清 华 大 学

综 合 论 文 训 练

题目：清华大学学位论文 LaTeX 模板使用示例文档 v7.5.2

系　　别：软件学院

专　　业：软件工程

姓　　名：陈敬文

指导教师：刘　璘　教授

2025 年 5 月 8 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____ 导师签名：_____ 日　期：_____

中文摘要

论文的摘要是对论文研究内容和成果的高度概括。摘要应对论文所研究的问题及其研究目的进行描述，对研究方法和过程进行简单介绍，对研究成果和所得结论进行概括。摘要应具有独立性和自明性，其内容应包含与论文全文同等量的主要信息。使读者即使不阅读全文，通过摘要就能了解论文的总体内容和主要成果。

论文摘要的书写应力求精确、简明。切忌写成对论文书写内容进行提要的形式，尤其要避免“第 1 章……；第 2 章……；……”这种或类似的陈述方式。

关键词是为了文献标引工作、用以表示全文主要内容信息的单词或术语。关键词不超过 5 个，每个关键词中间用分号分隔。

关键词：关键词 1；关键词 2；关键词 3；关键词 4；关键词 5

ABSTRACT

An abstract of a dissertation is a summary and extraction of research work and contributions. Included in an abstract should be description of research topic and research objective, brief introduction to methodology and research process, and summary of conclusion and contributions of the research. An abstract should be characterized by independence and clarity and carry identical information with the dissertation. It should be such that the general idea and major contributions of the dissertation are conveyed without reading the dissertation.

An abstract should be concise and to the point. It is a misunderstanding to make an abstract an outline of the dissertation and words “the first chapter”, “the second chapter” and the like should be avoided in the abstract.

Keywords are terms used in a dissertation for indexing, reflecting core information of the dissertation. An abstract may contain a maximum of 5 keywords, with semi-colons used in between to separate one another.

Keywords: keyword 1; keyword 2; keyword 3; keyword 4; keyword 5

目　　录

插图索引 VIII

表格索引 X

第 1 章　引言 1

1.1　选题背景 1

1.2　问题提出 1

1.3　本文结构 2

第 2 章　相关工作 3

2.1　LLM 代码生成的上下文理解能力 3

2.2　支持实时反馈的交互式开发工具 4

2.2.1　GitHub Copilot (2021) 5

2.2.2　Cursor (2023) 5

2.2.3　ChatGPT Code Interpreter (2023) 5

2.2.4　bolt.new and bolt.diy (2024) 5

2.2.5　工具功能对比 6

2.3　LLM 与现代软件工程实践的融合 7

2.3.1　测试驱动开发（TDD）与 LLM 7

2.3.2　持续集成与交付（CI/CD） 7

2.3.3　API 优先开发（API-First） 8

2.3.4　多代理协同与软件工程流程 8

第 3 章　bolt.SE 介绍 10

3.1　选择 bolt 的原因 10

3.2　系统整体流程 10

3.3　UI 与操作流程介绍 11

3.3.1　用户需求输入与模型选择 11

3.3.2　代码生成与编辑 11

3.3.3　实时预览与调试 11

3.4 LLM 介绍 12
3.4.1 多 LLM 支持及其好处 12
3.4.2 大语言模型的超长文本处理能力 12
3.5 MessageParser 12
3.5.1 Prompt 定义的 XML 结构 13
3.5.2 结构化数据的优势 14
3.6 WebContainer 简介 15
3.6.1 实现原理 15
3.6.2 在 bolt 系统中的应用 15
3.7 改进 15
第 4 章 API 优先开发 16
4.1 API 优先开发的概念与意义 16
4.1.1 API 优先开发在软件工程中的优势 16
4.1.2 API 优先开发对 LLM 驱动软件开发的重要性 17
4.2 OpenAPI 规范与 API 设计 17
4.2.1 OpenAPI 规范简介 18
4.2.2 使用 OpenAPI 规范进行 API 设计的优势 19
4.3 bolt.se 中的 APIActions 实现 20
4.3.1 APIActions 与 Token 效率优化 20
4.3.2 整体架构设计与模块划分 21
4.4 实例应用场景 24
4.4.1 添加与配置 APIActions 24
4.4.2 自然语言触发任务规划与代码生成 24
4.4.3 代码结构与依赖初始化 26
4.4.4 交互效果与 API 调用验证 26
4.4.5 与传统开发方式对比 26
第 5 章 测试驱动开发 29
5.1 测试驱动开发的概念与意义 29
5.1.1 TDD 在软件工程中的优势 29
5.1.2 TDD 对 LLM 驱动软件开发的重要性 30

5.2 Jest 测试框架与 TDD 实践 30
5.2.1 Jest 测试框架简介 30
5.2.2 TDD 的实践方法与工作流程 31
5.3 bolt.se 中的测试功能实现 32
5.3.1 整体架构设计与模块划分 32
5.3.2 测试解析与可视化 35
5.3.3 测试与 LLM 集成机制 35
5.4 实例应用场景 36
5.4.1 创建和编辑测试 36
5.4.2 使用测试引导代码生成 37
5.4.3 测试执行与验证 37
5.5 TDD 与传统开发方式对比 38
5.6 实例应用场景 39
5.6.1 插入 Jest 测试代码与需求声明 39
5.6.2 首次执行——绿灯 40
5.6.3 引入红灯：提高规格造成断言失败 41
5.6.4 模型修复——重新转绿 41
5.6.5 重构与可视化预览 42
参考文献 43
附录 A 外文资料的书面翻译 45
附录 B 补充内容 75
致 谢 77
声 明 79
在学期间参加课题的研究成果 81

插图索引

图 3.1 bolt.SE 代码生成时序图 11
图 4.1 API 使用工作流程图：展示了用户创建、编辑和使用 API 的完整流程，以及 API 信息如何在系统各组件间传递 21
图 4.2 API 数据模型类图：描述了系统中 API 定义的核心数据结构及其关系，包括 API 操作、认证方式和存储管理 23
图 4.3 添加 NWS Weather API：通过粘贴 OpenAPI 规范定义并配置端点与认证方式。 24
图 4.4 APIActions 总览界面：展示所有注册的 API、认证方式与可用操作数。25
图 4.5 对话框中勾选相关 API，配合自然语言指令触发任务生成。 25
图 4.6 系统生成项目结构与依赖安装命令，覆盖前端组件与服务调用逻辑。 26
图 4.7 用户请求”show me a cute dog”，系统调用 The Dog API 返回图片。 27
图 4.8 询问天气时，系统识别意图并调用 NWS Weather API 提供实时天气信息。 27
图 5.1 测试功能流程图：展示了用户创建、编辑和使用测试的完整流程，以及测试信息如何在系统各组件间传递 33
图 5.2 测试数据模型类图：描述了系统中测试定义的核心数据结构及其关系，包括测试代码项、测试用例、测试套件和用户界面组件 34
图 5.3 在 AddTestModal 中粘贴 `calculator_functions.test.js`。系统解析出 *Calculator Functions* 测试套件与 12 条断言，以树形结构展示。39
图 5.4 随后在聊天窗口输入自然语言需求“Build a calculator”。系统自动把测试文件和指导语注入同一条提示，为 LLM 提供明确的功能规格。 40
图 5.5 LLM 依据测试生成 `calculator.js` 等文件。运行 Vitest 后 12 条断言全部通过，终端显示绿灯状态。 40
图 5.6 修改断言后再次运行测试，出现 1 条失败记录。终端输出红色标记，进入红灯阶段。 41
图 5.7 bolt.se 将失败堆栈回传给 LLM。模型识别出 `add` 函数需要对 (2, 3) 做特殊处理，因此生成补丁并自动应用到 `calculator.js`。 41
图 5.8 补丁生效后测试套件重新全部通过。 42

图 5.9　点击 Preview 面板即可实时操作计算器界面。当开发者进行后续重构（如提取公共运算模块或优化样式）时，只要测试继续通过，就能确保行为未被破坏。 …… 42

表格索引

表 2.1　典型 LLM 辅助开发工具功能对比 ……6

主要符号表

LLM	大型语言模型（Large Language Model）
IDE	集成开发环境（Integrated Development Environment）
CI/CD	持续集成/持续部署（Continuous Integration/Continuous Deployment）
TDD	测试驱动开发（Test-Driven Development）
API	应用程序编程接口（Application Programming Interface）
API-first	API 优先开发（API-First Development）
CI/CD	持续集成/持续部署（Continuous Integration/Continuous Deployment）

第 1 章　引言

1.1　选题背景

近年来，大语言模型（Large Language Models, LLMs）在人工智能领域取得了突破性进展，显著推动了软件开发技术与方法论的革新。这些模型具备强大的自然语言处理和代码生成能力，能够将用户描述的需求快速而准确地转化为代码，从而极大降低了软件开发领域的技术进入壁垒。特别是以 GitHub Copilot 为代表的 LLM 辅助开发工具，已展现出令人瞩目的代码自动生成与辅助编程性能，在提升开发效率和代码生产力方面取得了积极的初步成果。然而，尽管当前 LLM 工具在特定编程任务中表现突出，仍然存在诸多技术瓶颈制约其更广泛和更深入的应用。例如，这些工具往往缺乏对复杂代码仓库与运行环境上下文的有效感知与理解，进而难以实现与既有开发环境的无缝衔接；同时，由于缺乏完善的实时代码执行及在线预览机制，开发者难以即时验证生成代码的准确性和适用性；此外，这类工具普遍缺乏对完整软件工程流程的系统性支持，如体系架构设计、自动化测试、持续集成与部署等关键环节。因此，探索如何有效融合 LLMs 的先进特性与软件工程的最佳实践，建立系统化的集成工具链，以应对上述挑战，已成为当前软件工程领域的重要研究课题。

1.2　问题提出

尽管已有的 LLM 辅助编程工具在实际应用中初步证明了其潜力，但仍然存在多个关键挑战亟待解决。首先，在处理大型与复杂的软件代码库时，目前的 LLM 工具普遍面临对代码上下文环境理解不足的问题。这种不足具体体现在对于代码模块间接口、依赖关系、命名规范、编码标准以及系统整体架构设计约定等信息的识别与理解不够充分，从而导致生成的代码难以与既有环境实现高效融合，降低了生成代码的实际应用价值与长期可维护性。其次，现有工具缺乏完整、成熟的网页端实时代码运行及预览能力，使开发人员难以实时直观地观察和验证生成代码的具体表现及效果，极大限制了开发过程的敏捷性与交互性。此外，目前 LLM 工具的设计尚未有效整合广泛接受的现代软件工程实践，导致开发流程缺乏系统化和自动化的管理机制，进而削弱了代码质量保障和开发过程掌控的有效性。

例如，API 优先开发（API-First）、测试驱动开发（TDD）、数据库自动化配置，以及持续集成与持续交付（CI/CD）等先进实践，均是本研究所重点关注的软件工程流程和最佳实践。

基于以上现实问题，本研究提出了一种面向无代码基础用户与编程初学者的交互式软件开发环境——bolt.SE。其中，“bolt”来源于开源项目 bolt.diy，bolt.diy 是一款基于 Web 技术构建的开源开发平台，旨在提供全栈开发环境，并支持大语言模型集成，体现了该平台的技术基础，“SE”则表示软件工程（Software Engineering），突出该环境对软件工程领域方法论的深度融合与系统化支持。该环境旨在将大语言模型的自然语言处理与代码生成优势，与现代软件工程领域公认的最佳实践深度融合，进而实现更加高效且易用的 LLM 驱动的软件原型开发工具。具体而言，bolt.SE 将在上下文理解机制、代码实时执行与预览功能、软件工程方法论的自动化与系统化支持等方面进行重点突破，力图有效解决现存 LLM 工具的上述挑战，最终显著提高软件开发的生产效率与代码质量。

1.3 本文结构

TODO

第 2 章　相关工作

近年来，大型语言模型（Large Language Models, LLM）在代码生成领域取得了飞跃式发展。OpenAI 的 Codex 模型（Chen 等人，2021[1]）和 GPT-3.5/4 系列（OpenAI，2022–2023[2]）展示了从自然语言描述自动生成代码的能力，引发了软件开发范式变革的讨论[3]. 2022 年以来，学术界和工业界涌现出许多专注于代码的 LLM，如 Google 的 AlphaCode（Li 等人，2022[4]）、Meta 的 CodeLlama（Roziere 等人，2023[5]）以及开源社区的 StarCoder（Li 等人，2023[6]）等[3]. 这些模型在函数级别的编程挑战（如 LeetCode、HumanEval）上已经能达到甚至超越人类平均水平[7]。然而，将 LLM 真正融入软件工程实践，还面临着上下文理解、交互反馈和流程集成等多方面挑战。本文围绕上述三个方面，对 2022 年后的相关研究工作进行综述，总结构建 *bolt.SE* 这类交互式代码生成开发环境的最新进展。

首先，我们关注 LLM 在复杂代码场景下的上下文理解能力，特别是在大型代码库中的表现，如对跨模块接口、依赖关系和体系结构的感知。接着，我们探讨支持实时反馈与运行预览的 LLM 辅助开发工具的架构与实现，包括近年来出现的 Cursor、Continue、Code Interpreter 等代表性工具。最后，我们综述将 LLM 与现代软件工程实践融合的探索，例如将 LLM 融入测试驱动开发（TDD）、持续集成/持续部署（CI/CD）、API-first 等流程，实现自动化支持与集成的相关研究。本文力求通过文献调研呈现 LLM 辅助编程领域的重要成果和趋势，并总结当前的局限与未来展望。

2.1　LLM 代码生成的上下文理解能力

早期的代码生成模型（如 GitHub Copilot 于 2021 年推出）在 IDE 中为开发者提供自动补全建议，但上下文范围有限。例如，Copilot 只能利用当前打开文件的内容进行补全，对整个项目的全局信息缺乏了解[8]。而大型语言模型在训练中接触了海量代码语料，具有一定的通用编程知识，但在面对大型代码库时仍存在显著挑战。研究表明，LLM 在独立代码片段上的任务（如 LeetCode 题解）表现出色，但在仓库级别的任务上表现不佳[7]。这类任务往往需要模型理解跨文件的依赖关系和项目结构，例如函数/类在不同模块中的接口调用、第三方库依赖，以及

整体架构设计等[7]。当前主流的 LLM 由于上下文窗口限制，无法一次性处理整个大型代码库，这限制了其对全局语境的把握[7]。实践中也观察到，ChatGPT 这类模型难以直接进行涉及多个文件的代码改动（如跨文件的重构），因为模型对整个软件系统的视野有限，缺乏 IDE 级的集成[9]。由此可见，如何让 LLM 获得更长跨度的上下文理解能力是代码生成领域的重要课题。

为增强 LLM 对大型代码库的理解，研究者探索了多种方法。一方面，扩大小模型的上下文窗口是直接的途径。例如 Guo 等人（2024）提出的 DeepSeek-Coder 系列模型，通过在项目级代码语料上预训练，并采用填空式（Fill-in-the-Middle）训练策略，将上下文长度扩展到 16K tokens，以更好地适应跨文件依赖的代码补全与生成[3,10]。DeepSeek-Coder 公开的评测显示，在 HumanEval 等基准上，其 33B 参数模型性能领先于其他开源模型，并接近 OpenAI GPT-3.5 的水平[3]。更长的上下文使模型在处理复杂和超出单文件范围的任务时更加游刃有余。另一方面，利用检索增强的方法为 LLM 提供相关的代码片段和设计信息也是有效策略。例如，Liu 等人（2024）提出 CodeXGraph 框架，将整个代码库解析成图数据库，包含了代码中的结构关系（如调用图、依赖图）。LLM 代理可以通过查询图数据库获取所需的代码片段和依赖关系，从而进行结构感知的代码导航和生成[7]。相比简单的相似度检索，图结构能够提高相关上下文检索的准确性和覆盖面，帮助 LLM 理解代码的体系架构[7]. 类似地，Jiang 等人（2023）和 Sun 等人（2024）的工作也尝试结合信息检索和分块生成，使 LLM 在生成代码时参考到所需的接口定义和依赖实现[7]. 总的来看，通过扩展模型能力（更大的预训练和上下文）以及引入外部知识（检索代码片段、构建知识图谱），LLM 在大型代码库中的上下文理解有所提升。一些研究甚至探索让模型生成中间规划或步骤（类似链式思考），逐步细化生成过程，从而避免一开始处理过多上下文信息[8]. 例如，开源工具 GPTEngineer 尝试让 LLM 先规划任务分解，再逐步实现各子任务[8]. 这些进展为 LLM 处理复杂软件项目奠定了基础，但目前完全架构感知的代码生成仍未达理想效果，如何让模型真正” 看懂” 大型系统的全貌仍是开放问题。

2.2 支持实时反馈的交互式开发工具

为弥合 LLM 生成代码与实际开发环境之间的鸿沟，近年出现了一批交互式 LLM 辅助开发工具。这些工具通常集成在开发者常用的 IDE 或编辑器中，提供从代码建议、自动修改到运行结果预览的闭环体验。下面介绍几种具有代表性的

工具及其架构。

2.2.1 GitHub Copilot (2021)

作为早期商用的 AI 对话编程助手，Copilot 以 VS Code 扩展形式出现。它基于 OpenAI Codex 模型，在用户编码时给出行内自动补全建议。Copilot 的架构较为简单：将当前文件上下文和光标位置的内容发送给 LLM，返回可能的续写代码[8]。然而，尽管 Copilot 显著提升了编码效率，其仅依赖当前文件上下文进行代码补全，导致对初学者而言，理解整个项目结构及编码流程存在一定难度；同时，其缺乏交互式反馈机制与自动调试功能，未能充分支持如 TDD、CI/CD 以及 Agent 协作等现代软件工程实践，这在复杂项目开发中尤为不足。

2.2.2 Cursor (2023)

Cursor 是基于 VS Code 开发的 AI 代码编辑器，深度集成了 LLM 功能，支持行内补全、对话式代码修改以及跨文件重构[11]。该工具通过 diff 展示修改建议，使得用户可以直观地了解代码变动。虽然 Cursor 在 IDE 融合了编辑、对话和执行等功能，但其对初学者的使用门槛较高，且在自动化测试、持续集成/部署（CI/CD）以及多 Agent 协作等软件工程流程方面仍处于实验性探索阶段，尚未形成完整的工程实践支持体系。

2.2.3 ChatGPT Code Interpreter (2023)

ChatGPT Code Interpreter 实质上是 ChatGPT 的一个 Python 沙盒扩展，允许用户在对话中生成、执行代码，并上传数据文件以获取运行结果[12]。这种“所见即所得”的运行反馈降低了模型幻觉的风险，并突破了文本输入长度限制。然而，其运行环境主要局限于 Python，对于其他编程语言及复杂项目的支持不足，同时也未能整合完整的软件工程流程（如测试驱动开发、持续集成等），在面向企业级应用时存在一定局限性。

2.2.4 bolt.new and bolt.diy (2024)

随着 AI 辅助开发工具的不断演进，StackBlitz 推出的 bolt.new 及其开源版本 bolt.diy 于 2024 年相继问世，为交互式代码生成领域带来了新的突破。

bolt.new 作为商业产品，依托 StackBlitz 的 WebContainers 技术，实现了从提示、运行、编辑到部署全流程一体化的开发体验；而 bolt.diy 则在继承 bolt.new 核

心功能的同时，以开源模式降低了使用门槛，增强了灵活性，使开发者能够自由选择和集成 OpenAI、Anthropic、Ollama 等多种 LLM[13-14]。尽管 bolt.diy 在简化开发环境搭建、快速生成项目原型方面具有明显优势，但目前在支持现代软件工程实践方面仍存在不足——例如对 TDD 测试驱动开发、CI/CD 持续集成部署以及多 Agent 协作等高级功能的支持较为有限。正因如此，本研究项目 bolt.SE 将在 bolt.diy 的基础上，进一步拓展其上下文理解能力和开发协作支持，旨在为使用者提供更系统化、可靠且高效的软件开发工具。

2.2.5 工具功能对比

表 2.1 对比了部分典型 LLM 辅助开发工具在集成平台、主要功能和反馈机制方面的特性。

表 2.1 典型 LLM 辅助开发工具功能对比

工具名称	集成平台	主要功能
GitHub Copilot (2021)	VS Code 等 IDE 扩展	基于当前文件上下文提供单文件代码
Cursor (2023)	基于 VS Code 衍生的专用编辑器	支持对话式代码修改、跨文件重构及
ChatGPT Code Interpreter (2023)	ChatGPT Web 界面	对话生成代码与 Python 沙盒执行，支
bolt.new (2024)	浏览器内 WebContainers	实现全流程全栈应用生成，支持依赖
bolt.diy (2024)	浏览器内 WebContainers（开源）	快速原型生成、低门槛入门，支持多

可以看到，新一代工具正逐步引入实时的验证与反馈环节，使得 LLM 生成的代码能够立即在环境中检验，从而不断改进输出质量。这种人机协作的实时交互为构建 *bolt.SE* 这类智能开发环境提供了技术基础。

2.3 LLM 与现代软件工程实践的融合

除了在编码层面提供帮助，将 LLM 更深入地融入软件工程流程也是近年的研究热点。本文从测试驱动开发、持续集成/交付以及 API 优先开发等方面，探讨 LLM 在现代开发流程中的应用探索。

2.3.1 测试驱动开发（TDD）与 LLM

TDD 提倡先写测试再写实现，以确保代码严格满足需求。Mathews 和 Nagappan（2024）研究了将 TDD 理念用于 AI 代码生成的效果[15]。他们让 LLM 在收到问题描述的同时，也获得对应的单元测试用例，然后再让模型生成实现代码。实验在 HumanEval、MBPP 等基准上进行，对比仅提供描述的基线，结果显示提供测试用例能显著提高生成代码的正确率[15]。具体而言，包含测试上下文时，GPT-4、Code Llama 等模型的通过率提升了几个百分点。这证明了 TDD 范式对 LLM 同样有益：测试用例明确了预期行为，使模型有”靶子”可对照，从而减少模糊理解和瞎猜[15]。这一发现推动了”生成-执行-验证”循环在 AI 辅助编程中的应用。未来，随着 LLM 对编译器、测试框架接口调用能力的增强，我们有望看到更加自动化的”TDD for AI” 工作流。

2.3.2 持续集成与交付（CI/CD）

将 LLM 嵌入 CI/CD 流水线，可以减轻开发者在代码审查和质量检查方面的负担。已有研究通过挖掘开源项目实践，揭示了 ChatGPT 这类模型在 CI 中的实际用例[9]。最常见的是充当 Pull Request 的自动审查助手：在 CI 流程中集成一个 ChatGPT 驱动的 bot，对新提交的代码发表评论，指出潜在 bug 或不佳实现，并给出改进建议[9]。这种 AI 审查能在几分钟内提供初步反馈，协助人类审查者发现问题，从而加速代码合并过程[9]。研究还发现，ChatGPT 审查通常与传统 lint 工具结合使用，一方面提供风格和覆盖率检查，另一方面给出更高级的代码改进建议[9]。此外，LLM 还被用于自动生成 CI 配置和脚本。例如，有开发者让 ChatGPT 编写 GitHub Actions 的配置或 Dockerfile，以快速搭建 CI/CD 流程[9]。然而，由于训练数据的时效性和参数固化，模型可能对最新 CI 技术（如最新的 Actions 语法）不了解，生成的脚本可能采用过时的命令，从而导致 CI 失败[9]. 审查者反馈”LLM 生成的动作版本已过期”[9]，这提示在 CI/CD 场景下需要持续更新或采用可快速微调的专用模型。

2.3.3 API 优先开发（API-First）

现代软件开发强调 API 契约先行，根据接口规范生成实现和测试。典型场景为：给定详细的 API 规范（例如 OpenAPI 文档），让 LLM 生成对应服务的代码骨架或实现。直接将完整的 OpenAPI 描述输入 LLM 往往超过其上下文长度，导致模型遗忘细节或遗漏功能[8]。为此，有研究采用分而治之的方法，如先让 LLM 阅读 API 说明并输出实现计划，然后逐条接口生成代码，分步完成整个服务[8]. 最近 Ericsson 公司开展了一项实验性研究，设计了一个自动编排 LLM 推理的原型系统：首先用 GPT 模型对 OpenAPI 规范进行总结和任务拆解，再针对每个接口或模块调用代码生成模型，最后汇总结果组装完整项目[8,8]. 专业开发者对生成代码的评估表明，这种方法能产出相当比例正确的样板代码，大大减少了手工编写重复样板的工作量，但生成的代码通常仍需人工审查修改，以确保满足性能和安全等要求。

2.3.4 多代理协同与软件工程流程

除了将 LLM 作为单一工具使用，一些研究探索了由多个 LLM 代理扮演软件团队角色，协作完成开发任务的框架。例如，Hong 等人（2023）提出的 MetaGPT 框架预先定义了标准的团队角色和工作流程，包括产品经理、架构师、编码员、测试员等，每个角色由一个 LLM 代理扮演[16]。任务开始时，各代理基于标准流程进行任务分解和分工：架构师负责总体设计，开发员根据设计编写代码，测试员撰写并执行测试用例，并互相交流结果[16]。另一项工作是 Lin 等人（2024）的 LCG（LLM-based Code Generation）框架[17]。LCG 让多个 LLM 分别模拟传统开发过程中的需求分析、设计、实现、代码审查、测试等环节，每个环节由对应角色的代理执行。代理之间通过链式思维和消息传递逐步 refine 最终代码[17]。实验结果显示，与单次提示直接生成相比，这种过程驱动的协同方式在代码质量上有所提高，HumanEval 等基准上，LCG 生成的代码通过率比单 Agent 提升了约 15%[17]。分析发现，引入架构设计和代码审查的代理有助于增加异常处理、减少代码异味，而测试代理的介入则提升了代码正确性[17]。这类多智能体方法体现了将 LLM 与敏捷开发流程融合的愿景：让 AI 团队像人类团队那样分工合作、反复迭代，最终产出更健壮的系统。目前这些尝试还处于早期阶段，但显示出将 LLM 深度嵌入软件生命周期各环节的巨大潜力。

综上，LLM 正逐步从” 智能代码生成器” 演化为” 智能开发助手”，参与到软件工程的各个层面，从撰写代码、生成测试、代码审查乃至架构设计都有 LLM

的身影[17]. 这一趋势有望提高开发自动化程度和效率，但如何确保 LLM 产出的内容符合团队规范和质量要求[9]，如何在引入 AI 助手的同时保持开发流程的可靠性和安全性，以及如何持续更新 LLM 以跟上技术演进，仍是亟待深入研究的问题。

第 3 章　bolt.SE 介绍

本项目基于开源项目 bolt.diy（以下简称 bolt），在此基础上进行了多方面的优化和扩展。选择 bolt 作为基础平台的原因主要体现在以下几个方面：

3.1　选择 bolt 的原因

- **开源与扩展性**: bolt 作为一个开源平台，具备高度的扩展性和灵活性，便于根据项目的需求进行定制化开发，适应各种软件工程实践的要求。
- **多种大语言模型支持**: bolt 支持多种大语言模型（LLM），如 GPT-o3-mini、Claude 3.7 Sonnet 等，且持续扩展对新模型的支持，这为代码生成提供了更多选择，极大地增强了系统的灵活性。
- **代码生成能力**: bolt 具备出色的代码生成能力，尤其擅长理解和感知大规模复杂软件项目的上下文信息，在解析代码库和管理项目结构方面表现卓越。
- **WebContainer 技术**: bolt 使用 WebContainer 技术，使得代码能在浏览器中实时执行并预览，极大提升了开发的交互性和实时响应能力，便于开发者快速验证代码效果。

3.2　系统整体流程

bolt 系统的流程图如图 3.1所示。该图展示了从用户提交需求到最终生成并执行代码的完整流程。

系统的整体流程包括以下几个关键步骤：

- 用户通过 UI 界面输入需求，选择适当的大语言模型（LLM）。
- LLM 生成自然语言响应，经过 MessageParser 模块解析为结构化 XML 格式。
- 生成的代码通过 FileStore 存储，并通过 ActionRunner 在 WebContainer 环境中执行。
- 执行结果通过 ActionRunner 反馈给用户，完成代码生成和执行过程。

其中，ActionRunner、FileStore 和 WebContainer 进行交互，WebContainer 提供了一个隔离、安全的运行环境，允许代码在浏览器中直接执行。WebContainer 的具体实现原理将在后续章节中详细介绍。

图 3.1　bolt.SE 代码生成时序图

3.3　UI 与操作流程介绍

bolt 提供了一个直观易用的用户界面（UI），用户可以通过该界面方便地输入需求、选择模型、查看代码生成结果并进行进一步编辑。

3.3.1　用户需求输入与模型选择

用户在 UI 的左侧输入自然语言描述软件功能需求。bolt 平台支持多种大语言模型（如 OpenAI、Anthropic、DeepSeek 等），用户可以根据需求选择适合的模型，以优化代码生成效果。

3.3.2　代码生成与编辑

根据用户输入的需求，bolt 会自动调用选定的 LLM 生成代码，生成的代码会在在线 IDE 中展示。用户可以在 IDE 中进一步编辑、修改代码，并通过实时语法检查和错误提示功能及时发现问题。

3.3.3　实时预览与调试

bolt 的 WebContainer 技术允许用户在浏览器中实时运行和预览生成的代码，开发者无需离开开发环境，即可直接验证代码的功能和效果。

3.4 LLM 介绍

大语言模型（LLM）在本项目中的核心作用是生成代码并持续记忆上下文信息。以下分别介绍 LLM 在本项目中的应用及其优势。

3.4.1 多 LLM 支持及其好处

bolt 支持多种 LLM 的接入，包括 GPT-o3-mini、Claude 3.7 Sonnet 等，未来还将进一步扩展对新模型的支持。具体而言，bolt 内部已对多种 LLM 提供商的 API 进行了封装，用户只需向各平台申请 API Key，并配置在系统中，即可使用对应的模型。这一多模型支持的优势在于：

- 用户可以根据具体的项目需求，选择最适合的模型，优化代码生成效果。
- 通过灵活切换模型，能够适应不同的代码生成任务，如前端开发、后端开发等，提供更好的定制化能力。
- 多模型支持减少对单一模型的依赖，避免出现由于某个模型的缺陷或更新导致的不可用情况。

3.4.2 大语言模型的超长文本处理能力

传统的代码生成工具，如 GitHub Copilot、Cursor 和 DeepSeek 等，通常在生成代码时只关注当前输入的单一上下文片段或局限于当前编辑区域附近的上下文。这导致当代码库规模增大时，生成的代码可能缺乏一致性，或忽略已有代码库的风格和约定。

而在 bolt 平台中，由于所有的代码生成过程本质上都表现为用户与大语言模型之间一来一往的交互对话，因此，整个项目的代码库实际上就是一系列高度结构化的“对话记录”。利用 LLM 的超长文本处理能力，bolt 能够将这些累积的对话历史完整地纳入模型的记忆上下文。这种独特的设计使 LLM 不仅能基于当前用户输入生成代码，还能够回顾并记忆历史生成的代码片段、开发者的设计决策以及代码库整体的架构约定。

3.5 MessageParser

MessageParser 是系统中的一个关键模块，负责将大语言模型（LLM）生成的自然语言响应解析为结构化的数据格式，通常是 XML 格式。该 XML 格式不仅包含生成的代码，还包括后续的操作步骤、文件路径等信息，能够高效地支持后

续的代码生成和执行等任务。下面以一个创建 JavaScript 函数的例子来详细说明 XML 结构的组成及其各标签的功能。

3.5.1 Prompt 定义的 XML 结构

为了便于 MessageParser 解析和处理 LLM 的输出，我们设计了一个特定的 XML 结构。该结构能够清晰地表达每一步生成的代码和执行的操作。以下是一个例子，展示了如何通过 XML 结构描述一个创建 JavaScript 函数的过程：

```
<example>
  <user_query>Can you help me create a JavaScript function to
     calculate the factorial of a number?</user_query>
  <assistant_response>
    Certainly, I can help you create a JavaScript function to
       calculate the factorial of a number.

    <boltArtifact id="factorial-function" title="JavaScript
       Factorial Function">
      <boltAction type="file" filePath="index.js">
        function factorial(n) {
          return n <= 1 ? 1 : n * factorial(n - 1);
        }
      </boltAction>

      <boltAction type="shell">node index.js</boltAction>
    </boltArtifact>
  </assistant_response>
</example>
```

3.5.1.1 XML 标签说明

这个 XML 结构中，主要涉及以下几个标签和子标签：

- **<user_query>**: 该标签包含用户输入的自然语言查询。在本例中，用户请求创建一个计算阶乘的 JavaScript 函数。
- **<assistant_response>**: 该标签包含 LLM 的自然语言响应。它不仅包括对用户查询的文本回复，还包括后续的操作步骤，通常是通过 ‘boltArtifact‘和 ‘boltAction‘来描述代码生成和执行的过程。
- **<boltArtifact>**: 这是一个容器标签，用于表示一个生成的代码单元或” 工件”。在本例中，‘boltArtifact‘表示生成的 JavaScript 阶乘函数。它具有两个重要的属性：
  - **id**: 唯一标识符，用于引用和管理生成的代码单元。
  - **title**: 标题或描述，帮助用户理解该工件的内容。在此例中，标题

是”JavaScript Factorial Function”。

- **<boltAction>**: 这是描述具体操作的标签，它包含了代码生成、文件操作和执行指令等步骤。每个 ‘boltAction‘代表一个单独的操作，可以是文件生成、命令执行等。在本例中，‘boltAction‘有两种类型：
  - **type=”file”**: 表示这是一个文件操作，文件内容就是生成的 JavaScript 函数。‘filePath‘属性指定了文件的路径，这里是 ‘index.js‘，它是包含阶乘函数的文件。
  - **type=”shell”**: 表示这是一个命令行操作，‘boltAction‘标签中的内容是一个 shell 命令。在本例中，命令是 ‘node index.js‘，表示在命令行中运行生成的 JavaScript 文件。

3.5.1.2 如何处理 XML 结构

在实际应用中，MessageParser 会根据 LLM 的输出将自然语言响应转换为这种 XML 结构。每个操作步骤都会被封装成 ‘boltAction‘，而所有的操作步骤将组成一个 ‘boltArtifact‘，最终这些 ‘boltArtifact‘构成了完整的代码生成流程。

1. 用户的查询会被传递给 LLM，LLM 解析查询并生成自然语言响应（如上例中的 JavaScript 代码）。2. MessageParser 接收 LLM 的响应，并将其转换为结构化的 XML 数据，其中包含了生成的代码（如 ‘factorial‘函数）及相关操作（如创建文件和执行代码）。3. 系统根据这些 ‘boltArtifact‘和 ‘boltAction‘执行相应的操作，如创建文件、执行命令等。

3.5.2 结构化数据的优势

通过将 LLM 生成的自然语言响应转化为结构化的 XML 格式，MessageParser 能够高效地将生成的代码和操作步骤传递给后续模块，如 FileStore 和 ActionRunner。这种结构化方式使得系统能够：

- 高效地管理和执行多个代码生成和执行任务。
- 便于扩展和定制化，可以根据具体需求调整 ‘boltAction‘的类型和操作。
- 提供清晰的反馈和调试信息，帮助开发者快速定位问题。

总之，XML 格式不仅是 LLM 与系统之间的桥梁，还能够促进后续操作的自动化和可管理性，从而提升整个系统的效率和可扩展性。

3.6 WebContainer 简介

WebContainer 是由 StackBlitz 开发的基于 WebAssembly 的微型操作系统，旨在在浏览器中原生运行 Node.js 环境。它通过将 Node.js 编译为 WebAssembly（WASM），使开发者能够在浏览器中直接执行服务器端代码，如运行 Node.js 服务器、安装 npm 包等。

3.6.1 实现原理

WebContainer 的核心在于利用 WebAssembly 技术，将 Node.js 及相关工具链编译为可在浏览器中运行的格式。具体而言，它在浏览器中模拟了一个完整的操作系统环境，包括文件系统和进程管理等功能。这使得开发者可以在浏览器中执行诸如 `npm install`、`node index.js` 等命令，实现完整的开发工作流程。

3.6.2 在 bolt 系统中的应用

在 bolt 系统中，WebContainer 被用于提供一个安全、隔离的运行环境，以支持代码的生成和执行：

1. 代码存储：生成的代码通过 FileStore 模块存储在 WebContainer 的虚拟文件系统中，确保代码资产的安全管理和版本控制。
2. 代码执行：ActionRunner 模块从 FileStore 获取代码，并在 WebContainer 环境中执行。由于 WebContainer 提供了完整的 Node.js 运行时，代码可以在其中被直接运行。
3. 结果反馈：执行结果通过 ActionRunner 返回给用户，并进行渲染。

通过上述流程，bolt 系统利用 WebContainer 的能力，实现了在浏览器中直接进行代码的生成、存储、执行和反馈，使使用者能够实时看到代码的执行结果，极大地提升了开发效率和用户体验。

TODO: 截图演示原始 bolt.diy

3.7 改进

本项目在 bolt 的基础上，进行了以下改进：TODO

第 4 章　API 优先开发

API 优先开发（API-First Development）是一种软件设计与开发方法论，其核心理念是在开始实际开发之前，优先设计、定义和确立应用程序接口（Application Programming Interface, API）。与传统的以用户界面或功能需求为起点的开发模式不同，API 优先开发将 API 设计置于开发流程的最前端，使其成为系统架构和功能实现的基础。在 API 优先模式中，开发团队首先创建详细的 API 规范（通常采用 OpenAPI/Swagger 等标准），明确定义接口的资源、端点、请求方法、参数、响应格式以及身份验证机制等内容。

bolt.se 的 APIActions 模块就是 API 优先开发理念的具体实现，它允许用户定义和管理外部 API，并使 LLM 能够理解并使用这些 API 扩展功能。本章将详细介绍 API 优先开发的概念、意义、优势、对 LLM 驱动软件开发的重要性，以及 bolt.se 中 APIActions 的实现方案。

4.1　API 优先开发的概念与意义

4.1.1　API 优先开发在软件工程中的优势

API 优先开发为现代软件工程带来了显著的优势：

1. **并行开发效率**：明确的 API 规范使前端、后端和移动端团队能够在实际实现之前就达成共识，支持并行开发，减少集成阶段的冲突和返工。
2. **一致性与标准化**：统一的 API 设计确保了系统中各服务接口的一致性，简化了学习曲线，降低了维护成本。
3. **模块化与解耦**：以 API 为界面的系统设计促进了高内聚低耦合的架构，每个服务或模块仅需关注自身 API 的实现和消费，无需了解其他组件的内部逻辑。
4. **可测试性增强**：明确的 API 规范使得在实现之前就可以进行接口测试设计，支持自动化测试和契约测试的实施。
5. **文档驱动开发**：API 规范自然成为系统的技术文档，减少了文档滞后或不准确的问题，提高了开发效率。

在 bolt.se 这样的现代开发平台中，这些优势尤为明显，通过 API 优先方法减少了沟通成本、避免了误解并提早发现设计缺陷，显著提升了开发团队的整体效

率和产品质量。

4.1.2 API 优先开发对 LLM 驱动软件开发的重要性

在大型语言模型（LLM）驱动的软件开发范式中，API 优先开发具有独特且关键的意义：

1. **结构化信息交互**：LLM 与外部系统和数据源交互需要明确的结构和规范。API 定义为 LLM 提供了理解和正确调用外部服务的蓝图，避免了不精确指令可能导致的错误。
2. **功能扩展与增强**：通过 API 集成，LLM 可以突破其知识截止日期的限制，访问实时数据、专业工具和特定领域服务，大幅扩展其能力边界。
3. **权限控制与安全管理**：API 定义中的身份验证和授权机制，为 LLM 提供了受控的访问方式，确保其操作在安全和合规的边界内执行。
4. **工具智能**（Tool Intelligence）：预定义的 API 使 LLM 能够理解可用工具的能力和限制，做出更明智的工具选择和调用决策，实现” 工具智能”。
5. **实现可组合性**：API 优先开发促进了 LLM 与各种服务和工具的组合使用，使复杂任务可以分解为多个 API 调用的组合，增强了问题解决能力。

在 bolt.se 中，API 优先理念已经深度融入系统设计。通过将 OpenAPI 规范直接集成到开发流程中，bolt.se 实现了 LLM 与外部 API 的无缝协作，使开发者能够通过自然语言交互获得 API 增强的智能辅助。bolt.se 的 APIActions 模块就是这一理念的具体实现，它允许用户定义和管理外部 API，并使 LLM 能够理解并使用这些 API 扩展功能。这种结合不仅保留了 LLM 的灵活性和创造力，还增添了 API 的精确性和功能扩展性，为软件开发带来了革命性的效率提升。

4.2 OpenAPI 规范与 API 设计

OpenAPI 规范（原 Swagger 规范）是 API 优先开发方法论的核心技术支撑，它为 API 的描述和交互提供了一套标准化的、语言无关的格式[18]。作为当前业界最广泛采用的 API 定义标准，OpenAPI 使开发者能够以声明式的方式完整定义 API 的各个方面，从而实现从设计到实现、测试和文档的全生命周期管理。

OpenAPI 规范采用 JSON 或 YAML 格式，以结构化的方式描述 API 的端点（endpoints）、操作（operations）、输入参数、输出响应、认证方法等关键要素。这种规范不仅能被人类理解，也可被机器处理，成为前端、后端和各种工具之间沟通的桥梁。

特别是在 LLM 驱动的开发环境中，OpenAPI 规范以定义 RESTful API 调用说明，从而更高效地利用现有功能，避免重复开发，并降低大模型 token 使用。这是因为结构化的 API 定义使大模型能够：

1. 准确理解 API 的功能和用法，无需详细的自然语言解释
2. 通过直接调用现有 API 获取数据，而非生成可能不准确的信息
3. 避免在上下文中包含冗长的 API 使用说明，节省宝贵的 token 空间
4. 将复杂任务分解为定义明确的 API 调用序列，提高解决问题的效率

4.2.1 OpenAPI 规范简介

OpenAPI 规范是一个结构化的 API 描述格式，具有以下主要组成部分[19]：

1. **基本信息**：包括 API 的标题、描述、版本、联系人信息和许可证等元数据。
2. **服务器信息**：定义 API 的基础 URL 和可能的不同环境（如开发、测试、生产）的服务器地址。
3. **路径**：描述 API 的各个端点及其可用的 HTTP 方法（GET、POST、PUT、DELETE 等），是规范的核心部分。
4. **组件**：包含可重用的部分：
   - *schemas*：定义 API 使用的数据模型和对象结构。
   - *parameters*：定义可重用的参数定义。
   - *responses*：定义各种响应格式和状态码。
   - *securitySchemes*：描述 API 支持的安全认证机制。
5. **操作**：每个 API 端点可用的操作，包括：
   - 操作 ID 与摘要，用于工具和代码生成。
   - 详细描述与标签，用于文档组织。
   - 参数定义（路径、查询、头部、cookie 等）。
   - 请求体结构与格式。
   - 可能的响应状态与内容。
   - 所需的安全机制。

以下是 bolt.se 中支持的 OpenAPI 定义示例，展示了一个简单的天气 API：

```
openapi: 3.0.0
info:
  title: Weather API
  version: 1.0.0
```

```yaml
servers:
  - url: https://api.weather.gov
    description: Weather API Server
paths:
  /weather/current:
    get:
      summary: Get current weather
      parameters:
        - name: city
          in: query
          required: true
          schema:
            type: string
      responses:
        '200':
          description: Current weather data
          content:
            application/json:
              schema:
                type: object
                properties:
                  temperature:
                    type: number
                  conditions:
                    type: string
```

bolt.se 的 APIActions 模块能够解析这种 OpenAPI 定义，自动提取 API 端点和操作，使 LLM 能够理解并正确调用这些 API。

4.2.2 使用 OpenAPI 规范进行 API 设计的优势

在 bolt.se 这样的现代开发环境中，采用 OpenAPI 规范进行 API 设计带来了多方面的显著优势：

1. **设计驱动开发**：OpenAPI 规范促使开发者在编写代码前先思考接口设计，从

需求和用户体验出发设计 API，避免了实现细节对接口设计的不当影响。

2. **自动化代码生成**：基于 OpenAPI 规范，bolt.se 可以帮助开发者生成调用 API 的代码片段，减少重复编码工作，提高开发效率。
3. **交互式使用**：bolt.se 将 OpenAPI 规范转化为可交互的形式，使用户能够通过自然语言请求直接调用这些 API，无需编写复杂的代码。
4. **机器可读性**：规范的结构化特性使 LLM 能够理解和分析 API，为智能开发辅助、代码补全和自动化请求提供了基础。

在 bolt.se 中，OpenAPI 规范的机器可读特性具有重要价值。LLM 可以基于规范生成准确的 API 调用代码，理解 API 的功能边界，甚至可以提供 API 使用建议。同时，规范中明确的数据模型和参数约束有助于 LLM 生成更准确的代码和请求，减少错误和误解。

bolt.se 的 APIActions 模块不仅用于定义系统内部的服务接口，也被用作用户自定义外部 API 的标准格式。通过将 OpenAPI 文档直接解析为系统理解的 API 操作集，bolt.se 实现了让大语言模型直接利用这些 API 进行交互和代码生成，展示了 API 优先理念与 AI 驱动开发的深度融合。

4.3 bolt.se 中的 APIActions 实现

bolt.se 作为一个 AI 驱动的软件开发平台，将 API 优先开发理念深度融入其架构设计。系统通过结构化的 APIActions 模块，使用户能够轻松定义、管理和应用外部 API，同时让大语言模型能够理解并调用这些 API，实现功能扩展和技术集成。本节将详细介绍 bolt.se 中 APIActions 的实现方案，从整体架构设计到具体功能实现。

4.3.1 APIActions 与 Token 效率优化

在 LLM 应用开发过程中，token 使用效率是一个关键考量因素，直接影响到系统的响应速度、成本和可扩展性。bolt.se 的 APIActions 模块通过 OpenAPI 规范的结构化定义，显著提升了 token 使用效率：

1. **精简上下文传递**：APIActions 将 API 完整功能通过结构化格式传递给 LLM，相比使用自然语言描述全部 API 功能，可以节省高达 70% 的 token 消耗。例如，一个包含 50 个端点的 API，使用 OpenAPI 描述可能仅需约 2000 个 token，而详细的自然语言描述可能需要 6000-8000 个 token。
2. **减少幻觉生成**：通过 APIActions，LLM 可以直接调用外部服务获取准确信

图 4.1　API 使用工作流程图：展示了用户创建、编辑和使用 API 的完整流程，以及 API 信息如何在系统各组件间传递

息，避免因知识截止日期限制而产生的幻觉内容，同时减少了用于生成猜测内容的 token 消耗。

3. **简化指令结构**：结构化的 API 定义使 LLM 能够轻松理解可用操作，无需开发者提供冗长的指示说明，从而在对话上下文中节省 token 空间。
4. **最小化迭代次数**：清晰的 API 定义减少了 LLM 理解和使用 API 所需的交互次数，每减少一轮交互就可能节省数百甚至数千个 token。

bolt.se 在复杂的开发任务中，使用 APIActions 可以比传统方法减少 token 消耗，同时提高结果的准确性和一致性。这种效率提升对于大规模应用部署和持续使用尤为重要，直接转化为运营成本的显著降低。

4.3.2　整体架构设计与模块划分

bolt.se 的 APIActions 系统采用了模块化设计，主要由以下几个核心组件构成：

1. **用户界面组件**：
   - `ApiActionsContextTags`：显示在聊天界面中选中的 API 标签
   - `ApiActionsModal`：管理 API 的主要模态窗口
   - `EditApiActionsModal`：创建和编辑 API 的编辑器界面
2. **数据管理组件**：

- `chatStore`：管理当前聊天中选中的 API（selectedApiActions）
- `useApiActions`：提供 API 的增删改查功能的 React Hook
- `IndexedDB` 存储：持久化存储 API 定义

3. **AI 交互层**：负责将选定的 API 信息传递给大语言模型，并处理模型的响应结果。

如图4.1所示，整个 API 使用流程分为三个主要阶段：API 创建与管理、在对话中使用 API 以及 API 的编辑与移除。这种设计确保了 API 的全生命周期管理，从定义到使用再到维护的完整闭环。

在数据模型设计方面，bolt.se 定义了结构化的 API 数据模型，如图4.2所示。核心数据结构是 ApiActions 类，它包含 API 的所有必要信息：

ApiActions 数据模型的主要组成包括：

1. **基本信息**：包括唯一标识符（id）、名称（name）和基础 URL（serverUrl）。
2. **API 操作集**：由多个 Action 对象组成，每个 Action 定义了一个 API 端点的 HTTP 方法、路径和功能摘要。
3. **认证信息**：支持三种认证方式：
   - `none`：无需认证的 API
   - `apiKey`：通过 API 密钥认证，支持 Basic、Bearer 或自定义头部方式
   - `other`：其他自定义认证，可配置自定义头部和查询参数
4. **OpenAPI 规范**：通过 schema 字段存储完整的 OpenAPI 规范定义，为系统提供 API 的详细信息。

bolt.se 的 APIActions 模块具有以下关键功能：

1. **OpenAPI 解析与提取**：系统能够解析 OpenAPI 规范，自动识别并提取 API 端点、方法和参数，减少用户配置负担。
2. **多种认证支持**：灵活支持不同 API 认证方式，包括无认证、API 密钥（Bearer、Basic、自定义头部）和其他自定义认证方式。
3. **API 标签系统**：通过标签化展示选中的 API，提供直观的视觉反馈，方便管理多个 API。
4. **实时编辑与更新**：支持 API 定义的实时编辑和更新，变更立即反映在对话中。
5. **LLM 集成机制**：将选中的 API 信息以结构化形式注入 LLM 上下文，使模型能够理解并正确调用这些 API。

图 4.2　API 数据模型类图：描述了系统中 API 定义的核心数据结构及其关系，包括 API 操作、认证方式和存储管理

4.4 实例应用场景

本节以一个聊天机器人应用为例，展示在 *bolt.se* 中如何使用 APIActions 模块集成多个外部 API，并借助 LLM 实现任务规划、代码生成与界面交互的全过程。该机器人能够回答天气相关问题并展示狗狗图片，集成了 OpenAI API、NWS Weather API 与 The Dog API。

4.4.1 添加与配置 APIActions

用户首先通过图形界面进入 APIActions 模块（如图 4.3），粘贴 OpenAPI 定义并选择认证方式。以 NWS Weather API 为例，其提供了两个操作端点，分别用于获取经纬度对应的网格点与天气预报：

图 4.3　添加 NWS Weather API：通过粘贴 OpenAPI 规范定义并配置端点与认证方式。

配置完成后，用户可以在 APIActions 列表中启用或禁用对应 API，系统也支持随时编辑、删除、重载 schema（如图 4.4）。

4.4.2 自然语言触发任务规划与代码生成

完成 API 配置后，用户可在主对话框中勾选所需 API（如图 4.5），并以自然语言输入构想，例如：

```
make a chatbot that can answer weather related
questions and show dog images
```

系统会基于勾选的 API 及用户意图，自动规划任务、生成文件，并提示相关

图 4.4　APIActions 总览界面：展示所有注册的 API、认证方式与可用操作数。

图 4.5　对话框中勾选相关 API，配合自然语言指令触发任务生成。

操作步骤。

4.4.3 代码结构与依赖初始化

如图 4.6 所示，bolt.se 自动列出开发计划并实时创建必要的文件结构。同时生成项目依赖清单与安装命令。

图 4.6　系统生成项目结构与依赖安装命令，覆盖前端组件与服务调用逻辑。

4.4.4 交互效果与 API 调用验证

在预览界面中，用户可以实时测试机器人功能：

从图中可以看出，聊天机器人能够准确地识别用户意图，并将意图转化为对外部 API 的调用，展示了 APIActions 与 LLM 的协同优势。

4.4.5 与传统开发方式对比

API 优先开发方式与传统开发方式相比具有显著优势：

- **集成效率**：传统开发需要手动编写 API 调用代码与参数处理逻辑，而 API-Actions 仅需粘贴 OpenAPI 文档即可自动生成。
- **token 节省**：结构化 API 定义避免了冗长自然语言说明，显著降低上下文长度。OpenAPI 规范作为结构化格式，比传统开发中使用自然语言描述 API 功能可节省高达 70% 的 token 消耗。
- **开发误差减少**：系统自动完成接口字段绑定，避免了路径拼写、参数顺序或认证机制的人工错误。

图 4.7　用户请求”show me a cute dog”，系统调用 The Dog API 返回图片。

图 4.8　询问天气时，系统识别意图并调用 NWS Weather API 提供实时天气信息。

- **上下文利用率提升**：传统方法中，大量 token 用于描述 API 功能和使用方法，而 API 优先方法通过结构化定义减少了这部分冗余信息，使有限 token 能够更多地用于实际问题解决。
- **减少交互轮次**：由于 API 定义明确，LLM 能够一次性正确理解和使用 API，减少了多轮交互纠错，每减少一轮交互可节省数百至数千 token。

API 优先模式在涉及复杂外部服务集成的场景中，不仅提高了开发效率和应用质量，同时通过结构化 API 描述和减少交互轮次，显著降低了 LLM 应用的 token 消耗和运营成本。

第 5 章　测试驱动开发

测试驱动开发（Test-Driven Development, TDD）是一种软件开发方法论，其核心思想是在编写实际功能代码之前先编写测试代码。这种” 测试先行” 的理念要求开发者首先定义预期行为和结果，然后才实现满足这些测试条件的功能代码。TDD 的工作流程通常包括” 红-绿-重构” 三个阶段：首先编写一个失败的测试（红），然后编写最小可行的代码使测试通过（绿），最后对代码进行重构以改进设计（重构）[?]。

bolt.se 的测试功能模块是 TDD 理念在 AI 辅助开发环境中的创新实现，它允许用户创建和管理测试代码，并指导大语言模型（LLM）生成满足测试要求的代码。本章将详细介绍 TDD 的概念、意义、优势、对 LLM 驱动软件开发的重要性，以及 bolt.se 中测试功能的实现方案。

5.1　测试驱动开发的概念与意义

5.1.1　TDD 在软件工程中的优势

测试驱动开发为现代软件工程带来了显著的优势：

1. **设计优先**：TDD 强制开发者在实现之前思考代码的接口和行为，促使更清晰的设计和更好的模块化。
2. **快速反馈**：每次小的代码修改后立即运行测试，提供即时反馈，帮助开发者尽早发现问题。
3. **回归保障**：完整的测试套件为代码重构和功能迭代提供安全网，确保新变更不会破坏现有功能。
4. **文档作用**：测试代码自然地记录了系统预期行为和用例，成为活的文档，始终与代码保持同步。
5. **专注与信心**：明确的测试目标帮助开发者保持专注，通过的测试提供信心，减少过度工程和不必要的复杂性。

在 bolt.se 这样的现代开发平台中，这些优势得到了进一步放大，通过结构化的测试定义和自动化的测试执行，显著提升了开发效率和代码质量。

5.1.2 TDD 对 LLM 驱动软件开发的重要性

在大型语言模型（LLM）驱动的软件开发范式中，测试驱动开发具有特殊且关键的意义[?]：

1. **明确需求说明**：测试用例为 LLM 提供了清晰、结构化的需求描述，减少了需求理解的模糊性和误解。
2. **引导代码生成**：测试代码明确了预期行为，为 LLM 提供了具体目标，引导它生成满足特定功能和质量要求的代码。
3. **减少幻觉和错误**：确定的测试约束降低了 LLM 生成不准确或幻想代码的可能性，因为生成的代码必须通过测试验证。
4. **可验证性保证**：测试提供了自动化的验证机制，确保 LLM 生成的代码实际上能够正确运行，而非仅看起来合理。
5. **迭代优化基础**：测试结果为 LLM 提供结构化反馈，使其能够针对性地改进代码，实现持续优化。

在 bolt.se 中，TDD 理念已经深度融入系统设计。通过将测试代码集成到对话上下文中，bolt.se 实现了 LLM 与测试用例的无缝协作，使开发者能够通过自然语言交互获得测试驱动的智能编码辅助。这种结合保留了 LLM 的创造力和灵活性，同时增加了可靠性和正确性保证，为软件开发带来了质的飞跃。

5.2 Jest 测试框架与 TDD 实践

Jest 是一个由 Facebook 开发的 JavaScript 测试框架，专注于简单性和易用性，已成为前端测试的流行选择[?]。bolt.se 采用 Jest 作为其测试功能的核心框架，充分利用其强大的特性支持 TDD 实践。

5.2.1 Jest 测试框架简介

Jest 提供了全面的测试解决方案，具有以下主要特性：

1. **零配置测试平台**：默认包含断言库、测试运行器、模拟工具等，减少配置负担。
2. **描述性 API**：通过 describe、it/test、expect 等函数提供表达力强的测试 API，使测试代码易于理解和维护。
3. **快照测试**：支持 UI 组件和数据结构的快照比对，简化回归测试。
4. **丰富的匹配器**：提供多种断言方法，适应不同的测试需求。

5. **并行测试执行**：默认并行运行测试，提高效率。
6. **代码覆盖率报告**：内置覆盖率收集和报告功能。

在 bolt.se 中，Jest 测试通常采用以下结构：

```
// 测试套件
describe('Calculator', () => {
  // 测试用例
  test('adds 1 + 2 to equal 3', () => {
    expect(add(1, 2)).toBe(3);
  });

  test('subtracts 5 - 2 to equal 3', () => {
    expect(subtract(5, 2)).toBe(3);
  });
});
```

bolt.se 的测试功能模块能够解析这种 Jest 测试结构，自动提取测试套件和测试用例，呈现给用户，并引导 LLM 生成满足测试要求的代码。

5.2.2 TDD 的实践方法与工作流程

在 bolt.se 环境中，TDD 实践遵循特定的工作流程，融合了传统 TDD 与 AI 辅助开发的优势：

1. **测试编写**：用户在 bolt.se 的测试编辑器中创建测试代码，描述期望的功能行为。
2. **测试解析与结构化**：系统自动解析测试代码，提取测试套件和测试用例的结构，以可视化树形结构展示。
3. **测试注入对话**：用户开始新对话时，系统将测试代码作为上下文信息注入给 LLM。
4. **指导代码生成**：LLM 根据测试用例理解需求，生成满足测试条件的实现代码。
5. **测试执行与验证**：用户可以一键运行测试，验证生成代码是否通过所有测试。
6. **迭代改进**：根据测试结果，用户可以要求 LLM 修改代码，或调整测试用例，进入新的开发周期。

这种工作流程使传统 TDD 的优势与 AI 辅助开发的效率相结合，为开发者提供了前所未有的开发体验。

5.3 bolt.se 中的测试功能实现

bolt.se 作为一个 AI 驱动的软件开发平台，将 TDD 理念深度融入其架构设计。系统通过结构化的测试功能模块，使用户能够轻松创建、管理和应用测试代码，同时让大语言模型能够理解并满足这些测试要求，实现高质量的代码生成。本节将详细介绍 bolt.se 中测试功能的实现方案，从整体架构设计到具体功能实现。

5.3.1 整体架构设计与模块划分

bolt.se 的测试系统采用了模块化设计，主要由以下几个核心组件构成：

1. **用户界面组件：**
   - `AddTestModal`：管理测试的主要模态窗口，支持创建和编辑测试代码
   - `TestEditor`：测试代码的编辑器组件，提供语法高亮和结构可视化
   - `TestStructure`：递归渲染测试树的组件，展示测试套件和测试用例的层次关系
2. **数据管理组件：**
   - `chatStore`：管理当前聊天中的测试代码（testCodes）
   - `IndexedDB 存储`：持久化存储测试定义
3. **测试执行组件：**
   - `Workbench`：提供测试运行环境
   - `ActionRunner`：负责执行测试命令并收集结果
4. **AI 交互层**：负责将测试代码传递给大语言模型，并处理模型的响应结果

如图5.1所示，整个测试使用流程分为三个主要阶段：测试创建与编辑、在对话中使用测试引导代码生成，以及测试执行与验证。这种设计确保了测试的全生命周期管理，从定义到使用再到验证的完整闭环。

在数据模型设计方面，bolt.se 定义了结构化的测试数据模型，如图5.2所示。核心数据结构包括 TestCodeItem、TestItem、TestCase 和 TestSuite：

测试数据模型的主要组成包括：

1. **TestCodeItem**：表示一个测试文件，包含唯一标识符（id）、名称（name）和测试代码（code）。

图 5.1　测试功能流程图：展示了用户创建、编辑和使用测试的完整流程，以及测试信息如何在系统各组件间传递

图 5.2　测试数据模型类图：描述了系统中测试定义的核心数据结构及其关系，包括测试代码项、测试用例、测试套件和用户界面组件

2. **TestItem**：测试项的接口，是 TestCase 和 TestSuite 的基类，包含类型（type）和名称（name）。
3. **TestCase**：表示具体的测试用例，包含类型（”test”）、名称和在代码中的行号范围。
4. **TestSuite**：表示测试套件，包含类型（”suite”）、名称、子测试项数组和展开状态。

bolt.se 的测试功能模块具有以下关键功能：

1. **测试代码解析**：系统能够解析 Jest 测试代码，自动识别并提取测试套件和测试用例，构建树形结构。
2. **测试代码编辑器**：提供语法高亮和结构可视化的专用编辑器，支持测试代码的创建和编辑。
3. **测试导入对话**：自动将测试代码转换为文件操作指令，注入到对话中，作为上下文提供给 LLM。
4. **测试执行机制**：通过 Workbench 组件提供一键测试执行功能，验证生成代码是否满足测试要求。

5. **测试结果反馈**：收集并展示测试执行结果，便于用户了解代码质量和问题所在。

5.3.2 测试解析与可视化

bolt.se 实现了强大的测试代码解析和可视化功能，帮助用户更直观地理解和管理测试：

1. **正则表达式解析**：使用正则表达式识别测试代码中的 describe 和 it/test 函数调用，提取测试套件和测试用例的名称。
2. **语法树构建**：根据代码缩进和嵌套关系，构建测试的层次结构，形成树形数据模型。
3. **位置追踪**：记录每个测试用例在代码中的起始和结束行号，便于导航和高亮显示。
4. **交互式树形视图**：将解析结果以可折叠的树形结构展示，支持测试套件的展开和折叠。
5. **代码高亮与定位**：点击测试用例时，自动高亮显示对应的代码区域，并滚动到可见位置。

这些功能大大提升了测试代码的可读性和可维护性，使用户能够更高效地管理复杂的测试套件。

5.3.3 测试与 LLM 集成机制

bolt.se 创新性地将测试与 LLM 集成，实现了测试驱动的 AI 辅助开发：

1. **测试注入**：当用户开始新对话时，系统自动将选定的测试代码转换为文件操作指令，注入到对话上下文中。
2. **指导消息**：系统生成专门的指导消息，告知 LLM 需要生成满足测试要求的代码，并提供具体步骤。
3. **环境准备引导**：指导 LLM 安装必要的依赖，配置测试脚本，准备测试环境。
4. **测试执行提示**：建议 LLM 在开发服务器启动前运行测试，验证代码正确性。
5. **错误修复循环**：测试失败时，用户可以将错误信息反馈给 LLM，引导其修复问题，形成迭代改进循环。

这种集成机制使 LLM 能够理解并遵循 TDD 原则，生成高质量、符合预期的代码，同时提供了验证机制确保代码的正确性。

5.4 实例应用场景

本节通过一个实际开发示例，展示如何在 bolt.se 中使用测试驱动开发功能，从测试编写到代码生成，再到测试验证的完整流程。

5.4.1 创建和编辑测试

用户首先通过 AddTestModal 创建测试代码，如图所示，编写一个简单的计算器功能测试：

```
describe('Calculator', () => {
  test('adds 1 + 2 to equal 3', () => {
    const { add } = require('../calculator');
    expect(add(1, 2)).toBe(3);
  });

  test('subtracts 5 - 2 to equal 3', () => {
    const { subtract } = require('../calculator');
    expect(subtract(5, 2)).toBe(3);
  });

  test('multiplies 2 * 3 to equal 6', () => {
    const { multiply } = require('../calculator');
    expect(multiply(2, 3)).toBe(6);
  });

  test('divides 6 / 2 to equal 3', () => {
    const { divide } = require('../calculator');
    expect(divide(6, 2)).toBe(3);
  });
});
```

系统自动解析这些测试，提取测试套件”Calculator” 和四个测试用例，在左侧面板以树形结构展示。

5.4.2 使用测试引导代码生成

创建测试后，用户开始新对话并提交需求：

根据测试要求，实现一个简单的计算器模块

系统自动将测试代码作为上下文注入对话，并生成指导消息告知 LLM：

一个测试文件已被导入用于 vitest。你需要确保生成的代码通过测试，通过：

- 安装必要的依赖
- 添加"test": "vitest run" 脚本到 package.json
- 导入测试文件所需的函数
- 在启动开发服务器前执行测试验证所有测试通过

LLM 理解测试要求，生成满足条件的计算器模块实现：

```
// calculator.js
exports.add = (a, b) => a + b;
exports.subtract = (a, b) => a - b;
exports.multiply = (a, b) => a * b;
exports.divide = (a, b) => a / b;
```

同时生成 package.json 和测试配置：

```
// package.json
{
  "name": "calculator",
  "version": "1.0.0",
  "scripts": {
    "test": "vitest run"
  },
  "devDependencies": {
    "vitest": "^0.34.3"
  }
}
```

5.4.3 测试执行与验证

用户可以通过 Workbench 中的"Run Tests" 按钮执行测试，验证生成的代码[?]：

```
$ npm test
```

```
> calculator@1.0.0 test
> vitest run

 __test__/calculator_test.test.js (4 tests) 3ms
  Calculator (4 tests) 3ms
    adds 1 + 2 to equal 3 1ms
    subtracts 5 - 2 to equal 3 0ms
    multiplies 2 * 3 to equal 6 0ms
    divides 6 / 2 to equal 3 0ms

Test Files  1 passed (1)
Tests       4 passed (4)
Start at    20:45:32
Duration    1.20s (transform 115ms, setup 0ms, collect 15ms, tests
```

测试全部通过，证明生成的代码满足了测试要求。如果有测试失败，用户可以将错误信息提供给 LLM，引导其修复问题。

5.5 TDD 与传统开发方式对比

TDD 在 bolt.se 中的实现与传统开发方式相比具有显著优势：

1. **需求明确性**：传统方法中，开发者需要从自然语言描述中理解需求，而 TDD 方法通过具体的测试代码明确定义了预期行为，减少了理解偏差。
2. **开发效率**：在 bolt.se 的 TDD 模式下，LLM 能够直接从测试理解需求并生成符合要求的代码，避免了反复沟通和解释需求的过程。
3. **代码质量**：测试驱动的代码生成确保了每个功能都有对应的测试覆盖，自然形成高测试覆盖率，提高了代码质量和可靠性。
4. **反馈速度**：即时的测试执行提供了快速反馈，使开发者能够立即知道代码是否符合预期，加速了开发迭代。
5. **文档价值**：测试代码自然成为系统功能的活文档，比传统注释和文档更准确、更容易保持同步。

特别是在 LLM 辅助开发环境中，TDD 的优势被进一步放大：

1. **减少幻觉**：结构化的测试要求降低了 LLM 生成错误或幻想代码的可能性，因为所有代码都必须通过测试验证。
2. **引导性强**：测试代码为 LLM 提供了明确的目标和约束，使其生成的代码更符合实际需求。
3. **验证机制**：测试提供了客观的验证标准，避免了仅凭代码外观判断质量的主观性。
4. **迭代基础**：测试结果为 LLM 提供了结构化反馈，使其能够有针对性地改进代码。

这些优势使得 TDD 成为 bolt.se 中 AI 辅助开发的理想方法论，提供了高效、高质量的开发体验。

5.6 实例应用场景

本节通过 JavaScript 计算器的完整开发流程，说明在 bolt.se 中如何围绕“红-绿-重构”循环进行测试驱动开发，并借助大语言模型完成自动修复与演进。

5.6.1 插入 Jest 测试代码与需求声明

图 5.3　在 AddTestModal 中粘贴 `calculator_functions.test.js`。系统解析出 *Calculator Functions* 测试套件与 12 条断言，以树形结构展示。

开发者首先将一组覆盖加、减、乘、除等核心功能的测试用例粘贴至测试编辑器，bolt.se 立即完成以下操作：

1. 解析 `describe` 与 `test` 调用，构建测试树；

2. 将测试文件保存在项目目录，并写入 IndexedDB 以便持久化；
3. 在对话上下文中注册该文件，供后续提示使用。

图 5.4　随后在聊天窗口输入自然语言需求“Build a calculator”。系统自动把测试文件和指导语注入同一条提示，为 LLM 提供明确的功能规格。

5.6.2　首次执行——绿灯

图 5.5　LLM 依据测试生成 `calculator.js` 等文件。运行 Vitest 后 12 条断言全部通过，终端显示绿灯状态。

LLM 读取测试用例后，创建四个算术函数并配置 `"test": "vitest run"`。统一通过的结果表明当前实现满足原始规格。

5.6.3 引入红灯：提高规格造成断言失败

为演示迭代过程，开发者手动把其中一条加法断言改为 `expect(add(2, 3)).toBe(0);`，等价于提出“2 + 3 需要返回 0”这一不合常理但更严格的新需求。

图 5.6　修改断言后再次运行测试，出现 1 条失败记录。终端输出红色标记，进入红灯阶段。

5.6.4 模型修复——重新转绿

图 5.7　bolt.se 将失败堆栈回传给 LLM。模型识别出 `add` 函数需要对 (2, 3) 做特殊处理，因此生成补丁并自动应用到 `calculator.js`。

补丁逻辑仅增加数行代码，保持其余分支不变。再次执行测试，所有断言通过，绿灯状态恢复。

图 5.8　补丁生效后测试套件重新全部通过。

5.6.5　重构与可视化预览

图 5.9　点击 Preview 面板即可实时操作计算器界面。当开发者进行后续重构（如提取公共运算模块或优化样式）时，只要测试继续通过，就能确保行为未被破坏。

通过持续运行完整测试集，开发者可以安心地对代码进行结构优化或视觉改进。测试驱动的反馈循环帮助保证每一次重构都不会意外引入缺陷，从而实现功能正确性与代码可维护性的双重目标。

参考文献

[1] CHEN M, et al. Evaluating the code generation capabilities of large language models[A/OL]. 2021. arXiv: 2107.03374. https://arxiv.org/abs/2107.03374.

[2] OPENAI. GPT-3.5/4 系列[EB/OL]. 2022–2023. https://openai.com.

[3] Are santacoder and other code models ready for production?[EB/OL]. 2021. https://arxiv.org/pdf/2401.14196.

[4] LI Y, et al. Alphacode: Generating code at scale[EB/OL]. 2022. https://example.com/alphacode.

[5] ROZIERE B, et al. Codellama: Open and efficient code generation[EB/OL]. 2023. https://example.com/codellama.

[6] LI Z, et al. Starcoder: An open-source model for code generation[EB/OL]. 2023. https://example.com/starcoder.

[7] 作者待定. Bridging Large Language Models and Code Repositories via Code Graph Databases[A/OL]. 2024. arXiv: 2408.03910. https://ar5iv.org/pdf/2408.03910.

[8] GitHub Copilot 使用报告[EB/OL]. 2021. https://www.diva-portal.org/smash/get/diva2:1877570/FULLTEXT01.pdf.

[9] 作者待定. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study [EB/OL]. 2023. https://arxiv.org/html/2402.16480v1.

[10] GUO X, et al. Deepseek-coder: Extending context windows for code generation[EB/OL]. 2024. https://arxiv.org/pdf/2401.14196.

[11] 作者待定. How I write code using Cursor: A review[EB/OL]. 2023. https://www.arguingwithalgorithms.com/posts/cursor-review.html.

[12] 作者待定. ChatGPT's Code Interpreter a Giant Leap Forward[EB/OL]. 2023. https://aigenealogyinsights.com/2023/07/11/first-blush-chatgpts-code-interpreter-a-giant-leap-forward/.

[13] STACKBLITZ. bolt.new: Ai-powered full-stack web development in the browser[Z]. 2024.

[14] LABS S. bolt.diy: Open source ai coding assistant[Z]. 2024.

[15] MATHEWS A, NAGAPPAN N. Test-driven development for code generation[EB/OL]. 2024. https://arxiv.org/abs/2402.13521.

[16] HONG J, et al. Metagpt: Meta programming for a multi-agent collaborative framework [EB/OL]. 2023. https://ar5iv.org/pdf/2308.00352.

[17] LIN P, et al. When llm-based code generation meets the software development process [EB/OL]. 2024. https://ar5iv.org/pdf/2403.15852.

[18] INITIATIVE O. Openapi specification[EB/OL]. 2023. https://spec.openapis.org/oas/latest.html.

[19] INITIATIVE O. Openapi specification version 3.1.0[EB/OL]. 2023. https://spec.openapis.org/oas/v3.1.0.

附录 A　外文资料的书面翻译

面向软件工程的大型语言模型：综述与开放问题

(Large Language Models for Software Engineering: Survey and Open Problems)

目录

A.1　摘要 46
A.2　简介 47
A.3　开场白 49
　A.3.1　大型语言模型 49
　A.3.2　大型语言模型的类别 50
　A.3.3　面向软件工程的大型语言模型 50
A.4　需求工程与设计 51
　A.4.1　需求工程 LLMs 中的开放问题 51
A.5　代码生成和补全 52
　A.5.1　代码生成模型 52
　A.5.2　改进的 Prompt 工程代码生成 54
　A.5.3　LLM 和其他技术的混合体 54
　A.5.4　基于 LLM 的代码生成的科学评估 55
　A.5.5　代码生成和补全中的开放性问题 57
A.6　软件测试 58
　A.6.1　使用 llm 生成新测试 58
　A.6.2　测试充分性评估 60
　A.6.3　测试最小化 61
　A.6.4　测试输出预测 61
　A.6.5　测试不稳定性 61
　A.6.6　用于软件测试的 LLMs 中的开放问题 61
A.7　维护、演进和部署 63
　A.7.1　调试 63
　A.7.2　程序修复 63

A.7.3 性能改进 65
A.7.4 克隆检测和复用 67
A.7.5 重构 67
A.7.6 维护和演化中的开放性问题 68
A.8 文档生成 69
A.8.1 文档生成和代码摘要的开放性问题 69
A.9 软件分析和存储库挖掘 69
A.10 人机交互 70
A.11 软件工程过程 70
A.12 软件工程教育 71
A.13 横切开放研究主题 71
A.13.1 为 SE 构建和调优 llm 71
A.13.2 需要动态自适应提示工程和参数调整 72
A.13.3 杂化 72
A.13.4 控制幻觉 72
A.13.5 稳健、可靠、稳定的评估 73
A.13.6 全面测试 73
A.13.7 处理较长的文本输入 74
A.13.8 软件工程覆盖较少的子领域 74
参考文献 74

A.1 摘要

本文综述了近年来出现的面向软件工程（SE）的大型语言模型（LLMs）。它还为 LLMs 应用于软件工程师面临的技术问题提出了开放的研究挑战。LLMs 的新特性为软件工程活动带来了新颖性和创造性，包括编码、设计、需求、修复、重构、性能改进、文档和分析。然而，这些相同的新特性也带来了重大的技术挑战；我们需要可靠的技术来排除不正确的解决方案，例如幻觉。调查揭示了混合技术（传统 SE + llm）在开发和部署可靠、高效和有效的基于 llm 的 SE 中必须发挥的关键作用。

A.2 简介

本文综述了基于 LLM 的 SE 的最新发展、进展和实证结果；大型语言模型（LLMs）在软件工程（SE）应用中的应用。我们利用这项调查来强调这一迅速发展但仍处于萌芽状态的研究文献中的差距。基于文献和技术机会的差距，确定了软件工程研究界的开放问题和挑战。

虽然对这样一个快速扩展的领域的任何调查都不能期望或声称是全面的，但我们希望这项调查将提供一个有用的和相对完整的早期宇宙的软件工程这个令人兴奋的新分支学科：基于 LLM 的软件工程。虽然该领域的科学和技术结构仍在发展中，但已经能够确定未来研究的趋势、生产性途径和需要解决的重要技术挑战。

特别是，我们已经能够识别与软件工程中现有趋势和完善的方法和子学科的重要联系（并与之产生共鸣）。此外，尽管我们找到了相当多的乐观的理由，仍然存在着重要的技术挑战，这些挑战可能会在未来几年影响研究议程。许多作者强调，科学和轶事，幻觉是一个普遍的问题，LLM [1]，也为基于 LLM 的 SE [2] 提出了具体的问题。与人类智能一样，幻觉意味着 LLM 可以创造虚构的输出。在软件工程的环境中，这意味着工程制品可能是不正确的，但看起来是合理的；LLM 可能会引入 bug。

然而，与 LLM 的许多其他应用不同，软件工程师通常拥有可自动化的基本事实（软件执行），可以根据这些事实评估大多数软件工程制品。此外，软件工程研究社区已经投入了大量时间来生成自动化和半自动化技术，以检查人类产生的潜在错误结果。这意味着，对于学科和研究界来说，在应对幻觉等问题带来的挑战时，有大量的经验和专业知识可供借鉴。

显然，自动化测试技术 [3]–[5] 将在确保正确性方面发挥核心作用，就像它们已经为人类工程的工件所做的那样。当生成全新的特性和系统时，自动化测试数据生成会受到缺乏的影响——一个可自动化的 oracle [6]（一种自动技术，用于确定给定输入刺激的输出行为是否正确）。鉴于 LLM 的幻觉倾向，Oracle 问题将保持高度相关性，它的解决方案将变得更加有影响力 [7]。

然而，一些社会工程应用涉及对现有软件系统的改造、改进和开发，为此有一个现成的自动化预言：原始系统的功能行为。在本文中，我们称其为“自动回归 Oracle”，这是一种在遗传改良领域已被证明具有优势的方法 [8]。自动回归 Oracle 只是使用软件系统的现有版本作为基准，对任何后续调整和更改的输出进

行基准测试。

当然，存在“烘焙”函数错误的风险，因为自动回归 Oracle 不能检测系统应该做什么，而只能捕获它当前做什么。因此，自动化回归 Oracle 只能测试功能回归，因此，它最适合于需要维护现有功能的用例。例如，用于非功能的改进，如性能优化和保留语义的重构。

提供给 LLM 的输入自然会成为不断增长的研究的重点，我们可以期待关于 Prompt Engineering 和 Prompt Optimisation 的文献的快速发展 [9]。本文强调了在软件工程的几个特定方面，Prompt Engineering 的现有工作和开放挑战。

LLM 的输出不需要仅仅局限于代码，还可以包括其他软件工程制品，如需求、测试用例、设计图表和文档。一般来说，LLM 基于语言的性质允许它生成任何语言定义的软件工程制品。

我们通常认为软件工程人工制品是 LLM 的主要输出，但它不是唯一的输出。初级输出提供的解释也是任何 LLM 的重要输出。本文综述强调了需要进行更多研究，不仅要优化 Prompt Engineering（关注于 LLM 的输入），还需要对初级输出提供的解释进行优化。

LLM 本质上是不确定性的：相同的提示在不同的推理执行上产生不同的答案（除非温度设置为零，在多次执行中通常被发现是次优的）[10]。此外，无论温度设置如何，提示中的细微变化都可能导致非常不同的输出 [10]。以及激励“提示工程”和输出处理，这种不确定性行为为基于 LLM 的软件工程的科学评估提出了挑战：

- 如果每次运行过程的结果都可能不同，我们如何确定所提出的技术是否比最先进的技术取得了进步？

这个问题已经在经验软件工程 [11] 和基于搜索的软件工程（SBSE）[12] 的背景下得到了很好的研究。特别是，SBSE 与基于 LLM 的软件工程有许多相似之处，它们都需要在存在噪声、不确定和不完整结果的情况下实现鲁棒的科学评估 [13], [14]。因此，已经有了关于健壮的科学评估技术的成熟的软件工程文献需要迎合基于 LLM 的科学评估。

例如，经过充分研究的技术，如参数和非参数推断统计，现在通常用于在 SBSE 学科中存在高度不确定性算法的情况下提供可靠的科学结论。

为了了解基于 LLM 的软件工程的增长趋势，我们对 arXiv 中特定主题的出版物数量进行了人工分析。表 I 包含原始数据 [15]，这些数据是从 Kaggle（https://www.kaggle.com/datasets/Cornell-University/arxiv）公开的 arXiv 元数据转

储中手动提取的，可在 2023 年 7 月 27 日访问。我们首先过滤掉了分类代码不以 CS 前缀开头的出版物（即计算机科学），结果为 A 列。

识别与 LLM 相关的计算机科学论文，我们将出版物筛选为子类别人工智能（CS.AI）、机器学习（CS.LG）、神经与进化计算（CS.NE）、软件工程（CS.SE）和编程语言（CS.PL）使用标题或摘要中的查询“大型语言模型”、“LLM”和“GPT”（我们手动排除了过多的缩写实例，如通用规划工具的 GPT），结果列 L。最后，使用相同的查询来识别软件工程（CS.SE）和编程语言（CS.PL）中基于 LLM 的软件工程论文。这些查询本质上是近似的，因此我们只局限于基于有强有力证据的总体趋势的结论，而不是观察到的数字的具体细节。然而，我们报告了观察到的原始数字，以支持其他人的复制。

由于这种增长，我们可以期待许多基于 LLM 的 SE 的其他调查。文献的快速扩展使得进一步全面的全社会研究不太可能适应一篇论文的空间限制，但我们可以期待许多感兴趣的子领域的具体全面调查，以及系统文献综述（SLRs），通过在系统综述中提出主要文献的具体研究问题来解决全社会的横切问题。这种 SLR 已经出现了。例如，Hou 等人 [16] 最近提供了一份优秀的 SLR，涵盖了 2017 年至 2023 年 229 篇研究论文，报告了所解决的 SE 任务、数据收集和预处理技术以及优化 LLM 性能的策略（如 Prompt Engineering）。

本文的其余部分按照顶层软件开发活动和研究领域进行组织，如图 1 所示。

A.3 开场白

A.3.1 大型语言模型

大型语言模型（LLM）是指在大量数据上训练过的人工智能（AI）模型，能够以类似人类的方式生成文本。表 III 提供了一个 LLM 术语表，使论文自成一体。

LLM 通常基于深度学习技术，如 Transformer，并有能力生成有用的语言输出。因此，人们发现它们能够执行广泛的语言相关任务，包括文本生成 [16]、回答问题 [17]、翻译 [18]、摘要 [19] 和情感分析 [20]。

Rumelhart 等人 [21] 引入了循环神经网络（RNN）的概念，开辟了处理序列数据的可能性。长短期记忆（LSTM）架构是 Hochreiter 和 Schmidhuber 提出的 RNN 架构的扩展 [22]，显著提高了它们在许多应用中的性能。

2017 年，Vaswani 等人 [23] 介绍了 Transformer 架构，该架构通过自注意力机制捕获单词关系。Transformer 架构对语言建模产生了深远的影响，并引发了

LLM 领域的激增。

2018 年，OpenAI 发布了生成式预训练 Transformer（GPT）模型，随后进行了后续迭代（GPT-2、GPT-3、GPT-3.5 和 GPT-4）。对于 GPT-3 和 3.5，许多观察者注意到生成性能的显著变化，从而吸引了人们对 GPT（特别是 ChatGPT）以及更广泛的 LLM 的极大兴趣。

LLM 能够达到这种性能，部分原因是它们训练的语料库很大。例如，GPT-3 是在 45TB 的文本数据上训练的，有 1750 亿个参数。Meta 在 2023 年 2 月推出了开源的 LLaMA，它在 1.4 万亿个 token 上进行训练，各种模型大小从 70 亿到 650 亿参数 [24]。

A.3.2 大型语言模型的类别

大型语言模型通常可分为三类：

1. **纯编码器模型**：也称为自编码器，由编码器网络组成，但没有单独的解码器网络。它接收一个输入序列并将其映射到低维表示。自动编码器的目的是学习输入数据的编码。纯编码器的 LLM 示例有谷歌的 BERT、Meta 的 RoBERTa 和 Microsoft 的 DeBERTa [1]。
2. **编码器-解码器模型**：除了编码器网络之外，还有一个解码器网络，它通过基于上下文向量和之前生成的 token 迭代地生成 token 或符号来生成输出序列。它可以用于机器翻译或文本生成等任务。编码器-解码器 LLM 的例子是谷歌的 T5 和 Meta 的 BART [1]。
3. **仅解码器模型**：与前两种类型的 LLM 不同，仅解码器 LLM 没有处理输入数据的编码器组件，而只有一个解码器组件，该组件直接根据给定的上下文或输入生成输出序列。纯解码器模型通常基于自回归模型等架构，其中输出是逐 token 生成的。解码器生成的每个 token 都以之前生成的 token 和上下文为条件。纯解码器模型的流行示例是 OpenAI 的 GPT 系列（如 GPT-3、GPT-4）、Meta 的 LLaMA、Anthropic 的 Claude 和谷歌的 PaLM [1]。

A.3.3 面向软件工程的大型语言模型

虽然 LLM 已被广泛应用于涉及自然语言的任务，它们在软件开发任务中的应用，包括编程语言，最近也引起了极大的关注。

2021 年，OpenAI 推出了 CodeX，这是 GPT-3 的一个经过微调的后代。CodeX 被 GitHub 的 Copilot 使用，它为用户提供 Visual Studio Code、Visual Studio、Neovim 和 JetBrains 的代码补全。Copilot 的新版本 GitHub Copilot X [2] 是基于 GPT-4 的。

2023 年 2 月，GitHub 报告称，平均 46% [3] 的开发人员的代码是由 Copilot 编写的。仅对 Java 而言，这个数字是 62%。GitHub 的首席执行官 Thomas Dohmke 预测 Copilot 将在 2023 年 6 月编写 80% 的代码 [4]。

在 2022 年，DeepMind 引入了 AlphaCode [5]，使用 40B 参数进行训练选定的公共 GitHub 仓库。在有 5000 多名参与者参加的模拟评估中，它平均排名前 54%。

最新的 GPT 模型 GPT-4 也执行代码生成。根据 GPT-4 的技术报告 [6]，使用 GPT-4 在 HumanEval 上的零样本 pass@1 准确率为 67%，HumanEval 是 OpenAI 的一个开源数据集，由 164 个编程问题组成。

在 100 个 LeetCode 问题的基准测试中，GPT-4 的性能与人类开发人员相当 [7]。在 2023 年 8 月 24 日，Meta 发布了开源代码 LLaMA [8]，这是公开可用的 LLM 在编码任务上的最新技术。

表 II 列出了为基于自然语言描述的代码生成/完成而设计的代表性 LLM。

A.4 需求工程与设计

需求工程是软件工程中的一门重要学科。它形成了软件工程师所构建的系统的技术属性与构建系统的目的之间的基本联系。有一个成熟的文献和一个大型的研究社区专门关注与需求工程问题相关的问题 [31]。

以前也有关于人工智能方法的工作来支持需求工程，特别是以计算搜索需求工程的形式 [32]。然而，迄今为止，基于 LLM 的软件工程的新兴文献对需求工程学科的关注较少。

Zhang 等人 [33] 对 ChatGPT 在四个数据集上的两个需求分析任务上的零样本需求检索性能进行了初步评估。尽管这些结果只是初步的，但它们提供了乐观的看法，即 LLM 可以用作高效和有效的需求工程的支持。罗等人 [34] 利用 BERT 进行 Prompt Engineering，实现需求的自动分类。Luitel 等人 [35] 专注于需求的完整性，并使用 BERT 生成填补需求中被掩盖的位置的预测。

A.4.1 需求工程 LLMs 中的开放问题

与其他软件开发活动不同，我们没有在基于 LLM 的需求工程或基于 LLM 的设计方面找到太多工作。事实上，甚至有证据表明，实践中的工程师不愿意依靠 LLM 来实现更高层次的设计目标 [36]。因此，有一个很好的机会来扩展这个开放的研究领域。

大多数 LLM 应用程序专注于代码生成、测试和修复等任务。这些任务受益

于 LLM 生成代码的能力。然而，LLM 也有很大的支持潜力需求工程活动，得益于其自然语言处理能力。

例如，可追溯性是一个长期存在的，交叉的关注软件工程。特别是，识别需求之间的可追溯性链接而其他工程人工制品，如代码和测试，尤其具有挑战性，因为需求通常是用自然语言编写的语言；天生适合 LLM。

A.5 代码生成和补全

LLMs 的所有软件工程应用领域，代码补全是迄今为止探索得最彻底的领域。甚至在 LLM 出现之前，就有人认为从现有的代码存储库中学习是成功和智能的代码补全的关键 [37]：预训练的 LLM 实现了这些代码补全的早期愿望。

虽然幻觉被认为是 LLM 的弱点，但代码补全的特定任务通过向开发人员提供推荐系统来回避幻觉问题。因此，开发人员有责任在任何幻觉的 LLM 输出泄漏到代码库之前清除它。

当然，高度的幻觉会使代码补全推荐系统无效。代码补全的广泛和快速采用，以及已经报告的积极结果，提供尚未发生这种情况的早期迹象。例如，Murali 等人 [38] 报告了在 Meta 上部署 codecomposition 的经验，codecomposition 是一个基于 Incoder LLM [39] 的代码补全工具。在 15 天的时间里，codecomposition 提出了 450 万条代码补全建议，开发人员的接受率为 22%。质量反馈是高度积极的，92% 的人是积极的。类似地，Peng et al. [40] 报告称，与没有获得任何此类支持的对照组相比，程序员在使用 GitHub Copilot 时可以更快地完成一项重要任务（用 JavaScript 实现 HTTP 服务器）56%。

许多软件工程师似乎已经认定，好处超过任何必要的人工过滤工作，人们的热情程度和采用率已经被报道了。一旦基于 LLM 的代码完成被完全采用，人们期望程序员将花费更多的时间审查而不是编写代码 [41]。

A.5.1 代码生成模型

自动代码生成有着悠久的历史，它的起源可以追溯到自动程序合成的早期愿景 [42]，这些愿景一直在继续发展并产生了令人印象深刻的结果 [43]。

来自 Hindle 等人关于软件自然性的开创性工作 [44]，我们知道程序员编写代码（和语言强制代码编写风格），使代码高度可预测。此外，Barr 等人 [45] 发现对大型 Java 项目仓库的提交有 43% 可以从现有代码中重构。他们将其称为“整形手术假说”，因为自动修复是通过清除现有代码来修补其他地方的问题 [46]。

他们的实证研究为这种清除方法的有效性提供了证据，但也强调了软件的重复性和可预测性。在更大的存储库（sourceforge）中，Gabel 和 Su [47] 发现一个程序员必须要写超过六行代码才能创建一个新颖的代码片段。

这些关于代码自然性、可重用性和可预测性的研究发现，使 LLM 能够做到这一点并不奇怪，利用相同的可预测重用性来为代码生成提供有效的建议。这些观察结果支持了修复和遗传改良的“生成-测试”方法的增长 [8], [46]。生成-测试方法提供了更大的代码转换自由（与更传统的通过构造来纠正的方法相比 [48]），这正是因为生成的代码可能无法保持严格的、数学定义的（并不总是适当的，也不是有用的）对正确性的解释。

这种探索“语义近邻”更广阔空间的自由，允许基因改进找到戏剧性的优化（见 VI-C 节）。遗传改进方法、命名法和评估方法还提供了一个科学框架，用于理解和评估基于 LLM 的代码生成。两种技术分享程序转换和代码生成的“生成-测试”方法，有可能使许多现有的基因改良工作直接适用到基于 LLM 的代码生成。

2021 年，Chen 等人 [49] 介绍了 CodeX，一个在 GitHub 公开代码上进行微调的 GPT 语言模型，并评估了它的 Python 代码编写能力。他们发布了一个名为‘HumanEval’的新评估集，以衡量从文档字符串合成程序的功能正确性，并发现 CodeX 在解决这些问题时表现优于 GPT-3 和 GPT-J。从那时起，关于基于 LLM 的代码生成的研究呈爆炸式增长，HumanEval 数据集已被用于许多后续研究。

在 2022 年，Li 等人 [27] 引入了 AlphaCode，这是一个用于代码生成的系统，为竞争性编程问题创造了新的解决方案。他们发现有三个关键组件对于实现可靠的性能至关重要：

1. 用于训练和评估的广泛编程数据集。
2. 大型且高效采样的基于 Transformer 的架构。
3. 大规模模型采样以探索搜索空间，然后是基于行为的过滤。

在 Codeforces 平台上编程比赛的模拟评估中，AlphaCode 在超过 5000 名参与者的比赛中平均取得了前 54% 的排名。

几篇论文还介绍了代码合成 LLMs [50]–[53]，基于很少对训练数据进行预过滤的大型数据集。然而，在 2023 年，Gunasekar 等人 [54] 报告称，通过仅使用教科书质量的代码语料库进行训练，具有较低参数数量的 LLM 可以实现与大得多的模型相当的性能。

他们使用 GPT-4 模型对现有的 Python 代码语料库进行分类，促使它确定给定代码对想要学习编程的学生的教育价值。其次，他们使用 GPT-3.5 创建关于

Python 的合成教科书。特定的代码生成用例也已经被解决，例如数值算法代码生成 [55]，以及从行为描述生成代码 [56]。现有 LLM 用于代码生成和代码生成排行榜的更多示例可以在表 II 和图 4 中找到。

A.5.2 改进的 Prompt 工程代码生成

提示工程被广泛用作改进代码生成的一种方法。例如，Li 等人 [57] 报道 Pass@1 改进了大约 50% 到 80% CodeX、CodeGeeX、CodeGen 和 InCoder 的几个基准测试（Python 的 MBPP、Java 的 MBJP 和 JavaScript 的 MBJSP）。Döderlein 等人 [58] 报告了 Copilot 和 CodeX 在 HumanEval 和 LeetCode 上的成功率从大约 1/4 到 3/4 的快速工程改进。他和 Vechev [59] 使用 Prompt Engineering 来提高 LLM 生成代码的安全性，安全性从 59%（考虑的情况）提高到 92%。White 等人 [60] 为各种软件工程任务（包括代码生成）提供了提示工程设计模式的目录。Denny 等人 [61] 认为，Prompt Engineering 是一种有用的学习活动，可以培养软件工程学生的计算思维。

其他作者考虑了将提示工程分解为与 LLM 迭代和多阶段对话的方法，使其更接近思维链推理。例如，Li 等人 [62], [63] 报告说，使用两阶段基于草图的方法，即 SkCoder，LLM 首先创建草图，然后随后实现这些草图，ChatGPT Pass@1 增加了 18%。Jiang 等人 [64] 和 Zhang 等人 [65] 也试图通过促使 LLM 反思和自我编辑来部署思维链式的推理。

现有的软件工程分析技术还可以为微调和提示工程提供额外的信息。例如，Ahmed 等人 [66] 展示了如何在提示中使用简单的静态分析，以通过少样本学习提高代码生成的性能。

Shin 等 [67] 比较对代码生成任务使用 GPT-4 进行提示工程和微调，表明微调比提示工程工作得更好。

A.5.3 LLM 和其他技术的混合体

通过对文献的调查，我们发现了强有力的证据，一些最有希望的结果可以通过混合实现；将 LLMs 与其他现有的软件工程技术相结合。本节概述用于代码生成的混合 LLM 工作。

一些作者开发了 LLM 与规划和搜索相结合的混合体。例如，Zhang et al. [68], [69] 报告称，与基线相比，性能提高了约 11% 到 27% Zhang et al. [70] 使用 API 搜索技术的混合代码生成。

混合方法也有使用现有的软件工程和/或人工智能技术从 LLM 的 top-n 输出

中选择最佳候选。例如，Chen 等人 [71] 使用测试生成来选择候选者，并报告了大约 20% 的改进 5 个预训练的 LLM；Inala 等人 [72] 使用基于神经网络的排序器预测代码的正确性和潜在的错误。Jain 等人 [73] 提出了 Jigsaw，基于对生成的代码进行后处理程序分析与综合技术。

Dong 等人 [74] 将 LLM 视为代理，让多个 LLM 在协作和交互式地处理代码生成任务方面发挥着不同的作用。他们报告了大约 30%-47% 的改善。

A.5.4 基于 LLM 的代码生成的科学评估

迫切需要进行更彻底的科学评估。许多作者都曾报道过 LLM 无法生成正确、安全和可靠代码的案例。Poldrack 等人 [75] 也强调了大量人工验证的必要性。在本节中，我们从正确性方面综述了基于 LLM 的代码生成的经验评估的文献，稳健性，可解释性，决定论，以及安全。

A.5.4.1 正确性评估

GPT-4 技术报告 [28] 上评估了 GPT-4 代码生成的正确性 HumanEval 数据集的零样本准确率为 67%，略有提高 Yetistiren 等人报道的（早期 ChatGPT）结果 [76]。

Borji [77] 对 ChatGPT 的 LLM 代码生成失败进行了严格、分类和系统的分析。在他们的工作中，介绍和讨论了 11 类失败，包括推理、事实错误、数学、编码和偏差。

图 4 根据 Papers With Code（一个突出 AI 研究趋势以及方法和模型背后的代码的平台），显示了在 HumanEval 数据集上 Pass@1（即 Top-1 代码候选的测试通过率）代码生成正确性的排行榜。每个方法背后的 LLM 模型都显示在括号中。在撰写本文时，最佳的代码生成模型 Reflexion [78] 可以为 90% 以上的生成任务生成正确的代码。然而，在这样一个快速发展的领域中，这些数字和不同语言模型的相对排名必然会发生变化。例如，原始 GPT-4 报告 [28] 中给出的 HumanEval 上正确代码的数字仅为 67%，因此更新后的数字为 80%（写作本书时，也就是 5 个月后）。

从 Papers-With-Code 网站检索到的数据大概代表了 GPT-4 自那时以来的演变。尽管在代码生成和补全的文献中有很好的结果，Din 等人 [79] 报告称，当上下文包含 bug 时，HumanEval 的代码补全性能下降了 50% 以上。

A.5.4.2 鲁棒性评估

LLM 代码生成的鲁棒性是相似的提示引起语义和语法相似的代码生成的程度。Treude [80] 介绍了 GPTCOMPARE，一个原型工具，用于视觉上突出 LLM 代码输出之间的相似性和差异性。Yan 等人 [81] 引入 COCO 来测试基于 LLM 的代码生成系统的健壮性和一致性。

A.5.4.3 可解释性评估

与之前的机器学习技术相比，LLM 的一个相当大的优势是代码生成工件伴随着解释的方式。通过提供额外的信心和更快的理解，这些解释有可能增加采用率。需要更多的工作来评估和优化伴随生成的代码和其他软件工程工件的解释。

MacNeil 等人的初步评估 [82] 在他们的交互式 Web 开发电子书上，表明大多数学生认为 LLM 生成的代码解释是有帮助的。Noever 和 Williams [83] 还展示了解释的潜力，以帮助人类工程师，特别是在代码混淆或缺乏足够的现有文档的情况下。通过这种方式，产生洞察力和解释的能力可能不仅仅是证明 LLM 本身生成的代码，而是可能成为有价值的教育和文档来源（见 XI 部分）。

Sun 等人 [84] 关注了用户对生成式 AI 在三个软件工程用例中的可解释性需求：基于自然语言描述的代码生成（使用 Copilot），不同编程语言之间的翻译（使用 Transcoder），以及代码自动补全（使用 Copilot）。他们的调查是由 43 名软件工程师参加的 9 个研讨会进行的，并确定了在生成式人工智能（GenAI）背景下对代码的可解释性需求的 11 类。

A.5.4.4 决定论评价

LLM 是不确定性的。欧阳等人 [10] 对 ChatGPT 在代码生成中的不确定性进行了实证研究，发现超过 60% 的任务在不同的请求中没有相同的测试输出。然而，他们对文献的研究表明，在基于 LLM 的代码生成研究中，只有 21.1% 的论文在实验中考虑了不确定性威胁。

A.5.4.5 安全评估

Hajipour 等人 [86] 提出了一种小样本提示的方法来检测安全漏洞，报告称他们的方法在几个模型中自动发现了数千个安全漏洞。Khoury 等人 [87] 发现 ChatGPT 生成的代码经常远远低于安全编码的最低标准。Rise 和 Bőhme [88] 报告的结果表明，由于模型对不相关的训练集特征过拟合，漏洞检测精度可能被过度报告。

此外，Yetistiren 等人 [76] 对其性能进行了综合评价，包括 Copilot、CodeWhisperer 和 ChatGPT，涵盖不同方面包括代码有效性、代码正确性、代码安全性和代码可靠性。他们的结果显示，在表现上有很大程度的差异，激发了进一步研究和调查的需要。例如，他们发现 65%、46% 和 31% 的程序分别由 ChatGPT、Copilot 和 CodeWhisperer 生成是正确的。

A.5.4.6 基准测试

与其他科学评估一样，软件工程评估依赖于公开可用且具有代表性的基准套件。其中一些已经出现，可以支持基于 LLM 的应用程序的软件工程评估。带代码的论文平台 [5] 提供了 15 个用于评估代码生成的基准的摘要。

评估通常依赖于编程课程 [89] 中的小编程问题、综合生成的问题集 [90]，以及 LeetCode [29], [65], [91] 等在线评判平台。尽管在训练集上不同的 LLM 报告的结果自然不同，但这些评估的总体结论表明，成功率在 20% 到 80% 之间。

然而，现有的代码生成基准往往依赖于测试套件来自动判断代码的正确性，这可能不够充分，导致错误判断 [92]。这突出表明需要对评估基准进行更多工作，以专门为基于 LLM 的代码生成评估量身定制。Liu 等人 [93] 引起了对这个问题的关注，展示了现有的测试套件如何导致高程度的假阳性结论（对于在线评判平台 [92] 也是一个严重的问题）。为了缓解这个问题，他们提出了 EvalPlus ——一个代码合成基准框架，可以自动生成测试输入，并严格评估 LLM 生成代码的功能正确性。他们对 14 个流行的 LLM（包括 GPT-4 和 ChatGPT）的评估表明，使用为 HumanEval 新生成的测试，Pass@k 的评估在考虑的问题上平均下降了 15%。

Jimenez 等人 [94] 介绍了 SW-Bench，目的是在现实的软件工程环境中评估 LLM 的代码生成问题。SW-Bench 包含 2294 个软件工程问题，来自真实的 GitHub 问题。实验结果表明，Claude 2 和 GPT-4 分别只解决了 4.8% 和 1.7% 的编码任务。

A.5.5 代码生成和补全中的开放性问题

评估生成的代码仍然是基于 LLM 的代码生成和补全的关键问题。虽然许多工作已经开始将现有的软件测试知识应用于此问题，但我们希望自动化测试技术与代码生成和补全技术更紧密地集成。

幸运的是，有大量关于自动化测试数据生成的现有工作 [3]–[5]，其中许多工作将在确保 LLM 生成的工程工件的正确性方面发挥重要作用。本文中涉及的挑战的一个反复出现的主题是，代码执行精确地提供了过滤幻觉响应所需的“基本事实”。它还可以作为交互推理/行动（‘ReAct’）对话 [95] 的一部分，在 LLM 之

间和内部提供指导。

自动化的测试数据生成允许软件工程师以探索运行时基本事实的最相关区域为目标。这种基于测试的目标可以帮助过滤、微调和优化提示，从而将幻觉带来的问题最小化。LLM 在自动化构建有效和高效的软件测试套件的过程中也具有相当大的潜力。

另一个重要问题是如何有效地微调预训练的 LLM，以便它们对特定的编程语言、代码库或领域表现更好。这一点尤其重要，因为从头开始训练 LLM 需要大量的计算资源。例如，当特定编程语言的训练示例数量不足时，迁移学习被提出作为一种提高代码补全性能的方法 [96]。

目前的研究重点是由 LLM 产生的代码。然而，LLM 产生的解释可能至少同样重要。人们可以想象，在许多情况下，工程师宁愿接受一个（可能）带有令人信服的解释的次优软件工程工件，而不是一个具有不那么令人信服的解释的潜在性能更高的解决方案。毕竟，工程师经常对人类设计的工程工件做出相同的判断，那么我们为什么会期望机器生产的产品有任何不同呢？与专注于优化 LLM 输入的 Prompt Engineering 一样，** 解释工程 ** 本身也可能成为一个研究领域。

A.6 软件测试

软件测试是一门成熟的研究学科可以追溯到图灵在 20 世纪 40 年代末的开创性工作 [97]。这项研究的重点主要集中在测试套件的自动生成上，能够以较低的计算成本实现高错误揭示潜力 [3]–[5]。这不仅为我们提供了能够淘汰不正确的 llm 生成代码的技术，而且还为我们提供了一个成熟的基线，用于比较新的基于 llm 和混合的测试套件生成技术。

已经有足够多的工作需要对基于 llm 的软件测试进行调查：Wang 等人 [98] 介绍了以测试为主，但也包括调试和修复的论文综述。他们报告了 52 篇论文（33 篇自 2022 年以来发表），其中大约三分之一涉及基于测试的 LLM 微调，而其余的依赖于快速工程。

A.6.1 使用 llm 生成新测试

在本节中，回顾了用于测试数据生成的 llm 的现有工作，然后强调了这一新兴领域发展的开放问题和挑战。生成的测试可能无法执行，因为 LLM 不保证生成可编译的代码。Nie 等人 [99] 报告使用 TeCo 生成的 29％的测试是可执行的，同时 Yuan 等人 [100] 发现，大约四分之一的测试由 ChatGPT 生成是可执行的，在

适当的提示工程下上升到三分之一。

在那些能够编译的测试中，一些作者报告了所实现的代码覆盖率。例如，Bareiß 等 [101] 报告说，从 10％的使用实现了增长 Randoop[102] 到 14％与 CodeX。Hashtroudi et al.[103] 报告说，通过微调代码 5 生成的测试代码的行覆盖率增加了 50％。Siddiq 等人 [104] 报道 80％的覆盖率使用 CodeX 的 HumanEval 数据集，但也发现没有被研究的 llm 能够在 EvoSuite SF110 数据集上达到 2％以上的覆盖率。

将现有的测试生成和评估技术（如基于模糊测试和基于搜索的测试）与 llm 相结合的混合方法已经显示出良好的效果。例如，Lemieux 等人 [105] 介绍了 CODAMOSA，一种结合基于搜索的软件测试（Search-Based Software Testing，SBST）[5] 和 CodeX 的算法，为被测程序生成高覆盖率的测试用例。当 SBST 的覆盖率改进停滞时，CODAMOSA 要求 CodeX 为未覆盖的功能提供示例测试用例。这有助于 SBST 将其搜索重定向到搜索空间中更有用的区域。在对 486 个基准的评估中，CODAMOSA 实现了比仅 SBST 和 llm 基线更高的覆盖率。

Hu 等人 [106] 引入了 ChatFuzz，它用 ChatGPT 对 AFL 进行了扩充，以获得更多符合格式的变异体。在三个基准测试程序中选取 12 个目标程序进行测试，ChatFuzz 的分支覆盖率比 AFL 高出 13％。Dakhel 等人 [107] 使用变异测试来帮助 llm 生成测试。特别是，他们增强 Codex 和 llama-2 聊天提示与幸存的变种人。他们报告说，他们的方法检测到的人为错误高出 28％。

Xia 等人 [108] 最近证明，LLMs 可以作为跨不同应用程序域和编程语言的系统的通用模糊器，包括 C/C++ 编译器，JDK，SMT 求解器，甚至是量子计算系统。Deng 等人 [109] 提出了 TitanFuzz，它使用 LLMs（即 Codex）来生成有效的输入 DL 程序来测试 DL 库。在 PyTorch 和 TensorFlow 上的结果表明，TitanFuzz 可以实现代码覆盖率比最先进的模糊测试器高 30％/51％。后来，他们进一步引入了 FuzzGPT[110]，它可以综合不寻常的程序来对 DL 库进行模糊测试。他们的结果表明，当重新针对基于模糊测试时，CodeX 和 CodeGen 在 PyTorch 和 TensorFlow 上的性能优于 TitanFuzz。

Li 等人 [111] 使用了差分测试和 ChatGPT 的混合，为了提高后者对缺陷程序生成诱发失效的测试用例的能力。他们报告说测试效率从 29％提高到 78％。

基于 llm 的测试生成的一个有前途的领域是 GUI 测试，因为通过 GUI 操纵应用程序状态通常需要对用户界面和应用领域。Sun 等 [112] 通过文本描述用户界面，并询问 ChatGPT 根据文本下一步想执行什么操作，然后将答案转换为实际的 GUI 交互。相比之下，活动覆盖率提高了 32％到最先进的水平。

对于经典技术来说，一个特别重要且具有挑战性的问题是从用户报告中构造测试用例。用户报告是用自然语言编写的。这对现有技术提出了相当大的挑战，但非常适合 llm。Kang 等人 [113] 介绍了 Libro，一种自动生成测试的少样本学习失败再现技术，基于 CodeX 的通用 bug 报告。天秤座成功复制大约三分之一的失败。

Feng 和 Chen[114] 使用带链的 LLM 开箱即用（ChatGPT），通过自然语言定义的重现步骤，证明了在 bug 报告上的重现率为 80%，思想本身就能促进工程。

一些作者考虑使用提示工程来改善测试生成的结果 [115], [116]。例如，Schafer 等人 [116] 提出了 TESTPILOT，它重新提示失败的测试和相关的错误消息，实现报告平均报表覆盖率为 68%。Xie 等 [117] 为测试生成创建提示解析项目并创建包含焦点方法及其依赖关系的自适应焦点上下文。他们进一步使用基于规则的修复来修复测试中的语法和简单编译错误。

尽管基于 llm 的测试结果可能不确定，研究人员已经探索了基于“自一致性”概念的交叉参考或多数投票 [118], [119] 方法来估计 llm 的置信度 [120]。例如，Kang 等人介绍的 Libro[113] 使用 CodeX 从可以重现故障的 bug 报告中生成测试。如果多个测试显示类似的失败行为，Libro 估计 LLM 对其预测是“有信心的”。此外，当存在部分 oracle 信息时，这也可以用于增强置信度估计。当整个过程的目标是改进现有代码时，这种部分 oracle 信息通常是可用的。例如，当提高现有测试的效率时，可以通过观察测试是否与原始测试行为相似（相同情况下是否通过）来收集自动化的部分 oracle 信息，并且执行速度也更快。

A.6.2 测试充分性评估

测试有效性通常根据“充分性标准”来衡量 [121], [122]。由于测试不能穷尽地探索每一种可能性，充分性标准提供了一套测试所达到的有效性的下限形式。变异测试是一种被广泛研究的评估软件测试套件充分性的技术 [123], [124]，其中故意注入合成错误（称为“变异体”）以评估测试充分性。变异测试已经被证明可以提供比其他基于结构覆盖的标准（如语句覆盖和分支覆盖 [125]）更严格的充分性标准。

变异测试中一个具有挑战性的开放问题是生成忠实地模拟现实世界中重要类别错误的变异体。Khanfir 等人 [126] 使用 CodeBert 生成类似开发者的突变体，发现他们的方法比 PiTest 具有更好的错误揭示能力。Garg 等人 [127] 应用 CodeBERT 来生成能够忠实捕获漏洞的突变体。他们评估发现 17%的突变体没有

通过测试，而这些测试分别被 89％的漏洞通过。Brownlee[128] 使用 GPT-3.5 生成用于遗传改进的突变体，并观察到基于 llm 的随机抽样编辑更容易编译和通过单元测试通常与标准的 GI 编辑比较。

A.6.3 测试最小化

测试最小化通过去除冗余的测试用例来提高软件测试的效率。Pan 等 [129] 应用要提取的代码有：CodeBERT、GraphCodeBERT 和 UniXcoder 嵌入测试代码以进行测试最小化。他们的方法达到了 0.84 的错误检测率，并且运行速度比基线快得多（平均 26.73 分钟）。

A.6.4 测试输出预测

Liu 等人 [130] 提出了 CodeExecutor，一种预训练的 Transformer 模型预测程序的整个执行轨迹。其目的是模仿现实世界中任意程序执行的行为。他们的评估比较了 CodeExecutor 和 CodeX，表明 CodeExecutor 在执行轨迹预测方面明显优于 CodeX（例如，教程数据集的输出精度为 76％vs. 13％）。

A.6.5 测试不稳定性

如果在执行上下文中没有任何明显的（测试人员可控的）变化，测试在某些情况下可以通过，而在其他情况下失败，那么测试就是不稳定的。测试不稳定是目前工业上最紧迫和影响测试效果的问题之一 [131]。llm 已被用于预测 flakiness，准确率很高（报告了 73％的 F1 分数 [132], [133] 和 97％的准确率 [134]）。

A.6.6 用于软件测试的 LLMs 中的开放问题

基于 llm 的软件测试数据生成存在许多开放性问题，它很好地掌握在现有的软件测试技术之内。因此，我们可以期待在未来几年中，基于 llm 的软件测试生成将出现令人兴奋的突破。本节概述了本研究议程的一些方向。

A.6.6.1 提示工程

一个好的软件测试有许多方面可以被合适的提示工程（prompt engineering）所优化。例如，我们需要了解如何设计提示：

- 预测并减少生成的测试不稳定性；
- 揭示可能的故障，例如通过对历史故障数据进行训练；
- 优化模拟测试和集成测试之间的平衡；
- 制作真实的数据构建器、模拟对象、参数和输入；

- 预测哪些测试最有可能引出覆盖边界情况的测试；
- 定制测试生成以关注生产中普遍存在的行为。

A.6.6.2 增加现有的测试

基于 llm 的测试生成的工作主要集中在新测试套件的自动生成。然而，考虑到现有的测试生成技术的多样性，仍然存在一个重要的（相对较少研究的）开放问题，即基于现有测试套件的扩充和再生 [135], [136]。

测试扩充和再生可以利用小样本学习和/或微调（在现有的测试数据集和历史错误上）来生成扩充的测试套件。llm 还需要进行更多的工作，以生成额外的测试断言，利用可用的训练数据来捕获边界情况、历史错误和可能的程序员错误。杂交 llm 和现有的自动化测试生成技术之间的关系也是一个富有成效的主题 [105]。

A.6.6.3 测试正确性

传统的软件测试生成受到 Oracle 问题的困扰 [6]，即，由于缺乏自动化的 Oracle 来确定测试结果是否正确，它们受到了限制。有两种情况与 ai 生成的测试有关：

1. 生成的测试在当前版本上通过：我们可以假设功能已经被正确地测试过，因此生成的测试作为一个回归测试，可以根据它检查未来的更改。
2. 生成的测试在当前版本中失败了：我们需要知道断言是错误的，还是生成的测试找到了 bug。

这两种情况都可能在缺乏自我调节的情况下产生有害的后果。通过的测试用例可能仅仅反映偶然的正确性 [137], [138]。更糟糕的是，代码实际上可能是不正确的（测试同样不正确，但捕获并执行了不正确的行为）。在这种情况下，生成的测试将倾向于阻止错误修复，因为在未来的修复中失败。这个问题也会影响 llm 生成的测试用例，并且在这些测试产生幻觉的情况下可能更有害，在生成的测试中嵌入这些不正确的 Oracle 断言。

另一方面，当生成的测试用例失败时，这可能表明存在 bug。这个 bug 的发现将意味着基于 llm 的测试的“胜利”。然而，如果假阳性比率远高于真正阳性比率，那么该技术的成本（例如，在人工评估中的成本）可能会使其不可行，即使它确实揭示了真正的 bug[131]。在置信度的自我评估，对生成的测试的正确性、一致性和健壮性的自我检查方面还需要更多的工作。我们需要开发自动评估、增强，并在向开发人员呈现“测试信号”之前，过滤执行基于 llm 的测试的原始结

果。

A.6.6.4 变异测试

还需要更多的工作来探索基于 llm 的测试生成的充分性，以及使用基于 llm 的技术来支持和增强测试充分性的调查和评估。llm 可以在故障模型上进行微调，从而用于建议与真实故障高度耦合的变异体，因此可以用于评估测试充分性。

A.7 维护、演进和部署

软件的维护和演化是一个重要的研究课题。它们关注现有的代码库，我们从这些代码库中寻求理解和业务逻辑提取，并为此寻求重新设计、修复和重构。诸如此类的维护问题都属于语言丰富的问题域。因此，正如我们在本节中回顾的那样，这一领域发现了许多基于 llm 的技术的应用并不令人惊讶。

A.7.1 调试

Kang 等人 [140] 研究了 GPT-3.5 的错误定位能力，并发现 LLM 通常可以在第一次尝试时识别出错误的方法。吴等人 [141] 对 GPT-3.5 和 GPT-4 在错误定位精度、稳定性和可解释性方面的能力进行了全面的研究。实验结果表明，GPT-4 达到了预期的错误定位精度，但当代码上下文变长时，性能会急剧下降。

Feng 和 Chen[142] 提出了 AdbGPT，它通过 ChatGPT 的提示工程，从缺陷报告中重现 Android 缺陷。在包含 88 个缺陷报告的数据集上，AdbGPT 能够成功重现 81％的缺陷报告，优于基线和消融。Joshi 等人 [143] 专注于多语言调试，提出了 RING，一种基于提示的策略，将修复概念化为本地化、转换和候选排序。

为了解决故障定位和程序修复中的数据泄漏威胁，Wu 等人 [144] 引入了与 1254 个 Java bug 和 1625 个 Python 错误之间的研究，涵盖 2021 年 10 月至 2023 年 9 月的数据。研究人员可以根据其创建周期选择代码样本，从而评估其有效性，根据不同的 llm 的训练数据截止日期。此外，也有使用 LLMs 预测 bug 严重性的工作 [145]。

A.7.2 程序修复

十多年来，在软件工程研究社区 [146], [147] 中，修复 bug 一直是一个非常有趣的话题，并且已经在最初的工业部署中找到了方法 [148]。

许多关于自动修复的工作使用了在遗传改进领域广泛采用的生成-测试方法，

并随时适用于基于 llm 的技术。因此，llm 肯定会对自动化软件修复产生积极影响，但正如我们在本节中报告的那样，在驯服幻觉问题和管理可扩展性方面仍然存在技术挑战。

为了实现可伸缩性，所有的生成和测试方法都需要解决构建时间问题 [149]。基于 llm 的修复也不例外；幻觉的倾向使测试阶段能够定期执行变得更加重要。很可能使用 ReAct 部署模型 [95] 将有助于找到高效和有效的工程权衡。当 ReAct 应用于修复时，整个方法将在‘Reason’阶段（生成候选修复）和‘Action’阶段（通过测试评估修复，这涉及构建问题）之间交替进行。

为了解决这个问题，我们可以参考关于软件修复的成熟文献 [46], [150]，这些文献建立在基于搜索的软件工程方法的二十多年发展的基础上 [12], [151]。该文献为研究社区提供了坚实的经验和专业知识基础，使其非常适合开发基于 llm 的生成和测试方法来解决问题。

最近的修复工作已经开始使用神经 AI 模型，如 Tufano 等人的开创性工作 [152]。最近，自 2022 年以来，关于基于 llm 的修复的新兴研究文献得到了快速发展。例如，Xia 等人 [153] 提出了 AlphaRepair。它将 APR 问题重新定义为一个填空（或填充）任务，其中 llm 可以根据潜在有错误的代码部分的双向上下文直接填充正确的代码。AlphaRepair 还首次证明了 llm 可以超过所有之前的 APR 技术。

他们进一步对使用三种不同语言的五个数据集的九个 llm 进行了实证研究 [154]。他们的发现不仅肯定了基于 llm 的 APR 的优越性（特别是完形风格的方法），而且提供了一些实用的指导方针。Wei 等人 [155] 通过合成一个补丁 LLM 和 Completion 引擎之间的交互，并发现该方法超过了表现最好的基线，修复了 14 和 16 个 bug。

程序修复自然适合 prompt 工程的对话模型。Xia 等人 [156] 提出 conversational APR，以对话的方式在补丁生成和验证之间交替进行。对 10 个 llm 的评估表明，该方法在效果和效率上都具有优越性。

他们进一步提出了 ChatRepair[157]，对话式方法修复了 337 个 bug 中的 162 个，每个 bug 的成本仅为 0.42 美元，因此也解决了所需计算资源的潜在问题。陈等人 [158] 介绍了自我调试，它教 LLM 通过少样本学习调试其预测的代码，自调试报告的基线精度提高了 12%。

例如，研究也报告了特定种类的修复结果，Pearce 等人 [159] 报告了五个商业 llm 对安全漏洞的修复结果，Charalambous 等人 [160] 将 ChatGPT 与形式化验证策略相结合，验证并自动修复软件漏洞。Cao 等人 [161] 报告了 ChatGPT 对深

度学习（DL）程序修复的结果。

修复并不总是从已存在的失败测试用例开始，而可以从生产环境中对失败的自然语言描述开始。自动化打开了对用户生成的 bug 报告做出更快响应的大门。Fakhoury 等人 [162] 的工作也探索了 llm 的修复路线，他们从自然语言问题生成了功能正确的代码编辑描述。他们提出了 Defects4J-nl2fix，这是一个由 283 个 Java 程序组成的数据集，来自具有 bug 修复的高级描述的 Defects4J 数据集。最先进的 llm 对此进行了基准评估，达到了高达 21％的 Top-1 和 36％的 Top-5 精度。

自动化修复还可以减轻工程师的负担，管理生产系统的 devops 式的随叫随到。例如，Ahmed 等人 [163] 研究了基于 llm 的微软云服务事件的根本原因和补救。他们使用 4 万起微软云服务事件数据，应用语义和词汇指标，在零样本、微调和多任务环境下评估了多个 llm，表明微调显著提高了事件响应的有效性。

针对特定任务或领域进行微调的能力可以显著提高程序修复中的模型性能。Jiang 等人 [164] 实证评估了 10 个不同的代码语言模型（CLMs）和 4 个故障基准的性能，并表明特定于修复的微调可以显著提高成功率。平均而言，10 个 CLMs 已经比最先进的基于 DL 的 APR 技术成功修复了 72％的故障。经过微调后，该数字增加到 160％。

Jin 等人 [165] 提出了 InferFix，其中包含一个在监督错误修复数据上进行微调的 LLM（Codex Cushman）。InferFix 在 Java 上实现了 76％的 Top-1 修复精度，在 C# 上使用 InferredBugs 数据集达到了 65％以上。Berabi 等人 [166] 介绍了 TFix，一种对错误修复数据进行微调的 T5 模型，报告称其性能优于现有的基于学习的方法。Xia 等人 [167] 结合了 LLM 微调和提示来自动化修复，并证明了他们的方法修复了 89 个和 44 个 bug（比基线高出 15 个和 8 个）。

llm 还可以帮助解释它们生成的补丁。Kang 等人 [168] 提出 AutoSD，用 llm 提供调试解释，以帮助开发人员判断补丁的正确性。他们发现 AutoSD 产生了与现有基线相当的结果，具有高质量的修复解释。Sobania[169] 研究了 GPT-3.5 在解释基于搜索的修复工具 ARJA-e 对 Defects4J 中的 30 个 bug 生成的补丁，84％的 LLM 解释被发现是正确的。

A.7.3 性能改进

自从计算机编程诞生以来，性能优化的重要性就已经得到了认可。事实上，Ada Lovelace 甚至在她 19 世纪的分析引擎笔记 [170] 中提到了性能优化。许多优化的初始实际部署发生在编译器开发中，通过优化编译器 [171]。这是当前实用

和高效计算的基础，但它必须是一种通用的方法；由于其通用性而广泛适用，但出于同样的原因，对于定制的问题领域不是最优的。因此，也有很多关于具体的源到源转换以提高优化的工作，可以追溯到 20 世纪 70 年代 [172], [173]。

长期以来，这项工作的重点是寻找合适的保意义变换集，其动机是可以将一个正确的程序转换为一个更高效的版本，同时保持其正确性。然而，最近，程序合成的研究发生了不同的转变：受遗传编程的启发 [174]，以及来自早期自动程序修复的研究 [146], [175]，它考虑了一种被称为“基因改良”的方法中的更广泛的转变 [8], [176]。

更广泛的转换集可能产生不正确的代码，但自动化测试可以过滤这些代码，以确保对预期语义的足够忠实。此外，将现有代码视为一种“遗传物质”的自由，在非功能属性方面产生了巨大的改进，例如执行时间、内存和功耗（例如，一个重要的基因测序系统的 70 倍加速 [177]）。

虽然进化算法等人工智能技术提高性能的潜力已经得到了充分的研究，但研究人员才刚刚开始考虑这种潜力用于基于 llm 的性能改进。在 Madaan 等人的工作中 [178]，作者使用 CODEGEN 和 CodeX 提出功能正确、性能提高的编辑（PIE），提高 Python 和 C++ 的执行时间（已经使用最大优化编译器选项-O3 进行了预优化）。类似地，Garg 等人 [179] 提出 DeepDev-PERF，一种针对 C# 应用程序的性能改进建议方法。DeepDev-PERF 使用英语预训练的 BART-large 模型，并进一步对源代码进行预训练。Kang 和 Yoo[180] 提出使用 llm 来建议用于遗传改进的目标特异性变异算子，并提供了提高效率和减少内存消耗的演示。Garg 等人 [181] 提出了 RAPGen，它为 llm 生成零样本提示以提高性能。提示信息是通过从预构建的性能知识库中检索指令生成的。Chen 等人 [182] 使用 GPT 模型作为他们的源代码优化方法超音速的基线，并发现超音速提高了运行时间 26.0％的项目，而 GPT-3.5-Turbo 仅提高 12.0％，GPT-4 则仅提高 4.0％。

Cummins 等人 [183] 重点研究了编译器的性能，介绍了用于优化编译器指令的 LLMs 的结果。他们的研究表明，一个相对较小的（7B 参数）LLM，经过训练以生成指令计数和优化编译器 LLVM 代码，可以在减少编译器指令计数方面带来 3％的改进，超越最先进的技术。他们的结果在正确性方面也很有希望，91％的代码可编译，70％的功能正确，与原始编译器输出相比有所提升。

在过去 50 年中，软件工程社区已经发展出如何将现有的软件系统转换为在保留功能行为的同时提高性能的等效系统的概念。在 20 世纪 70 年代，最关注的是正确性，因此转换集合被定义为仅由构造上（功能上）正确的转换步骤组成。

然而，到 2010 年，社区已经在探索更宽松的等价概念的应用，这些等价概念仅仅保留了对原始行为足够的操作忠实度。因此，20 世纪 70 年代严格的语义约束得到了相当大的放松，允许转换甚至可能使某些测试用例失败。在同一时期，业务性能变得越来越重要。该研究议程的一个关键基本原则是，当整个软件系统在一个低效率导致剩余资源不足的系统上运行时，没有一个整体软件系统可以被认为是功能正确的。这个原则甚至适用于软件已经完全被证明功能正确的情况（相对少见）。正如更精辟的口号所言：

“电池没电是不对的”[8]。

社区代码转换和合成方法的这种演变如图 5（红色和黄色区域）所示。

在语义约束日益放松的背景下，我们可以将基于 llm 的代码优化视为这一总体方向的进一步发展：由 llm 优化的代码甚至可能在语法上不正确，更不用说语义上正确了（如图 5 的绿色区域所示）。

尽管存在这些正确性挑战，但基于 llm 的软件工程固有的训练数据池很大，而且 llm 有表现出突发行为的倾向。这些观察结果结合起来产生了令人惊讶的结果，尽管不能保证是正确的，但可能会以有用的方式极大地改变性能特征。

当然，随着越来越多地允许更宽松的转换集合，以期优化多个非功能属性，同时更依赖于测试的能力，以提供功能的忠实性保证。测试对于检查那些非功能属性中的回归也至关重要，这些属性不是改进过程的目标。因此，一般的软件测试（特别是自动化的高覆盖率测试生成）将变得更加重要。

A.7.4 克隆检测和复用

以前有很多关于托管软件重用的工作 [184]，为了提取价值和避免重复，也使用 llm 解决了一个主题 [185]。软件通常包含大量克隆，这是由临时重用产生的，导致在自动克隆检测方面进行了大量工作 [186]。基于模糊的微调 llm 也已应用于此主题 [187]。

A.7.5 重构

重构代码时，我们通常希望其行为保持不变。这对于自动化方法特别有吸引力（例如基于搜索的重构 [188]），因为这意味着我们可以简单地依赖自动化回归 Oracle。这种“免费自动化 oracle”的优势非常重要，也适用于基于 llm 的重构。

Poldrack 等人 [75] 表明，对现有代码进行 GPT-4 重构可以显著提高代码质量结构指标，如 Halstead[189] 和 McCabe[190] 复杂性。Noever 和 Williams[83] 强调了 AI 驱动的代码助手在重构遗留代码和简化高价值存储库的解释或功能方面的

价值。

A.7.6 维护和演化中的开放性问题

由于许多软件维护和演化的子领域都与现有的遗留系统源代码有关，可以预见 LLMs 的应用将迅速增长。本节概述了这一新兴研究子领域中存在的一些开放问题。

A.7.6.1 性能改进中的开放问题

在开发基于 llm 的技术以自动发现性能改进方面还需要做更多的工作。与遗传改进一样，这些不需要仅仅局限于执行时间，还可以考虑其他非功能属性，如功耗 [191]–[193] 和内存占用 [194]，以及多目标非功能属性集之间的权衡 [195]。我们期待在遗传改进风格的基于 llm 的代码优化技术上开展更多工作，有可能取得许多重大进展和突破。

A.7.6.2 重构中的开放性问题

根据定义，重构不会改变语义，因此基于 llm 的重构可以依赖于自动回归 Oracle。因此，在基于 llm 的重构方面还没有更多的工作，这令人惊讶。在这一小节中，我们将概述可能的方向。

三十年来，设计模式在实际软件工程中发挥了关键作用 [196]。llm 可以帮助工程师重构现有代码以使用设计模式，同时提供对开发人员友好的解释和文档。

每当出现新技术时，重构也是必要的。例如，当 API 更新或新的 API 可用时。尽管它们可以（有时是自动的 [197]）修复，但 API 滥用仍然是软件工程 bug 的常见来源。由于自动化回归 Oracle 的存在，自动化新 API 的重构过程比其他代码转换更具挑战性。

最后，llm 的少样本学习能力可以实现更多的定制重构。基于 llm 的重构研究主要集中在全局重构上。然而，程序员通常有特定于项目的重构需求。高达三分之一的软件工程工作花费在大量重复的、乏味的、有潜在错误倾向的重构活动上，以实现这些特定于项目的重构需求。llm 的少次学习潜力可能会自动从特定示例中泛化，自动化我们所谓的“定制”重构。需要做更多的工作来开发可靠的少样本学习的定制重构技术。

A.8 文档生成

基于 llm 的软件工程的大部分工作都集中在代码的生成上，但基于 llm 的文档生成也有相当大的潜力。

Sun 等人 [198] 探索了 ChatGPT 在 Python 代码摘要方面的表现。他们使用了 CSN-Python，并将 ChatGPT 与 NCS、CodeBERT 和 Code5 进行了比较。他们采用了三个广泛使用的指标：BLEU、METEOR 和 ROUGE-L。实验结果表明，ChatGPT 的 BLEU 和 ROUGE-L 性能明显低于基线模型。

Ahmed 等人 [66] 在 GPT-3.5 上进行了代码摘要的快速工程。耿等人 [199] 在两个 Java 语言数据集 Funcom 和 TLC 上进行了实验，使用 Codex 生成多重意图的评论。Gent 等人 [200] 证明了预训练的 llm 已经有足够的上下文来从不同的技术角度生成多个不同的代码摘要。

A.8.1 文档生成和代码摘要的开放性问题

许多现有的代码摘要技术是基于检索的：给定的代码使用神经表示以向量格式表示，随后用于检索代码语料库中最相关的文本摘要。

这种方法有一个明显的限制，因为可以生成的摘要集受到训练语料库的约束。LLMs 可以在其自然语言处理能力的辅助下，生成不限于训练语料库的自动代码摘要。

虽然这可能会导致更丰富、更语义相关的摘要，本文还注意到，现有的评估指标往往是词汇性的，妨碍了比较和评估 llm 生成的更丰富摘要的能力 [198]。基于 ReAct 方法的最新进展 [95] 可能为生成的文档提供更大的保证，即使它无法执行。

A.9 软件分析和存储库挖掘

软件分析是一个成熟的领域，研究如何从现有软件制品中为人类工程师产生洞察力 [201]。大量在线公开的软件制品信息刺激了通过挖掘软件库（MSR）[202], [203] 获得的科学见解的增长。虽然 MSR 往往侧重于从这种挖掘中获得科学研究见解，但软件分析往往侧重于组织从自己的存储库分析中获得见解的机会，这也可以有利于 AI 的可理解性 [204]。

迄今为止，在这两种情况下，大部分的数据收集、策展和分析依赖于劳动密集型的人工分析。我们没有发现使用 llm 来支持此活动的研究。然而，由于许多

llm 已经采集了这种软件制品数据，并能够提供推理和洞察，因此似乎很自然地期望它们发挥重要作用。

例如，llm 可以根据其摄取大量数据的能力，识别出有趣的新的 MSR 研究问题，包括研究问题和假设之前被证明是研究人员感兴趣的内容。它们也可以帮助追溯软件工程师很难维护的代码 [205], [206]。

A.10 人机交互

在人类工程师和软件基础设施之间寻找有效的接口一直是软件工程开发的整个生命周期中反复出现的主题 [207], [208]，可以追溯到 20 世纪 60 年代该学科的开始 [209]。

我们发现了许多有趣的研究问题的证据。例如，Vaithilingam 等人 [210] 报告了 24 名参与者在理解、编辑和调试副驾驶生成的代码时遇到的困难。同时，Feldt 等人 [139] 提出了基于 LLM 代理的软件测试设计架构。Liang 等人 [36] 调查了 410 名实习软件工程师，发现 llm 被广泛使用以促进低级编程任务，但也有人抵制将 llm 用于更高层的软件设计活动。Feng 等人 [211] 收集了关于 ChatGPT 代码生成的 316K 条推文和 3.2K 条 Reddit 帖子，以了解社交媒体对人工智能辅助编码工具的态度。

他们发现，与 ChatGPT 代码生成相关的主要情绪是恐惧，这种情绪掩盖了快乐和惊喜等其他情绪。Ahmed 等人 [212] 探索了软件架构新手与 ChatGPT 交互的方式。

A.11 软件工程过程

软件工程关注的不仅是软件产品，还有构建软件产品的过程 [213]。之前关于软件助手 [207], [214]–[217] 的研究显然与基于 llm 的软件工程研究特别相关，这一主题已经引起一些作者的关注。例如，Ross 等人 [218] 介绍了一个基于 llm 的程序员助理，并使用 42 个实例评估了它的部署情况。Tian 等人 [219] 强调了 ChatGPT 的注意力广度局限性。

A.12 软件工程教育

教师们对确定学生是否依赖 llm 来构建他们的任务表示担忧 [220]，而其他作者则认为 llm 对教育的长期影响将是有益的 [221]。然而，我们目前的重点更局限于 llm 对软件工程教育领域的具体影响，其中目前的文献侧重于基于 llm 的教程支持。

例如，Jalil 等人 [222] 探索了 ChatGPT 在软件测试教育中的机会和问题。Savelka 等人 [223] 分析了在高等教育水平的入门和中级程序设计课程中，LLMs 回答多项选择题的三种模式。其他几个作者 [82], [83], [224] 探索了 CodeX 生成编程练习和代码解释的能力。他们的普遍发现是，大多数生成的内容是新颖的、明智的且有用的（参见 IV-D3 部分）。

A.13 横切开放研究主题

许多模式从基于 llm 的软件工程的萌芽文献中涌现出来。在本节中，我们概述了那些跨越所有软件工程应用的开放研究问题。

A.13.1 为 SE 构建和调优 llm

以前的大多数工作都将 llm 视为原子组件，重点是将它们纳入更广泛的软件工程工作流中。虽然曾有过调整行为的尝试，但这些尝试往往侧重于快速工程，并有一些微调的例子。

一个更具挑战性但具有潜在影响的问题在于训练和微调模型，特别是针对软件工程任务。Ding 等人 [225] 用执行输入和动态执行轨迹训练一个类似 BERT 的 LLM。他们展示了这种动态信息如何提高模型对下游软件工程预测任务的准确性（高达 25%）：漏洞和克隆检测以及覆盖预测（完整执行路径和分支覆盖）。

需要在新形式的 llm 上做更多的工作，特别是针对软件工程量身定做的，利用软件的独特属性并将其与自然语言区分开来。动态信息是目前大多数工作中缺失的关键区别之一。我们期待下一代特定于 SE 的 llm 来解决这个问题。

建立和培训 llm 的一个重要方面是它们的能源消耗。LLM 的功能已与它们的规模相关 [226]，导致模型尺寸的快速增长 [227], [228]。更大模型的训练和开发可能会对环境产生直接影响 [229]。虽然有人认为，模型性能不仅取决于模型大小，还取决于训练数据量 [230]，但达到预期性能所需的正确模型大小问题仍然不清楚。

更轻的模型还可能扩大采用范围，从而提高可部署性。最近，诸如低秩自适应（LoRA）[231] 和模型量化 [232] 等技术已显示出潜力，但仍需就特定应用进行经验评估。

A.13.2 需要动态自适应提示工程和参数调整

关于 prompt 工程的初步工作已经证明了它在显著改进 LLMs 生成的软件工程制品方面的潜力。然而，正如已经发现的 [58]，结果是高度具体的问题，所以一刀切的方法是不现实的。此外，很少有论文报告模型参数设置，但我们知道其中许多，如温度设置，在确定生成的 LLM 输出的性质方面发挥着至关重要的作用。

作为一个直接的起点，作者必须显著地报告这些参数设置以支持复制。然而，动态自适应提示工程和模型参数调优还需要更多的研究。该研究议程可能会从其他动态自适应任务的参数调优方面的现有工作中获得灵感，例如模糊测试 [233]。动态提示优化也可以利用与 SBSE[12] 相关的技术，将提示优化重新制定为一个多目标计算搜索过程。

A.13.3 杂化

llm 在单独使用时很少是最有效的，但作为整个 SE 过程的一部分可以非常有效。需要更多的工作来理解 llm 可以安全、高效和有效地驻留的 SE 工作流的设计模式。我们认为，现有的与生成-测试方法相关的软件工程理论和实践，如自动修复和遗传改进，已经高度适合 llm。

我们希望看到更多的工作将 llm 纳入这些现有的软件工程框架。然而，需要更多的工作来定制和扩展这些框架，以最好地利用基于 llm 的软件工程提供的机会。

特别是，我们期望看到静态和动态分析工作的快速发展，以便快速进行 LLM 响应的工程和后处理。我们还希望看到混合软件工程过程，适应持续集成管道以合并 llm。

A.13.4 控制幻觉

虽然幻觉被广泛认为是一个问题，正如在本调查中报告的那样，它也可能在应用于软件工程领域时提供好处。LLM 的幻觉很少是完全随机的错误反应。相反，由于它们固有的统计特性，它们应该被更好地描述为“合理的未来”。在正确的背景下，这可能经常使它们有用。

幻觉可以被重新利用，为软件增强提供潜在的有用建议。例如，当产生测试用例的幻觉时，LLM 可能会被重新用于建议新特征，而幻觉的代码摘要可能表明潜在的（人类）代码误解；如果 LLM“误解”了代码，人类难道不会也误解它吗？当 LLM 对一个不存在的 API 产生幻觉时，它可能会被重新用作一种建议重构以简化或扩展现有 API 的方法。我们还需要做更多的工作来开发这种积极的潜力，并利用幻觉来改进软件。

A.13.5 稳健、可靠、稳定的评估

Hort 等人 [234] 对 293 篇关于代码生成的 llm 论文进行了审查，以确定是否提供了充分的信息支持复制。他们发现，只有 33％的人共享源代码，27％的人共享经过训练的人工制品。他们还从能量消耗的角度评估了这些论文，分析了独立研究人员在训练过程中评估能量消耗的可能性。他们报告说，大约 38％（79 篇涉及模型训练的出版物中的 30 篇）共享了足够的信息来估计训练期间的能量消耗。

进一步的证据表明，在基于 llm 的软件工程文献中，可能存在一个日益增长的科学评估质量问题，例如 Wang 等人的基于 llm 的测试调查 [98]。他们的调查过滤了初始论文池，以删除那些不满足标准评估质量约束的论文。这些限制要求论文包含清晰、可重复的评估方法，并提供衡量有效性的控制或基线。这个过滤标准删除了 90％以上最初符合关键词搜索标准的论文。

这些对文献的分析表明，显然，需要更多的工作来为基于 llm 的软件工程这一新兴学科建立坚实的科学基础。这种基础可以借鉴一般经验软件工程的现有基础，更具体地说，基于人工智能的软件工程，如 SBSE（其中有自然的相似性 [105], [235]）。

然而，llm 有自己独特的属性，例如提供解释的能力，这将需要特定领域的理论和经验科学基础。

A.13.6 全面测试

幻觉的问题已经被广泛研究。在软件工程社区和更广泛的计算机科学社区中，它将继续是一个非常有趣的话题。虽然可能会取得巨大的进展，但幻觉的固有风险不太可能完全消除，因为它与 LLM 技术的特性相关，就像人类智能一样。幸运的是，60 多年来，软件工程师已经开发了健壮的自动化验证和测试技术，有助于减少人为错误的影响。我们预计这些技术也将延续到人工智能的错误中。

A.13.7 处理较长的文本输入

llm 在大型输入提示上的性能可能是人工智能社区非常感兴趣的主题 [236]。这一领域的进步将对软件工程产生重大影响，因为软件系统的规模相当大，而在处理较大的提示时，可能会带来新的机会和挑战。

A.13.8 软件工程覆盖较少的子领域

正如我们的调查显示，软件工程的一些子领域在文献中明显代表性不足；有些人出乎意料地如此。例如，需求工程与设计（III），以及重构（VI-E 节）的覆盖面较少。然而，考虑到重构严重依赖于分析的语言形式以及模式的识别和预测，其研究时机已经成熟。

参考文献

书面翻译对应的原文索引

[1] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*, 2023. URL https://arxiv.org/abs/2310.03533. Accessed: 2025-01-08.

附录 B　补充内容

附录是与论文内容密切相关、但编入正文又影响整篇论文编排的条理和逻辑性的资料，例如某些重要的数据表格、计算程序、统计表等，是论文主体的补充内容，可根据需要设置。

附录中的图、表、数学表达式、参考文献等另行编序号，与正文分开，一律用阿拉伯数字编码，但在数码前冠以附录的序号，例如“图 B.1”，“表 B.1”，“式（B.1）”等。

B.1　插图

图 B.1　附录中的图片示例

B.2　表格

B.3　数学表达式

$$\frac{1}{2\pi \mathrm{i}}\int_{\gamma} f = \sum_{k=1}^{m} n(\gamma; a_k)\mathscr{R}(f; a_k) \tag{B.1}$$

表 B.1　附录中的表格示例

文件名	描述
thuthesis.dtx	模板的源文件，包括文档和注释
thuthesis.cls	模板文件
thuthesis-*.bst	BibTeX 参考文献表样式文件
thuthesis-*.bbx	BibLaTeX 参考文献表样式文件
thuthesis-*.cbx	BibLaTeX 引用样式文件

B.4　文献引用

附录[?]中的参考文献引用[?]示例[??]。

参考文献

致　　谢

衷心感谢导师 ××× 教授和物理系 ×× 副教授对本人的精心指导。他们的言传身教将使我终生受益。

在美国麻省理工学院化学系进行九个月的合作研究期间，承蒙 Robert Field 教授热心指导与帮助，不胜感激。

感谢 ××××× 实验室主任 ××× 教授，以及实验室全体老师和同窗们学的热情帮助和支持！

本课题承蒙国家自然科学基金资助，特此致谢。

声　　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

在学期间参加课题的研究成果

个人简历

197× 年 ×× 月 ×× 日出生于四川 ×× 县。

1992 年 9 月考入 ×× 大学化学系 ×× 化学专业，1996 年 7 月本科毕业并获得理学学士学位。

1996 年 9 月免试进入清华大学化学系攻读 ×× 化学博士至今。

在学期间完成的相关学术成果

学术论文：

[1] Yang Y, Ren T L, Zhang L T, et al. Miniature microphone with silicon-based ferroelectric thin films[J]. Integrated Ferroelectrics, 2003, 52:229-235.

[2] 杨轶, 张宁欣, 任天令, 等. 硅基铁电微声学器件中薄膜残余应力的研究 [J]. 中国机械工程, 2005, 16(14):1289-1291.

[3] 杨轶, 张宁欣, 任天令, 等. 集成铁电器件中的关键工艺研究 [J]. 仪器仪表学报, 2003, 24(S4):192-193.

[4] Yang Y, Ren T L, Zhu Y P, et al. PMUTs for handwriting recognition. In press[J]. (已被 Integrated Ferroelectrics 录用)

专利：

[5] 任天令, 杨轶, 朱一平, 等. 硅基铁电微声学传感器畴极化区域控制和电极连接的方法: 中国, CN1602118A[P]. 2005-03-30.

[6] Ren T L, Yang Y, Zhu Y P, et al. Piezoelectric micro acoustic sensor based on ferroelectric materials: USA, No.11/215, 102[P]. (美国发明专利申请号.)