60	—	—
He等人[89]	模型训练	SafeCoder	CodeLlama-7B	28.60	35.90	35.90	35.10
Tian等人[99]	测试验证	μFiX	ChatGPT	73.78	**90.24**	—	—
			DeepSeek-Coder	76.22	83.54	—	—
Dong等人[102]	模型协同校正	Self-Collaboration	ChatGPT	57.30	74.40	52.20	68.20

许多研究在 HumanEval 的基础上设计了结构类似的新数据集, 以关注特定的代码生成问题和约束需求. 例如, CoderEval 关注非独立函数生成, Verilog-Eval 和 VHDL-Eval 则专注于嵌入式开发编码. 更复杂的数据集通常基于实际软件项目构建, 能够为更多的约束类型提供研究场景和评估基准, 如关注实际软工问题的 SWE-bench 和关注

项目级代码生成的 JavaBench.

这些数据集主要通过参考答案和测试用例来评估生成代码的正确性, 因此在受约束代码生成的研究中, 通常用于语法和语义约束的场景. 针对特定约束需求, 部分研究设计了专用的基准数据集, 主要集中在代码质量约束场景, 如一系列基于 LeetCode 平台构建的运行效率评估数据集和基于常见弱点枚举 (CWE) 构建的安全性评估数据集等. 尽管目前缺少专门针对代码可维护性和可读性的评估数据集, 部分研究在其他基准测试中也同时评估了这些指标.

在受约束代码生成的评价指标方面, 最常用的是与 HumanEval 紧密结合的 Pass@K 指标, 该指标能较好地评估代码在语法和语义方面的综合正确性. 基于 Pass@K, 研究者设计了多个变种, 以适配特殊任务场景或评估额外指标, 如运行效率和安全性. 针对代码运行效率的研究还使用了代码运行时间和内存使用情况作为评价指标, 且最近的研究中常出现通过综合效率排名评估代码性能的设计. 对于代码可维护性和可读性, 虽然存在一些自动化指标, 如代码长度和可维护性指数, 但这些指标只能作为参考, 准确的评估通常还需要基于人工评分进行.

总结而言, 受约束代码生成的评估方法大多基于现有代码生成评估方法, 通过修改数据集和评价指标来适应不同的约束类型. 专门针对受约束生成问题构建的数据集相对较少, 且主要集中在代码运行效率和安全性研究方向. 评价指标的设计通常与数据集紧密相关, 除了运行效率方面的独有指标外, 其他领域的指标多为 Pass@K 的变体, 且缺少关于代码可维护性和可读性等方面的自动化评估指标.

6 受约束代码生成的挑战与展望

受约束代码生成作为提升大语言模型代码生成能力的重要技术, 已经在学术界和产业界引起了越来越多的关注. 围绕这一主题, 现有工作从大语言模型代码生成中存在的问题、受约束代码生成的实现方法以及评估方法这 3 个方面进行了研究和探索, 并取得了初步的成果. 本文分析与总结了相关文献, 并提出下述挑战.

6.1 适用复杂约束机制的代码生成方法

目前受约束代码生成的研究主要处于理论验证阶段, 使用的约束机制较为简单. 当前技术的主要局限性在于, 复杂的任务需求往往对应了复杂、复合型的约束策略, 而简单的约束机制很难准确描述这些策略, 或是容易导致约束成本过高. 要在现实的复杂软件开发任务中发挥约束机制的作用, 未来还需要在以下 3 个方面进行探索和突破.

(1) 多阶段约束融合机制. 现有约束方法通常只在代码生成的特定阶段添加单一的约束机制, 其效果和适用范围都有很大的局限性. 未来应探索在预生成、生成过程中以及后处理阶段分别施加约束, 并设计各阶段之间的协同工作机制, 实现多层次、多维度约束的有机融合. 例如, 在预生成阶段先利用检索增强技术自动匹配任务相关的语法知识, 在生成中阶段和后生成阶段调用该语法知识, 以支持约束解码的补全引擎和后置的语法检查器.

(2) 基于用户反馈的动态约束策略. 目前约束方法通常基于静态的约束策略, 而面向开发者的应用场景要求代码生成过程具有良好的交互性和实时反馈能力. 未来应关注如何在受约束代码生成的过程中引入开发者的协同与反馈机制, 实现动态可变约束策略. 例如, 设计一套智能体交互策略, 允许开发者在生成过程中根据模型输出结果实时修改约束条件, 添加新的规则或调整已有规则, 协助模型进行下一轮迭代. 另一种可能的方向是使用大语言模型自动总结对话历史信息并评估当前约束方法的效果, 据此对约束条件进行动态修改, 实现自适应的约束机制.

(3) 约束的场景适应性与领域知识注入. 现有的约束方法多数基于简化的问题场景, 而实际软件开发中不同领域对代码的要求差异很大. 即便是相同类型的约束, 其具体形式也常常不同. 未来研究应考虑如何根据具体场景自动生成约束策略, 结合领域知识对代码生成施加定制化的约束, 以满足不同行业不同场景的具体需求. 例如, 从领域内已有的软件开发文档中自动抽取领域关键特征, 由大语言模型合成一组约束规则, 配合约束解码等技术快速构建领域特定的约束机制.

6.2 约束方法在工业级代码生成的应用

目前受约束代码生成的研究主要面向简化的实验室环境, 初步的实验结果显示出此类技术在代码生成任务中的独特优势. 然而当前技术的主要局限性在于, 许多方法的设计脱离实际软件生产环境的特点, 无法真正满足现实

存在的需求, 例如部分约束解码技术牺牲功能正确性来保证安全性, 或是在约束机制中使用了现实很难获取的高精度规范. 如何在资源受限的环境中部署方法并达到性能要求也是当前研究很少考虑到的问题. 未来要把受约束代码生成技术应用于工业级代码生成场景, 还需要在以下 3 个技术点进行探索和研究.

(1) 安全攸关软件的自动生成. 安全攸关软件指那些对代码正确性和安全性要求极高的软件, 例如航空航天软件和汽车软件. 这类软件往往复杂度较高且需要反复核查, 人工编写的成本很高. 将大语言模型应用于安全攸关软件的自动生成可以有效节省人力, 但首先需要采取技术手段消除模型的幻觉、随机性等特性带来的负面影响, 最大程度保证代码生成过程的稳定性和确定性, 同时提高生成结果的可用性. 受约束代码生成技术可以在这些方面有效减少模型的不确定性缺陷, 帮助大语言模型在工业级场景中落地. 具体的技术挑战包括如何将软件所遵循的安全标准提取出来并整理为模型容易理解的形式, 选用何种约束机制将安全标准应用于模型的生成过程, 以及在获取生成代码后如何判断是否正确遵守了安全标准.

(2) 仓库级和模块级等复杂代码生成场景. 现实中的代码生成任务往往不是处理独立的程序或代码片段, 而是在软件仓库或软件系统中生成某个部分的代码. 在这种情况下, 仓库级、模块级的上下文对代码生成的影响很大. 例如, 在为企业的软件项目生成新的功能代码时, 需要考虑该项目或关联项目中已有的宏定义和函数, 而不是从零开始生成代码. 对大语言模型而言, 由于这些上下文很多都是私有性质, 不会出现在开源数据中, 因此理解起来存在一定困难. 受约束代码生成技术可以帮助开发者将复杂的代码上下文通过多种途径准确地描述给大语言模型, 协助模型理解仓库级和模块级的代码生成场景. 例如, 源代码知识和相关文档可以在预处理之后通过检索增强技术约束模型; 从已有代码中提取的编码规范和 API 使用模式可以通过约束解码技术限制模型的生成过程, 或是用于模型的微调; 外部的代码分析和验证工具可以作为后置的约束协助模型进行迭代.

(3) 受限环境中的受约束代码生成. 在许多工业级场景中, 软件所能使用的硬件资源都是非常受限的. 对于需要部署大语言模型的场景, 这意味着选择的模型不能超过一定的参数规模. 从现有实验可以看出, 参数规模较小的模型各方面能力都有所不足, 例如很难准确遵循提示词中的指示, 或是经常生成格式不规范的输出. 受约束代码生成技术可以有效提升小参数量模型在特定方面的表现, 使其在部分任务中可以达到接近大型模型的生成效果. 例如, 需要从文档中提取特定数据填入配置文件的场景, 约束机制可以通过引入语法规范强制模型输出的结果满足配置文件要求, 消除可能的语法不匹配问题; 同时, 还可以结合后置的规则检测器, 通过后生成约束的方法验证提取结果和原始文档数据的匹配情况, 引导模型迭代修正. 这些机制会有效降低任务对模型参数量的要求.

6.3 约束方法适配不同基座模型

目前受约束代码生成的研究通常使用不同的基座模型. 尽管多数方法都证明了约束机制在它们选择的基座模型上正确生效, 但约束方法和基座模型的适配性问题尚缺少系统性的研究. 当前研究的主要局限性在于, 多数工作没有深入分析约束机制与基座模型之间的关联关系, 其实验部分也只是列出了在数据集和评价指标上表现最好的基座模型, 而缺少对背后原理的分析. 随着大语言模型的快速发展, 模型能力迭代很快, 在旧版本模型上有效的方法是否能被迁移到新版本模型上, 目前也缺少相关的实验或证明. 未来要在更多型号、更新版本的大语言模型上应用约束方法, 还需要在以下两个方面进行探索和突破.

(1) 深入分析基座模型和约束机制的适配关系. 当前研究通常会寻找能使评价指标达到最优的基座模型和约束方法的组合, 并会为此尝试和对比多种可能的基座模型. 尽管这些对比实验已经从一定程度上揭示了基座模型和约束方法的适配性, 但仍缺少相关研究能够解释为什么某些模型在应用特定种类的约束时效果更好. 目前的主流基座模型选择, 例如 ChatGPT、Claude 和 CodeT5 等, 主要是基于它们在常规任务上优秀的综合表现, 并未真正考虑模型与约束方法之间的交互情况. 未来可以结合模型探测技术, 分析不同约束机制产生作用时具体影响的模型局部, 定位和理解约束生效的机理; 也可以设计更细粒度的消融实验, 通过拆解现有的约束机制, 逐步分析其作用于模型的方式和规律. 通过这些研究, 可以有效预测约束机制将在哪些种类的模型上产生更好的效果, 并可以据此对模型进行针对性的调整和优化.

(2) 探究约束效果随模型能力变化的规律. 当前研究中, 一部分工作对比了多种基座模型, 初步探索了约束方

法在不同模型上的效果. 但整体而言, 目前缺少系统性的实验和分析, 能够说明约束效果是否会随着模型能力增强而提升. 例如, 对于检索增强、测试验证和模型协同校正等外置于模型的技术, 目前的实验数据显示它们均适合于能力更强的模型, 且呈现出模型能力越强效果越好的趋势; 而对于约束解码和模型训练这类生成中约束方法, 由于需要调整模型内部的参数和结构, 盲目采用大型模型很可能导致约束成本过高, 或是因约束机制破坏了原有设计导致性能上的负优化, 但针对这一问题的研究目前仅有一些零散的尝试, 缺少系统性实验来支持相关结论. 未来需要开展较大规模的实证研究, 在选定的约束机制和评价指标上评估不同能力、不同规模的基座模型应用约束时的效果, 总结数据规律的同时分析背后的机制.

6.4 针对约束机制的精细化评估

受约束代码生成技术的推广和应用离不开一个科学、系统的评估体系. 目前, 受约束代码生成方法的评估大多通过比较模型施加约束前后生成的代码在不同指标上的表现, 这种方式可以直观体现约束机制的效果, 但忽略了大量过程细节和底层机制. 未来需要从以下两个方面进行探索.

(1) 基于代码差异的约束效果分析. 目前的研究主要比较施加约束前后的指标差异, 未能关注到代码本身的差异性以及这些差异性背后的原因. 未来应考虑通过定位和分析代码的变更部分, 推断约束机制使模型行为产生了什么样的变化, 并可以基于代码差异设计更具针对性的评价指标, 衡量引入约束带来的实际收益. 具体来说, 添加约束后模型生成代码的变动可能有多种不同情况, 包括但不限于标识符修改、符号修改、数据流修改和控制结构修改. 这些修改对代码正确性、代码质量的影响及其背后的成因都有很大差异, 但在目前的研究中多数只体现为指标的数值变化, 这实际上忽略了大量有效信息. 约束机制的特点之一在于它通常会给模型提出较为明确的规范, 未来的研究应利用好这些规范信息, 对生成结果中的代码变更进行细粒度分析, 使用与规范匹配的验证工具解释每处代码修改是否遵循了规范、遵循了哪些规范, 以此提供更加具体和可解释的约束效果评价方法.

(2) 约束机制的负面效果研究. 对模型施加约束通常都会影响正常的生成过程, 导致生成代码在某些方面的表现下降, 如约束语法导致代码功能不完善. 尽管现有研究注意到了这一问题, 但缺少合适的方法和指标来对约束负面影响进行量化评估. 另外, 约束带来的负面影响具体是如何作用于模型生成过程的, 目前也缺少系统性的分析, 这导致研究者难以评估约束机制可能带来的潜在危害. 未来应系统分类各种约束对应的负面作用, 并为其设计适配的评估方法. 例如可以将现有的各类评价指标进行整合, 在评估约束方法时同时查看多个类型的指标, 判断约束对代码的整体影响, 总结出不同约束机制产生负面效果的规律. 同时, 未来应对施加约束后生成的代码进行详细分析, 理解约束对生成过程的损害具体是如何产生的. 例如针对约束机制降低代码功能性的情况, 可以结合具体的代码实例分析是哪些语句或结构的变更影响了代码功能, 并通过监测大语言模型生成代码时的解码过程, 研究约束机制的施加为何最终会导致生成的代码出现特定变化.

7 总 结

大语言模型在自动代码生成中展现出巨大潜力, 但生成的代码常存在多方面问题. 本文将受约束代码生成定义为在大语言模型代码生成各阶段通过引入约束机制来提升生成代码正确性和质量的方法, 并基于此对现有文献进行收集和分析. 本文首先总结了模型代码生成存在的问题, 在此基础上详细介绍了 3 类受约束代码生成方法, 并归纳了相应的评估体系, 最后, 指出了当前研究面临的挑战并展望了未来的研究方向. 我们期望本文的工作能为大语言模型代码生成中的约束技术提供有益参考, 推动相关方法向更可靠、更实用的方向发展.

References

[1] Chen M, Tworek J, Jun H, *et al*. Evaluating large language models trained on code. arXiv:2107.03374, 2021.

[2] Li D, Murr L. HumanEval on latest GPT models—2024. arXiv:2402.14852, 2024.

[3] Jiang JY, Wang F, Shen JS, Kim S, Kim S. A survey on large language models for code generation. arXiv:2406.00515, 2024.

[4] Guo DY, Xu CW, Duan N, Yin J, McAuley J. LongCoder: A long-range pre-trained language model for code completion. In: Proc. of the 40th Int'l Conf. on Machine Learning. Honolulu: JMLR.org, 2023. 12098–12107.

[5] Wu D, Ahmad WU, Zhang DJ, Ramanathan MK, Ma XF. Repoformer: Selective retrieval for repository-level code completion. arXiv:2403.10059, 2024.

[6] Szafraniec M, Roziere B, Leather H, Charton F, Labatut P, Synnaeve G. Code translation with compiler representations. arXiv:2207.03578, 2023.

[7] Fan ZY, Gao X, Mirchev M, Roychoudhury A, Tan SH. Automated repair of programs from large language models. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 1469–1481. [doi: 10.1109/ICSE48619.2023.00128]

[8] Zhang JL, Cambronero JP, Gulwani S, Le V, Piskac R, Soares G, Verbruggen G. PyDex: Repairing bugs in introductory Python assignments using LLMs. Proc. of the ACM on Programming Languages, 2024, 8(OOPSLA1): 1100–1124. [doi: 10.1145/3649850]

[9] Bhatia S, Gandhi T, Kumar D, Jalote P. Unit test generation using generative AI: A comparative performance analysis of autogeneration tools. In: Proc. of the 1st Int'l Workshop on Large Language Models for Code. Lisbon: ACM, 2024. 54–61. [doi: 10.1145/3643795.3648396]

[10] Cui KZ, Demirer M, Jaffe S, Musolff L, Peng SD, Salz T. The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers. Management Science, 2025. [doi: 10.1287/mnsc.2025.00535]

[11] Wang ZJ, Zhou ZJ, Song D, Huang YH, Chen SM, Ma L, Zhang TY. Towards understanding the characteristics of code generation errors made by large language models. In: Proc. of the 47th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Ottawa: IEEE, 2025. 2587–2599. [doi: 10.1109/ICSE55347.2025.00180]

[12] Zhong L, Wang ZL. Can LLM replace Stack Overflow? A study on robustness and reliability of large language model code generation. In: Proc. of the 38th AAAI Conf. on Artificial Intelligence. Vancouver: AAAI, 2024. 21841–21849. [doi: 10.1609/aaai.v38i19.30185]

[13] OpenAI. GPT-4 technical report. arXiv:2303.08774, 2024.

[14] Liu JW, Xia CS, Wang YY, Zhang LM. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In: Proc. of the 37th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2023. 21558–21572.

[15] Licorish SA, Bajpai A, Arora C, Wang FY, Tantithamthavorn K. Comparing human and LLM generated code: The jury is still out! arXiv:2501.16857, 2025.

[16] Fu YJ, Baker E, Ding Y, Chen YZ. Constrained decoding for secure code generation. arXiv:2405.00218, 2024.

[17] Wu Y, Bojanova I, Yesha Y. They know your weaknesses—Do you?: Reintroducing common weakness enumeration. CrossTalk, 2015, 28(5): 44–50.

[18] Hu ZT, Yang ZC, Liang XD, Salakhutdinov R, Xing EP. Toward controlled generation of text. In: Proc. of the 34th Int'l Conf. on Machine Learning. Sydney: JMLR.org, 2017. 1587–1596.

[19] Luo FL, Li P, Yang PC, Zhou J, Tan YT, Chang BB, Sui ZF, Sun X. Towards fine-grained text sentiment transfer. In: Proc. of the 57th Annual Meeting of the Association for Computational Linguistics. Florence: ACL, 2019. 2013–2022. [doi: 10.18653/v1/P19-1194]

[20] Dathathri S, Madotto A, Lan J, Hung J, Frank E, Molino P, Yosinski J, Liu R. Plug and play language models: A simple approach to controlled text generation. arXiv:1912.02164, 2020.

[21] Yang K, Klein D. FUDGE: Controlled text generation with future discriminators. arXiv:2104.05218, 2021.

[22] Wuebker J, Green S, DeNero J, Hasan S, Luong MT. Models and inference for prefix-constrained machine translation. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics (Vol. 1: Long Papers). Berlin: ACL, 2016. 66–75. [doi: 10.18653/v1/P16-1007]

[23] Hokamp C, Liu Q. Lexically constrained decoding for sequence generation using grid beam search. arXiv:1704.07138, 2017.

[24] Yetiştiren B, Özsoy I, Ayerdem M, Tüzün E. Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. arXiv:2304.10778, 2023.

[25] Liu F, Liu Y, Shi L, Huang HK, Wang RF, Yang Z, Zhang L, Li ZQ, Ma YC. Exploring and evaluating hallucinations in LLM-powered code generation. arXiv:2404.00971, 2024.

[26] Kharma M, Choi S, AlKhanafseh M, Mohaisen D. Security and quality in LLM-generated code: A multi-language, multi-model analysis. arXiv:2502.01853, 2025.

[27] Tosi D. Studying the quality of source code generated by different AI generative engines: An empirical evaluation. Future Internet, 2024, 16(6): 188. [doi: 10.3390/fi16060188]

[28] Ray PP. ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope. Internet of Things and Cyber-physical Systems, 2023, 3: 121–154. [doi: 10.1016/j.iotcps.2023.04.003]

[29] Singh SK, Kumar S, Mehra PS. Chat GPT & Google Bard AI: A review. In: Proc. of the 2023 Int'l Conf. on IoT, Communication and Automation Technology (ICICAT). Gorakhpur: IEEE, 2023. 1–6. [doi: 10.1109/ICICAT57735.2023.10263706]

[30] Troshin S, Chirkova N. Probing pretrained models of source code. arXiv:2202.08975, 2022.
[31] Gu XD, Chen M, Lin YL, Hu YH, Zhang HY, Wan CC, Wei Z, Xu Y, Wang JH. On the effectiveness of large language models in domain-specific code generation. ACM Trans. on Software Engineering and Methodology, 2025, 34(3): 78. [doi: 10.1145/3697012]
[32] Lian XL, Wang SS, Ma JP, Tan X, Liu F, Shi L, Gao CY, Zhang L. Imperfect code generation: Uncovering weaknesses in automatic code generation by large language models. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering: Companion Proc. Lisbon: ACM, 2024. 422–423. [doi: 10.1145/3639478.3643081]
[33] Du XY, Liu MW, Wang KX, Wang HL, Liu JW, Chen YX, Feng JY, Sha CF, Peng X, Lou YL. ClassEval: A manually-crafted benchmark for evaluating LLMs on class-level code generation. arXiv:2308.01861, 2023.
[34] Campbell GA, Papapetrou PP. SonarQube in Action. New York: Manning Publications Co., 2013.
[35] Ebert C, Cain J, Antoniol G, Counsell S, Laplante P. Cyclomatic complexity. IEEE Software, 2016, 33(6): 27–29. [doi: 10.1109/MS.2016.147]
[36] Campbell GA. Cognitive complexity: An overview and evaluation. In: Proc. of the 2018 Int'l Conf. on Technical Debt. Gothenburg: ACM, 2018. 57–58. [doi: 10.1145/3194164.3194186]
[37] Li ZY, Avgeriou P, Liang P. A systematic mapping study on technical debt and its management. Journal of Systems and Software, 2015, 101: 193–220. [doi: 10.1016/j.jss.2014.12.027]
[38] Scalabrino S, Linares-Vásquez M, Oliveto R, Poshyvanyk D. A comprehensive model for code readability. Journal of Software: Evolution and Process, 2018, 30(6): e1958. [doi: 10.1002/smr.1958]
[39] Siddiq ML, Majumder SH, Mim MR, Jajodia S, Santos JCS. An empirical study of code smells in Transformer-based code generation techniques. In: Proc. of the 22nd IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM). Limassol: IEEE, 2022. 71–82. [doi: 10.1109/SCAM55253.2022.00014]
[40] Gulabovska H, Porkoláb Z. Survey on static analysis tools of Python programs. In: Proc. of the 8th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications. Ohrid: CEUR-WS.org, 2019. 3:1–3:10.
[41] Khoury R, Avila AR, Brunelle J, Camara BM. How secure is code generated by ChatGPT? In: Proc. of the 2023 IEEE Int'l Conf. on Systems, Man, and Cybernetics (SMC). Honolulu: IEEE, 2023. 2445–2451. [doi: 10.1109/SMC53992.2023.10394237]
[42] Vasilescu B, Filkov V, Serebrenik A. StackOverflow and GitHub: Associations between software development and crowdsourced knowledge. In: Proc. of the 2013 Int'l Conf. on Social Computing. Alexandria: IEEE, 2013. 188–195. [doi: 10.1109/SocialCom.2013.35]
[43] Shatnawi AS, Al-Duwairi B, Samarneh AA. Comprehensive empirical study of Python JWT libraries. Procedia Computer Science, 2024, 238: 827–832. [doi: 10.1016/j.procs.2024.06.099]
[44] Dai SC, Xu J, Tao GH. A comprehensive study of LLM secure code generation. arXiv:2503.15554, 2025.
[45] Schaad A, Götz S, Binder D. You still have to study on the security of LLM generated code. In: Proc. of the 40th IFIP Int'l Conf. on ICT Systems Security and Privacy Protection. Maribor: Springer, 2025. 111–124. [doi: 10.1007/978-3-031-92886-4_8]
[46] Huang D, Qing YH, Shang WY, Cui HM, Zhang JM. EffiBench: Benchmarking the efficiency of automatically generated code. In: Proc. of the 38th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2024. 11506–11544. [doi: 10.52202/079017-0367]
[47] Sikand S, Mehra R, Sharma VS, Kaulgud V, Podder S, Burden AP. Do generative AI tools ensure green code? An investigative study. In: Proc. of the 2nd Int'l Workshop on Responsible AI Engineering. Lisbon: ACM, 2024. 52–55. [doi: 10.1145/3643691.3648588]
[48] Niu C, Zhang T, Li CY, Luo B, Ng V. On evaluating the efficiency of source code generated by LLMs. In: Proc. of the 1st IEEE/ACM Int'l Conf. on AI Foundation Models and Software Engineering. Lisbon: ACM, 2024. 103–107. [doi: 10.1145/3650105.3652295]
[49] Austin J, Odena A, Nye M, Bosma M, Michalewski H, Dohan D, Jiang E, Cai C, Terry M, Le Q, Sutton C. Program synthesis with large language models. arXiv:2108.07732, 2021.
[50] Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA. The gem5 simulator. ACM SIGARCH Computer Architecture News, 2011, 39(2): 1–7. [doi: 10.1145/2024716.2024718]
[51] Du MZ, Luu AT, Ji B, Liu Q, Ng SK. Mercury: A code efficiency benchmark for code large language models. arXiv:2402.07844, 2024.
[52] Solovyeva L, Weidmann S, Castor F. AI-powered, but power-hungry? Energy efficiency of LLM-generated code. In: Proc. of the 2nd IEEE/ACM Int'l Conf. on AI Foundation Models and Software Engineering (Forge). Ottawa: IEEE, 2025. 49–60. [doi: 10.1109/Forge66646.2025.00012]
[53] Yin PC, Deng BW, Chen E, Vasilescu B, Neubig G. Learning to mine aligned code and natural language pairs from Atack Overflow. In: Proc. of the 15th Int'l Conf. on Mining Software Repositories. Gothenburg: ACM, 2018. 476–486. [doi: 10.1145/3196398.3196408]

[54] Ahmad WU, Chakraborty S, Ray B, Chang KW. Unified pre-training for program understanding and generation. arXiv:2103.06333, 2021.

[55] Pearce H, Tan B, Ahmad B, Karri R, Dolan-Gavitt B. Examining zero-shot vulnerability repair with large language models. In: Proc. of the 2023 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2023. 2339–2356. [doi: 10.1109/SP46215.2023.10179324]

[56] Res J, Homoliak I, Perešíni M, Smrčka A, Malinka K, Hanacek P. Enhancing security of AI-based code synthesis with GitHub Copilot via cheap and efficient prompt-engineering. arXiv:2403.12671, 2024.

[57] Wu OT, Chan FKS, Zhang ZH, Law YN, Drescher B, Lai ESB. Prompt selection and augmentation for few examples code generation in large language model and its application in robotics control. arXiv:2403.12999, 2024.

[58] Nazzal M, Khalil I, Khreishah A, Phan N. PromSec: Prompt optimization for secure generation of functional source code with large language models (LLMs). In: Proc. of the 2024 ACM SIGSAC Conf. on Computer and Communications Security. Salt Lake City: ACM, 2024. 2266–2280. [doi: 10.1145/3658644.3690298]

[59] Wang HW, Wang J, Wang JL, Zhao M, Zhang WN, Zhang FZ, Xie X, Guo MY. GraphGAN: Graph representation learning with generative adversarial nets. In: Proc. of the 32nd AAAI Conf. on Artificial Intelligence. New Orleans: AAAI, 2018. 2508–2515. [doi: 10.1609/aaai.v32i1.11872]

[60] Tihanyi N, Jain R, Charalambous Y, Ferrag MA, Sun YC, Cordeiro LC. A new era in software security: Towards self-healing software via large language models and formal verification. arXiv:2305.14752, 2024.

[61] Gadelha MR, Monteiro FR, Morse J, Cordeiro LC, Fischer B, Nicole DA. ESBMC 5.0: An industrial-strength C model checker. In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. Montpellier: ACM, 2018. 888–891. [doi: 10.1145/3238147.3240481]

[62] Shi TN, He JX, Wang Z, Li HW, Wu LY, Guo WB, Song D. Progent: Programmable privilege control for LLM agents. arXiv: 2504.11703, 2025.

[63] Parvez MR, Ahmad WU, Chakraborty S, Ray B, Chang KW. Retrieval augmented code generation and summarization. arXiv: 2108.11601, 2021.

[64] Chen JK, Hu X, Li ZH, Gao CY, Xia X, Lo D. Code search is all you need? Improving code suggestions with code search. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 73. [doi: 10.1145/3597503.3639085]

[65] Lu S, Duan N, Han H, Guo DY, Hwang SW, Svyatkovskiy A. ReACC: A retrieval-augmented code completion framework. arXiv:2203.07722, 2022.

[66] Zhou SY, Alon U, Xu FF, Wang ZR, Jiang ZB, Neubig G. DocPrompting: Generating code by retrieving the docs. arXiv:2207.05987, 2023.

[67] Koziolek H, Grüner S, Hark R, Ashiwal V, Linsbauer S, Eskandani N. LLM-based and retrieval-augmented control code generation. In: Proc. of the 1st Int'l Workshop on Large Language Models for Code. Lisbon: ACM, 2024. 22–29. [doi: 10.1145/3643795.3648384]

[68] Chen ZT, Xiong ZY, Yao XS, Glassman E. Sketch then generate: Providing incremental user feedback and guiding LLM code generation through language-oriented code sketches. arXiv:2405.03998, 2024.

[69] Ma YW, Yu Y, Li SS, Jiang Y, Guo Y, Zhang YL, Xie YT, Liao XK. Bridging code semantic and LLMs: Semantic chain-of-thought prompting for code generation. arXiv:2310.10698, 2023.

[70] Stahlberg F. Neural machine translation: A review. Journal of Artificial Intelligence Research, 2020, 69: 343–418. [doi: 10.1613/jair.1.12007]

[71] Steinbiss V, Tran BH, Ney H. Improvements in beam search. In: Proc. of the 3rd Int'l Conf. on Spoken Language Processing (ICSLP 94). Yokohama, 1994. 2143–2146.

[72] Scholak T, Schucher N, Bahdanau D. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. arXiv:2109.05093, 2021.

[73] Ugare S, Suresh T, Kang H, Misailovic S, Singh G. SynCode: LLM generation with grammar augmentation. arXiv:2403.01632, 2024.

[74] Poesia G, Polozov O, Le V, Tiwari A, Soares G, Meek C, Gulwani S. Synchromesh: Reliable code generation from pre-trained language models. arXiv:2201.11227, 2022.

[75] Zhang XY, Zhou Y, Yang G, Chen TL. Syntax-aware retrieval augmented code generation. In: Findings of the Association for Computational Linguistics: EMNLP 2023. Singapore: ACL, 2023. 1291–1302. [doi: 10.18653/v1/2023.findings-emnlp.90]

[76] Ma WC, Ni ZH, Cao K, Li X, Chin S. Seq2Tree: A tree-structured extension of LSTM network. In: Proc. of the 31st Conference on Neural Information Processing Systems. Long Beach: OpenReview.net, 2017.

[77] Park K, Wang JY, Berg-Kirkpatrick T, Polikarpova N, D'Antoni L. Grammar-aligned decoding. In: Proc. of the 38th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2024. 24547–24568.

[78] Ilager S, Briem LF, Brandic I. GREEN-CODE: Optimizing energy efficiency in large language models for code generation. arXiv:2501.11006, 2025.

[79] Storhaug A, Li JY, Hu TY. Efficient avoidance of vulnerabilities in auto-completed smart contract code using vulnerability-constrained decoding. In: Proc. of the 34th IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Florence: IEEE, 2023. 683–693. [doi: 10.1109/ISSRE59848.2023.00035]

[80] Pimparkhede S, Kammakomati M, Tamilselvam S, Kumar P, Kumar AP, Bhattacharyya P. DocCGen: Document-based controlled code generation. arXiv:2406.11925, 2024.

[81] Mernik M, Heering J, Sloane AM. When and how to develop domain-specific languages. ACM Computing Surveys (CSUR), 2005, 37(4): 316–344. [doi: 10.1145/1118890.1118892]

[82] Li ZK, Liu MW, Li AJ, He KF, Wang YL, Peng X, Zheng ZB. Enhancing the robustness of LLM-generated code: Empirical study and framework. arXiv:2503.20197, 2025.

[83] Mündler N, He JX, Wang H, Sen K, Song D, Vechev M. Type-constrained code generation with language models. Proc. of the ACM on Programming Languages, 2025, 9(PLDI): 601–626. [doi: 10.1145/3729274]

[84] Banerjee D, Suresh T, Ugare S, Misailovic S, Singh G. CRANE: Reasoning with constrained LLM generation. arXiv:2502.09061, 2025.

[85] Tipirneni S, Zhu M, Reddy CK. StructCoder: Structure-aware Transformer for code generation. ACM Trans. on Knowledge Discovery from Data, 2024, 18(3): 70. [doi: 10.1145/3636430]

[86] Han K, Xiao A, Wu EH, Guo JY, Xu CJ, Wang YH. Transformer in Transformer. In: Proc. of the 35th Int'l Conf. on Neural Information Processing Systems. Curran Associates Inc., 2021. 15908–15919.

[87] Lu S, Guo DY, Ren S, Huang JJ, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang DX, Tang DY, Li G, Zhou LD, Shou LJ, Zhou L, Tufano M, Gong M, Zhou M, Duan N, Sundaresan N, Deng SK, Fu SY, Liu SJ. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664, 2021.

[88] He JX, Vechev M. Large language models for code: Security hardening and adversarial testing. In: Proc. of the 2023 ACM SIGSAC Conf. on Computer and Communications Security. Copenhagen: ACM, 2023. 1865–1879. [doi: 10.1145/3576915.3623175]

[89] He JX, Vero M, Krasnopolska G, Vechev M. Instruction tuning for secure code generation. arXiv:2402.09497, 2024.

[90] Jain N, Zhang TJ, Chiang WL, Gonzalez J, Sen K, Stoica I. Improving code style for accurate code generation. In: Proc. of the 2023 NeurIPS Workshop on Synthetic Data Generation with Generative AI. New Orleans: OpenReview.net, 2023.

[91] Hendrycks D, Basart S, Kadavath S, Mazeika M, Arora A, Guo E, Burns C, Puranik S, He H, Song D, Steinhardt J. Measuring coding challenge competence with APPS. arXiv:2105.09938, 2021.

[92] Li YJ, Choi D, Chung J, *et al*. Competition-level code generation with AlphaCode. Science, 2022, 378(6624): 1092–1097. [doi: 10.1126/science.abq1158]

[93] Sun ZS, Du XN, Yang Z, Li L, Lo D. AI coders are among us: Rethinking programming language grammar towards efficient code generation. In: Proc. of the 33rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Vienna: ACM, 2024. 1124–1136. [doi: 10.1145/3650212.3680347]

[94] Islam NT, Khoury J, Seong A, Karkevandi MB, De La Torre Parra G, Bou-Harb E, Najafirad P. LLM-powered code vulnerability repair with reinforcement learning and semantic reward. arXiv:2401.03374, 2024.

[95] Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O. Proximal policy optimization algorithms. arXiv:1707.06347, 2017.

[96] Liu JT, Zhu YQ, Xiao KW, Fu Q, Han X, Yang W, Ye DH. RLTF: Reinforcement learning from unit test feedback. arXiv:2307.04349, 2023.

[97] He X, Li D, Wen H, Zhu YH, Liu C, Yan M, Zhang HY. CoSEFA: An LLM-based programming assistant for secure code generation via supervised co-decoding. In: Proc. of the 33rd ACM Int'l Conf. on the Foundations of Software Engineering. Trondheim: ACM, 2025. 1198–1202. [doi: 10.1145/3696630.3728609]

[98] Hu XY, Kuang K, Sun JK, Yang HX, Wu F. Leveraging print debugging to improve code generation in large language models. arXiv:2401.05319, 2024.

[99] Tian Z, Chen JJ, Zhang XY. Fixing large language models' specification misunderstanding for better code generation. In: Proc. of the 47th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Ottawa: IEEE, 2025. 1514–1526. [doi: 10.1109/ICSE55347.2025.00108]

[100] Chen B, Zhang FJ, Nguyen A, Zan DG, Lin ZQ, Lou JG, Chen WZ. CodeT: Code generation with generated tests. arXiv:2207.10397, 2022.

[101] Jin YY, Xu KZ, Li H, Han XT, Zhou YM, Li C, Bai J. ReVeal: Self-evolving code agents via iterative generation-verification. arXiv:2506.11442, 2025.

[102] Dong YH, Jiang X, Jin Z, Li G. Self-collaboration code generation via ChatGPT. ACM Trans. on Software Engineering and

Methodology, 2024, 33(7): 189. [doi: 10.1145/3672459]

[103] Zhang TY, Yu T, Hashimoto TB, Lewis M, Yih WT, Fried D, Wang SI. Coder reviewer reranking for code generation. In: Proc. of the 40th Int'l Conf. on Machine Learning. Honolulu: JMLR.org, 2023. 41832–41846.

[104] Bahl L, Brown P, De Souza P, Mercer R. Maximum mutual information estimation of hidden Markov model parameters for speech recognition. In: Proc. of the 1986 IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing. Tokyo: IEEE, 1986. 49–52. [doi: 10.1109/ICASSP.1986.1169179]

[105] Nunez A, Islam NT, Jha SK, Najafirad P. AutoSafeCoder: A multi-agent framework for securing LLM code generation through static analysis and fuzz testing. arXiv:2409.10737, 2024.

[106] Huang D, Zhang JM, Luck M, Bu QW, Qing YH, Cui HM. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. arXiv:2312.13010, 2024.

[107] Bayat A, Abate A, Ozay N, Jungers RM. LLM-enhanced symbolic control for safety-critical applications. arXiv:2505.11077, 2025.

[108] Amini A, Gabriel S, Lin P, Koncel-Kedziorski R, Choi Y, Hajishirzi H. MathQA: Towards interpretable math word problem solving with operation-based formalisms. arXiv:1905.13319, 2019.

[109] Dai JB, Lu JQ, Feng YL, Zeng GT, Ruan RJ, Cheng M, Huang D, Tan HC, Guo ZJ. MHPP: Exploring the capabilities and limitations of language models beyond basic code generation. arXiv:2405.11430, 2025.

[110] Cao JL, Chen ZY, Wu JR, Cheung SC, Xu C. JavaBench: A benchmark of object-oriented code generation for evaluating large language models. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Automated Software Engineering. Sacramento: ACM, 2024. 870–882. [doi: 10.1145/3691620.3695470]

[111] Jimenez CE, Yang J, Wettig A, Yao SY, Pei KX, Press O, Narasimhan K. SWE-bench: Can language models resolve real-world GitHub issues? arXiv:2310.06770, 2024.

[112] Yu H, Shen B, Ran DZ, Zhang JX, Zhang Q, Ma YC, Liang GT, Li Y, Wang QX, Xie T. CoderEval: A benchmark of pragmatic code generation with generative pre-trained models. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 37. [doi: 10.1145/3597503.3623316]

[113] Vijayaraghavan P, Shi LY, Ambrogio S, Mackin C, Nitsure A, Beymer D, Degan E. VHDL-eval: A framework for evaluating large language models in VHDL code generation. In: Proc. of the 2024 IEEE LLM Aided Design Workshop (LAD). San Jose: IEEE, 2024. 1–6. [doi: 10.1109/LAD62341.2024.10691836]

[114] Liu MJ, Pinckney N, Khailany B, Ren HX. Invited Paper: VerilogEval: Evaluating large language models for verilog code generation. In: Proc. of the 2023 IEEE/ACM Int'l Conf. on Computer Aided Design (ICCAD). San Francisco: IEEE, 2023. 1–8. [doi: 10.1109/ICCAD57390.2023.10323812]

[115] Yan WX, Liu HT, Wang YK, Li YZ, Chen Q, Wang W, Lin TY, Zhao WS, Zhu L, Sundaram H, Deng SG. CodeScope: An execution-based multilingual multitask multidimensional benchmark for evaluating LLMs on code understanding and generation. arXiv:2311.08588, 2024.

[116] Liu JW, Xie SR, Wang JH, Wei YX, Ding YF, Zhang LM. Evaluating language models for efficient code generation. arXiv:2408.06450, 2024.

[117] Wang JX, Luo XT, Cao LW, He HK, Huang HL, Xie JY, Jatowt A, Cai Y. Is your AI-generated code really safe? Evaluating large language models on secure code generation with CodeSecEval. arXiv:2407.02395, 2024.

[118] Siddiq ML, Santos JCS. SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. In: Proc. of the 1st Int'l Workshop on Mining Software Repositories Applications for Privacy and Security. Singapore: ACM, 2022. 29–33. [doi: 10.1145/3549035.3561184]

[119] Pearce H, Ahmad B, Tan B, Dolan-Gavitt B, Karri R. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. Communications of the ACM, 2025, 68(2): 96–105. [doi: 10.1145/3610721]

[120] Zheng JS, Cao BX, Ma ZZ, Pan RT, Lin HY, Lu YJ, Han XP, Sun L. Beyond correctness: Benchmarking multi-dimensional code generation for large language models. arXiv:2407.11470, 2024.

[121] Guo DY, Zhu QH, Yang DJ, Xie ZD, Dong K, Zhang WT, Chen GT, Bi X, Wu Y, Li YK, Luo FL, Xiong YF, Liang WF. DeepSeek-Coder: When the large language model meets programming—The rise of code intelligence. arXiv:2401.14196, 2024.

[122] Papineni K, Roukos S, Ward T, Zhu WJ. BLEU: A method for automatic evaluation of machine translation. In: Proc. of the 40th Annual Meeting of the Association for Computational Linguistics. Philadelphia: ACL, 2002. 311–318. [doi: 10.3115/1073083.1073135]

[123] Evtikhiev M, Bogomolov E, Sokolov Y, Bryksin T. Out of the BLEU: How should we assess quality of the code generation models? Journal of Systems and Software, 2023, 203: 111741. [doi: 10.1016/j.jss.2023.111741]

[124] Ling W, Grefenstette E, Hermann KM, Kočiský T, Senior A, Wang FM, Blunsom P. Latent predictor networks for code generation. arXiv:1603.06744, 2016.

[125] Popović M. chrF: Character n-gram F-score for automatic MT evaluation. In: Proc. of the 10th Workshop on Statistical Machine Translation. Lisbon: ACL, 2015. 392–395. [doi: 10.18653/v1/W15-3049]

[126] Lin CY. ROUGE: A package for automatic evaluation of summaries. In: Text Summarization Branches Out. Barcelona: ACL, 2004. 74–81.

[127] Ren S, Guo DY, Lu S, Zhou L, Liu SJ, Tang DY, Sundaresan N, Zhou M, Blanco A, Ma S. CodeBLEU: A method for automatic evaluation of code synthesis. arXiv:2009.10297, 2020.

[128] Tran N, Tran H, Nguyen S, Nguyen H, Nguyen T. Does BLEU score work for code migration? In: Proc. of the 27th IEEE/ACM Int'l Conf. on Program Comprehension (ICPC). Montreal: IEEE, 2019. 165–176. [doi: 10.1109/ICPC.2019.00034]

[129] Vartziotis T, Dellatolas I, Dasoulas G, Schmidt M, Schneider F, Hoffmann T, Kotsopoulos S, Keckeisen M. Learn to code sustainably: An empirical study on LLM-based green code generation. arXiv:2403.03344, 2024.

[130] Peng JJ, Cui LY, Huang KL, Yang JF, Ray B. CWEval: Outcome-driven evaluation on functionality and security of LLM code generation. In: Proc. of the 2025 IEEE/ACM Int'l Workshop on Large Language Models for Code (LLM4Code). Ottawa: IEEE, 2025. 33–40. [doi: 10.1109/LLM4Code66737.2025.00009]

作者简介

李宇轩, 博士生, CCF 学生会员, 主要研究领域为软件工程, 软件复用, 智能软件开发.

邹艳珍, 博士, 副教授, CCF 专业会员, 主要研究领域为软件工程, 软件复用, 知识图谱, 智能软件开发.

王玥, 博士生, 主要研究领域为软件工程, 软件重构, 智能软件开发.

谢冰, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件工程, 形式化方法, 软件复用, 智能软件开发.