



清华大学

综合论文训练

bolt.SE：基于大语言模型的软件开发
辅助工具

系别：软件学院

专业：软件工程

姓名：陈敬文

指导教师：刘璘 研究员

二〇二五年五月

关于论文使用授权的说明

本人完全了解清华大学有关保留、使用综合论文训练论文的规定，即：学校有权保留论文的复印件，允许论文被查阅和借阅；学校可以公布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论文。

作者签名：

导师签名：

日 期：

日 期：

摘要

本研究设计实现了一个交互式软件开发工具 bolt.SE，旨在将大语言模型（LLM）的代码生成能力与现代软件工程最佳实践深度融合。该系统基于开源项目 bolt.diy 构建，通过四个核心模块增强了 LLM 辅助开发的工程化程度：首先，API 优先开发模块基于 OpenAPI 规范实现 API 定义与管理，使 LLM 能够理解并生成符合接口规范的代码；其次，测试驱动开发（TDD）模块将“测试先行”理念与代码生成相结合，通过结构化测试约束引导 LLM 生成高质量代码；再次，模型上下文协议（MCP）模块提供标准化接口使 LLM 能够安全调用外部工具和数据源，突破知识局限；最后，持续集成与部署模块则结合了 TDD 模块与 MCP 模块，实现了从测试定义到自动部署的全链路自动化。bolt.SE 有效解决了当前 LLM 辅助开发面临的上下文理解受限、反馈机制不完善及工程化流程缺失等挑战，为构建更加高效、易用的软件开发支撑环境提供了新思路。

关键词：大语言模型；软件工程；测试驱动开发；API 优先开发；模型上下文协议

Abstract

This study introduces an interactive software development tool, bolt.SE, designed to deeply integrate the code generation capabilities of large language models (LLMs) with modern software engineering best practices. Built upon the open-source project bolt.diy, the system enhances the engineering rigor of LLM-assisted development through four core functional modules: First, the API-first development module leverages the OpenAPI specification to define and manage APIs, enabling LLMs to generate code compliant with interface standards; Then, the test-driven development (TDD) module incorporates the "test-first" approach, guiding LLMs to produce high-quality code through structured testing constraints; Thirdly, the Model Context Protocol (MCP) module provides standardized interfaces that allow LLMs to safely interact with external tools and data sources, thereby overcoming knowledge limitations; Finally, the Continuous Integration and Deployment module integrates the TDD and MCP modules to achieve end-to-end automation from test definition to deployment. Our experiments show that bolt.SE effectively addresses key challenges faced by current LLM-assisted development, such as limited context understanding, insufficient feedback mechanisms, and lack of engineering workflows, offering new ideas for building more efficient and user-friendly development environments.

Keywords: large language models; software engineering; test-driven development; API-first development; model context protocol

目 录

第 1 章 引言与研究背景.....	1
1.1 研究背景与意义	1
1.2 当前 LLM 辅助开发工具发展与挑战	1
1.2.1 上下文理解局限	3
1.2.2 交互反馈与即时验证不足	3
1.2.3 软件工程实践融合不足	3
1.3 bolt.SE 的目标.....	3
1.4 论文结构	4
第 2 章 bolt.SE 系统架构与技术创新.....	6
2.1 bolt.diy 介绍	6
2.1.1 系统架构	6
2.1.2 用户界面	8
2.1.3 MessageParser	8
2.1.4 WebContainer 技术	10
2.2 bolt.SE 的技术创新与改进.....	10
2.2.1 API 优先开发模块	10
2.2.2 测试驱动开发（TDD）模块.....	11
2.2.3 模型上下文协议（MCP）模块.....	11
第 3 章 API 优先开发的设计与实现.....	12
3.1 API 优先开发对于 bolt.SE 的意义	12
3.2 bolt.SE 中的 OpenAPI 应用	12
3.3 bolt.SE 中的 APIActions 实现	14
3.4 实例应用场景	17
第 4 章 测试驱动开发的设计与实现.....	21
4.1 测试驱动开发对 bolt.SE 的意义	21
4.2 bolt.SE 中的 Jest 测试框架应用	21
4.3 bolt.SE 中的测试功能实现.....	22
4.4 实例应用场景	25
第 5 章 模型上下文协议（MCP）的设计与实现.....	29
5.1 模型上下文协议对 bolt.SE 的意义	29

5.2 bolt.SE 中的 MCP 功能实现.....	30
5.3 实例应用场景	35
第 6 章 持续集成与持续部署（CI/CD）的设计与实现	41
6.1 环境准备与整体流程	41
6.2 配置 GitLab MCP 服务器	42
6.3 导入 Jest 测试并生成应用代码	42
6.4 自动生成 GitLab CI 配置.....	42
6.5 仓库初始化与首轮流水线触发	45
第 7 章 结论与展望.....	47
7.1 主要研究成果	47
7.2 与 bolt.diy 开源项目的连接	48
7.3 未来发展方向	48
7.4 总结	49
参考文献.....	50
附录 A 外文资料的书面翻译	52
致 谢.....	80
声 明.....	81

插图清单

图 2.1 bolt.SE 系统架构图：显示 bolt.diy 基础流程（灰色区域）与 bolt.SE 创新模块 （蓝色区域）的集成与交互	7
图 2.2 bolt.diy 用户界面示例	8
图 3.1 API 使用工作流程图：展示用户创建、编辑和使用 API 的完整流程，以及 API 信息在系统各组件间的传递方式	15
图 3.2 API 数据模型类图：描述系统中 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	16
图 3.3 添加 NWS Weather API：粘贴 OpenAPI 规范定义并配置端点与认证方式 ..	17
图 3.4 APIActions 总览界面：展示所有注册的 API、认证方式与可用操作数	18
图 3.5 对话框中勾选相关 API，配合自然语言指令触发任务生成	18
图 3.6 系统生成项目结构与依赖安装命令，包含前端组件与服务调用逻辑	19
图 3.7 用户请求“show me a cute dog”，系统调用 The Dog API 返回图片	19
图 3.8 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 ..	20
图 4.1 测试功能流程图：展示用户测试定义、使用及验证的完整流程与跨组件信息 传递机制	23
图 4.2 测试数据模型类图：描述测试定义的核心数据结构及关系，包含测试代码 项、测试用例与测试套件的层次组织	24
图 4.3 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，并以树形结构展示	25
图 4.4 开发者在聊天窗口输入 Build a calculator 的需求，系统自动注入 测试文件与指导语，为 LLM 提供明确功能规格	26
图 4.5 LLM 基于测试用例生成实现代码，运行测试后显示 12 条断言全部通过 ..	26
图 4.6 修改测试断言后执行测试，出现一条失败记录，系统进入红灯阶段	27
图 4.7 系统将测试失败信息传递给 LLM，模型识别出 add 函数需针对 (2,3) 输入 做特殊处理，并生成相应补丁代码	27
图 4.8 补丁应用后测试套件重新通过	28
图 4.9 Preview 面板提供计算器实时预览。开发者可进行代码重构与优化，测试套 件确保功能完整性不受影响	28
图 5.1 MCP 工作流程图：展示用户配置、服务器连接和工具调用的完整流程 ...	31

图 5.2 MCP 状态图：描述 MCP 连接的各状态及转换关系，从初始配置到工具调用 的完整流程.....	32
图 5.3 MCP 数据模型类图：展示系统中 MCP 相关模块的类结构及其关系，包括配 置管理、服务创建和连接检查.....	33
图 5.4 remote-sse 服务器配置与 IoTDB 工具发现.....	37
图 5.5 LLM 工具调用与响应	39
图 5.6 生成应用的 Workbench 预览，展示电池数据监测折线图与数据刷新功能 .	40
图 6.1 开发者输入指令.....	42
图 6.2 MCP 服务器 server-gitlab 配置与连通性检测	43
图 6.3 导入 Jest 测试文件并解析测试树.....	43
图 6.4 全部测试通过后的运行预览.....	44
图 6.5 生成 CI/CD 计划卡片及 .gitlab-ci.yml 代码差异.....	44
图 6.6 MCP 工具 create_repository 的调用参数与返回结果.....	45
图 6.7 远端仓库初始化结果（GitLab UI）	46

附表清单

表 1.1 典型 LLM 辅助开发工具功能对比.....	2
------------------------------	---

符号和缩略语说明

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）
MCP	模型上下文协议（Model Context Protocol）
SSE	服务器发送事件（Server-Sent Events）
HTTP	超文本传输协议（Hypertext Transfer Protocol）
REST	表述性状态转移（Representational State Transfer）
OpenAPI	开放 API 规范（OpenAPI Specification）
Jest	JavaScript 测试框架
IoTDB	物联网数据库（Internet of Things Database）

第 1 章 引言与研究背景

1.1 研究背景与意义

近年来，大语言模型（Large Language Models, LLMs）在代码生成领域展现出显著进步，成为自然语言与可执行代码之间桥梁能力提升的重要驱动力。相关系统性综述表明，LLM 在 HumanEval、MBPP 等主流基准任务中的表现持续提升，反映出模型在理解与生成能力上的重大突破^[1]。同时，学界也在不断完善对代码生成准确性、效率及可读性的评测方法，推动评测体系向更高标准发展^[2]。在工业界，AI 编程辅助工具的应用快速普及。GitHub 的调研数据显示，92% 的美国开发者已在实际开发中采用此类工具^[3]；SWE-bench 评测中 LLM 的问题解决率已从 2023 年的 4.4% 显著提升至 69.1%，进一步印证了 AI 驱动开发效率的提升^[4]。

尽管 LLM 相关研究和应用持续取得突破，如何将其能力真正融入大型软件工程实践，仍面临诸多挑战，包括上下文理解受限、反馈机制不完善以及工程化流程缺失等。因此，将 LLM 的生成能力与软件工程最佳实践紧密结合，如测试驱动开发（Test-Driven Development, TDD）、API 优先（API-First）开发和持续集成/持续部署（Continuous Integration/Continuous Deployment, CI/CD），已成为当前亟待解决的重要课题。

1.2 当前 LLM 辅助开发工具发展与挑战

为系统梳理主流 LLM 编程工具在实际开发环境中的差异，表 1.1 汇总了几款代表性产品在集成平台、主要功能及反馈机制方面的对比。当前具有影响力的典型工具包括 GitHub Copilot X、Cursor、v0.dev、Replit、Lovable、Windsurf 和 bolt.new 等。

表 1.1 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	反馈机制
GitHub Copilot ^[5]	VS Code / JetBrains 扩展	多模型 (GPT-4o、Claude 3.5、Gemini 1.5 Pro) 行内补全、代码审查支持	无内置 CI 流水线反馈
Cursor ^[6]	独立 AI IDE (VS Code 支分)	对话式多文件生成、跨文件重构、Diff 视图	实时代码差异预览；无原生 TDD / CI/CD 集成
v0.dev ^[7]	浏览器内 Chat UI	基于 React/Tailwind 的 UI 自动生成	页面预览与即时部署；缺少测试与流水线支持
Replit ^[8]	云端 IDE	Ghostwriter AI、Agent V2 对话式生成、协同编辑	实时代码运行、部署管道；无结构化测试反馈
Lovable ^[9]	Web App Builder	对话式前后端+数据库一体化生成	即时预览与调试；无原生 CI/CD 流水线
Windsurf ^[10]	AI 增强 IDE	行内补全、多文件编辑、聊天问答	变更高亮与上下文聊天；尚无 TDD / CI 插件
bolt.new ^[11-12]	浏览器 IDE (StackBlitz WebContainers)	从提示到部署：项目依赖管理、Node.js 运行、前端一体化生成	终端输出、即时预览与部署状态反馈；支持文件系统与第三方 API 调用

GitHub Copilot X 自 2023 年起，在多模型切换和多文件补全方面取得了显著进展，集成了 GPT-4o、Claude 3.5 和 Gemini 1.5 Pro，支持基于项目全局上下文的代码建议生成，并新增了代码审查和安全提示等能力，但与 CI/CD 或 TDD 流程的深度集成仍有待加强^[5]。Cursor 通过“vibe coding”工作流，将对话式多文件生成与实时 Diff 视图结合，提升了多文件协同开发效率。然而，其自动化测试和持续交付方案尚不完整，部分流程仍需借助外部工具补充^[6]。v0.dev 主打对话生成 React/Tailwind 前端代码和一键部署，适合原型设计及 UI 快速验证，但缺乏对测试的原生支持^[7]。Replit 借助 Ghostwriter AI 和 Agent V2，实现了云端协作开发与即时运行体验，但结构化测试和 CI/CD 功能仍主要依赖外部插件^[8]。Lovable 能在单轮对话中生成全栈前后端与数据库代码，支持即时预览，但缺乏完善的版本控制与自动化流水线，难以满足团队长期协作需求^[9]。Windsurf 将行内补全与上下文对话结合，提升代码建议的语义丰富性，但暂未实现与 TDD 或 CI/CD 的深度耦合^[10]。bolt.new 依托 StackBlitz WebContainers，提供浏览器内完整的依赖安装、Node.js 服务与代码生成能力，终端输出和部署状态反馈为快速验证提供便利，但在测试驱动和持续交付方面仍有扩展空间^[11-12]。

可以看到，尽管上述工具在各自领域取得了一定进展，但在应用于实际软件工程实践中仍存在几项关键挑战：

1.2.1 上下文理解局限

在处理大型与复杂软件代码库时，当前工具面临对代码上下文理解不足的问题。GitHub Copilot 等早期工具主要基于当前打开的文件内容进行补全，对项目全局信息缺乏深入理解。即使是 Cursor 和 Windsurf 等支持多文件编辑的工具，在理解代码模块间接口关系、命名规范以及整体架构设计约定等方面仍有不足，导致生成的代码难以与既有环境高效融合。

1.2.2 交互反馈与即时验证不足

现有工具普遍缺乏完整、成熟的网页端实时代码运行及预览能力，使开发人员难以即时观察和验证生成代码的实际表现。虽然 bolt.new 通过 WebContainers 提供了浏览器内运行环境，v0.dev 和 Lovable 支持 UI 代码即时预览，但它们在复杂项目的测试验证方面仍不完善。多数平台尚未实现生成代码的即时测试反馈，依赖与配置问题难以及时被发现和解决，开发流程中从代码生成到测试验证的环节也未能做到无缝衔接，用户体验仍然割裂。

1.2.3 软件工程实践融合不足

当前 LLM 辅助工具尚未有效整合现代软件工程的最佳实践。尽管 Replit 提供了部分部署管道，但对于测试驱动开发、API 优先设计以及持续集成与部署等核心流程的支持仍显不足。这种融合上的欠缺，导致工具在多个方面存在局限。首先，缺乏基于测试定义指导代码生成的系统化方法，使开发过程难以保证质量可控。其次，API 开发过程中规范定义与实现往往相互割裂，难以确保一致性和可维护性。最后，从代码生成到部署的全链路自动化程度较低，仍需开发者投入大量人工操作，难以实现高效闭环。这些问题共同凸显了一个系统化软件工程环境的必要性，即不仅要充分发挥 LLM 在代码生成方面的优势，还应将其与软件工程最佳实践紧密结合，形成覆盖需求定义、代码验证到系统部署的完整开发流程。

1.3 bolt.SE 的目标

基于以上分析，本研究提出了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，体现该平台的技术基础；“SE”则表示软件工程（Software Engineering），突出该环境对软件工程方法论的深度融合与系统化支持。

bolt.SE 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，实现更加高效且易用的 LLM 驱动软件原型开发工

具。具体目标包括：

- 系统引入 API 优先开发模块，增强 LLM 与外部服务的交互能力。通过基于 OpenAPI 规范的 API 定义与管理，让 LLM 理解并生成符合 API 规范的代码，实现外部服务与 LLM 的无缝对接，有效扩展模型功能边界。
- 通过结构化的测试驱动开发模块，改善代码生成质量与可靠性。该模块让用户能够在生成代码前先定义期望行为，将测试约束转化为 LLM 可理解的指导信息，建立基于“红-绿-重构”循环的验证机制，确保生成代码符合预定义行为。
- 通过模型上下文协议（MCP）模块，突破 LLM 知识与功能边界。此模块提供标准化接口，支持 LLM 安全调用外部工具和数据源，实现访问远程服务调用的多场景支持。
- 通过 MCP 与 TDD 协同实现 CI/CD 自动化流程，完成从测试定义到部署的全链路自动化。系统利用标准化工具接口自动完成仓库创建、配置生成、代码推送与流水线触发等操作，形成完整开发闭环，简化传统手动操作环节。

这些目标共同构成一个端到端软件开发环境，使开发者能够专注于创意表达与业务逻辑，而非繁琐的编程细节，提高软件原型设计与开发的可达性与效率。

本研究将基于 bolt.diy 开源框架构建 bolt.SE 平台，实现对 API 优先开发、测试驱动开发与模型上下文协议等核心技术模块的集成，以达到前述目标。

1.4 论文结构

论文结构安排如下：

第 1 章（本章）介绍研究背景、问题提出与目标，分析当前 LLM 辅助开发工具的发展与挑战，概述 bolt.SE 的核心设计。

第 2 章详细介绍 bolt.SE 系统的整体架构、核心技术及创新点。该章首先介绍 bolt.diy 的基础架构与技术特点，包括开源与扩展性、多种大语言模型支持、代码生成能力以及 WebContainer 技术等方面。随后重点阐述 bolt.SE 的创新模块及其在软件工程流程中的应用价值，包括 API 优先开发模块、测试驱动开发模块和模型上下文协议模块。

第 3 章聚焦 API 优先开发模块，介绍 OpenAPI 规范在 bolt.SE 中的应用，API 定义与管理机制，以及 LLM 如何理解并调用 API 实现功能扩展。该章将分析 bolt.SE 中 APIActions 的实现方案，展示结构化 API 定义如何帮助 LLM 突破知识局限，通过与外部服务交互扩展其能力边界。

第 4 章详细阐述测试驱动开发模块的概念与技术实现，分析 TDD 在 LLM 驱动

开发中的价值，并通过具体案例展示测试定义如何引导 LLM 生成高质量代码。该章将介绍 bolt.SE 测试功能的技术实现，并通过 JavaScript 计算器的开发流程，展示基于“红-绿-重构”循环的测试驱动开发过程。

第5章深入剖析模型上下文在 bolt.SE 中的原理与实现，分析 MCP 如何通过标准化工具接口使 LLM 能够安全地访问外部资源，突破知识截止限制，降低输出幻觉风险，并支持实时数据获取与专业工具调用。通过 IoTDB 数据库集成案例，展示 MCP 在统一自然语言交互中同时发现并调用多类工具的强大能力。

第6章综合前述核心技术，展示 MCP 与 TDD 协同驱动的 GitLab CI/CD 自动化流程，体现 bolt.SE 在全链路软件工程实践中的系统集成能力。该章通过应用实例，详述从 Jest 测试定义到 GitLab 仓库创建、CI 配置自动生成及部署的完整开发闭环，实现从测试、代码、仓库、流水线到部署的全链路自动化。

第7章总结全文研究成果，阐述 bolt.SE 与 bolt.diy 开源项目的紧密联系和相互促进的创新循环，以及对社区的贡献。同时展望 bolt.SE 的应用潜力与未来计划。

第 2 章 bolt.SE 系统架构与技术创新

本章将详细介绍 bolt.SE 系统的整体架构、核心技术及创新点。bolt.SE 基于开源项目 bolt.diy 构建，保留其核心优势的同时，引入多项技术创新以增强软件工程实践支持。本章首先介绍 bolt.diy 的基础架构与技术特点，随后重点阐述 bolt.SE 的创新模块及其在软件工程流程中的应用价值。

2.1 bolt.diy 介绍

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

- 开源与扩展性：bolt.diy 作为开源平台，具备高度扩展性和灵活性，便于根据项目需求进行定制化开发，适应各种软件工程实践要求。
- 多种大语言模型支持：bolt.diy 支持多种 LLM，如 GPT-o3-mini、Claude 3.7 Sonnet 等，并持续扩展对新模型的支持。
- 代码生成能力：bolt.diy 能够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出色。
- WebContainer 技术：bolt.diy 使用 WebContainer 技术，使代码能在浏览器中实时执行并预览，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。

2.1.1 系统架构

bolt.diy 系统的工作流程涉及多个核心组件的协同运作，如图 2.1 灰色部分所示。整体流程包括以下几个关键步骤：

1. 用户通过 UI 界面输入需求，选择适当的 LLM。
2. LLM 生成自然语言响应，经过 MessageParser 模块解析为结构化 XML 格式。
3. 生成的代码通过 FileStore 存储，并通过 ActionRunner 在 WebContainer 环境中执行。
4. 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 提供隔离、安全的运行环境，允许代码在浏览器中直接执行，关于 WebContainer 的详细介绍将在后面章节展开。

另外，由于所有代码生成过程本质上都表现为用户与大语言模型之间的交互

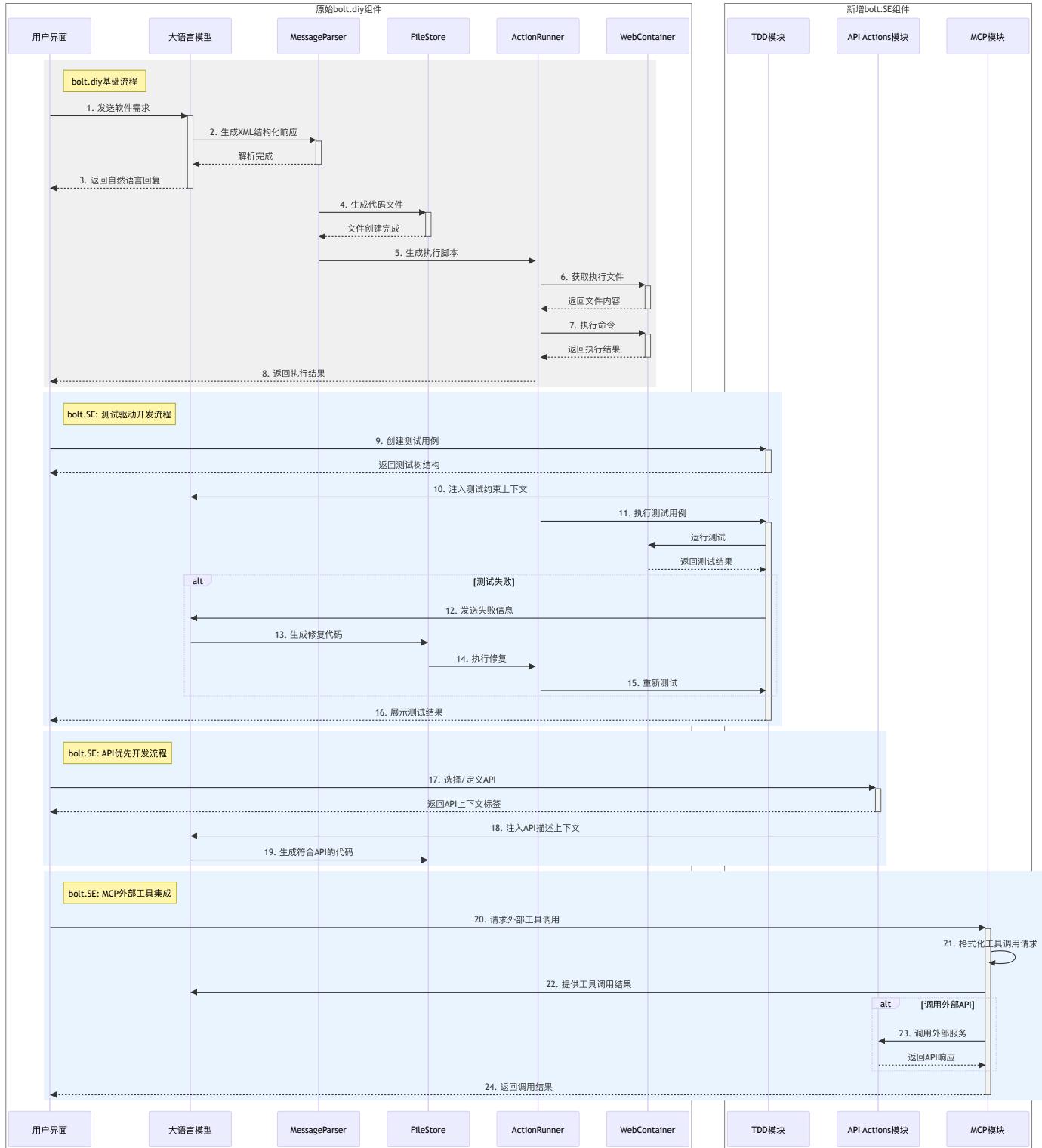


图 2.1 bolt.SE 系统架构图：显示 bolt.diy 基础流程（灰色区域）与 bolt.SE 创新模块（蓝色区域）的集成与交互

对话，因此，整个项目的代码库实际上是一系列高度结构化的“对话记录”。利用 LLM 的长文本处理能力，bolt.diy 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。这种设计使 LLM 不仅能基于当前用户输入生成代码，还能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。

2.1.2 用户界面

如图 2.2 所示，bolt.diy 提供直观易用的用户界面，用户在 UI 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），用户可根据需求选择适合的模型，以优化代码生成效果。根据用户输入的需求，bolt.diy 会调用选定的 LLM 生成代码，生成的代码在线 IDE 中展示。用户可在 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。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。

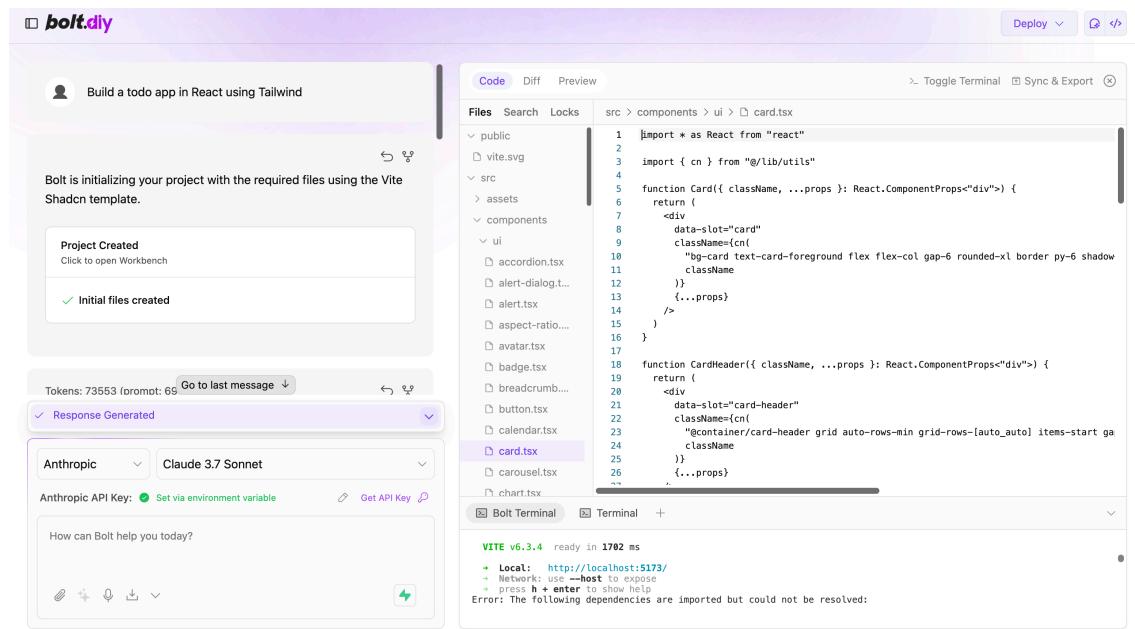


图 2.2 bolt.diy 用户界面示例

2.1.3 MessageParser

MessageParser 是 bolt.diy 的核心组件，负责解析 LLM 生成的 XML 格式输出。LLM 会生成结构化的 XML 响应，而 MessageParser 则识别和提取这些 XML 中的代码内容、文件路径、执行命令等元信息。以下示例展示了一个典型的 LLM 生成的 XML 结构，描述创建 JavaScript 阶乘函数的完整过程：

<example>

```

<user_query>
    Create a JS function to compute factorial.
</user_query>
<assistant_response>
    Certainly, Let's create a factorial function.
    <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>

```

在这个 XML 结构中，主要包含以下几个标签：

- <user_query>：包含用户的原始自然语言请求，本例中为创建计算阶乘的 JavaScript 函数。
- <assistant_response>：包含 LLM 的完整响应，包括文本回复和后续操作步骤，通常通过嵌套的 ‘boltArtifact‘ 和 ‘boltAction‘ 标签表示。
- <boltArtifact>：作为容器标签，表示一个完整的代码生成单元或”工件”，具有唯一标识符和描述性标题。
- <boltAction>：表示具体操作步骤，根据 type 属性的不同可执行不同类型的操作，例如：
 - type=”file”：创建或修改文件，其中 filePath 属性指定目标文件路径，标签内容为文件内容。
 - type=”shell”：执行命令行操作，标签内容为需执行的 shell 命令。

MessageParser 在运行过程中解析 LLM 输出的文本内容，识别其中的结构化信息。每个操作步骤被封装为 ‘boltAction‘，相关操作组合为 ‘boltArtifact‘，最终形成完整的代码生成工作流。系统随后根据解析结果，通过 FileStore 存储生成的代

码文件，并由 ActionRunner 执行相应的命令，实现从自然语言到可执行代码的转换过程。

2.1.4 WebContainer 技术

WebContainer 是 bolt.diy 的核心技术之一，由 StackBlitz 开发的基于 WebAssembly 的微型操作系统。它通过将 Node.js 编译为 WebAssembly，实现了在浏览器中原生运行 Node.js 环境的能力^[13]。在技术实现上，WebContainer 利用 WebAssembly 技术在浏览器中模拟完整的操作系统环境，包括文件系统和进程管理等功能。这使开发者能够在浏览器中执行各种命令，如 `npm install`、`node index.js` 等，实现完整的开发工作流程。在 bolt.diy 中，WebContainer 主要用于提供安全、隔离的代码运行环境。生成的代码通过 FileStore 存储在 WebContainer 的虚拟文件系统中，并由 ActionRunner 在该环境中执行。执行结果随后返回给用户，形成完整的代码生成-执行-反馈循环。

2.2 bolt.SE 的技术创新与改进

基于 bolt.diy 的原有架构，bolt.SE 引入了多项技术改进，重点强化了系统在现代软件工程实践中的适应性。如图 2.1 蓝色部分所示，bolt.SE 基于此核心流程，增加了四个关键模块：测试驱动开发（TDD）模块负责测试用例管理与执行；API 优先开发模块处理 API 定义与集成；模型上下文协议（MCP）模块提供外部工具调用能力。

2.2.1 API 优先开发模块

API 优先（API-First）是现代软件开发中的重要方法，强调在实现具体功能前先设计和定义 API 接口。bolt.SE 在此方面的创新体现为构建了基于 OpenAPI 规范的 API 定义与管理系统。

该模块允许用户通过界面定义 API 规范，系统随后将这些规范转化为 LLM 可理解的结构化指导信息。基于这些信息，LLM 能够生成符合预定义 API 规范的代码实现，确保生成代码与系统整体架构的一致性。

在具体实现上，bolt.SE 开发了全新的 APIActions 组件，包括：

- `ApiActionsContextTags`: 显示聊天界面中选中的 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

这些组件与 bolt.diy 的文件存储和对话管理系统整合，扩展了原有框架的功能

边界。例如，APIActions 模块在用户选择 API 时，将 API 描述信息压缩，仅保留端点信息与参数定义的核心要素，相比原始描述节省 token 消耗，优化了 bolt.diy 的上下文管理机制。

2.2.2 测试驱动开发（TDD）模块

测试驱动开发是一种强调“测试先行”的软件开发方法。bolt.SE 通过构建 TDD 模块，将这一理念与 LLM 代码生成相结合，形成了一套完整的测试驱动型代码生成流程。

TDD 模块与 bolt.diy 的 WebContainer 集成，利用其 JavaScript 执行能力，实现了测试用例的实时运行与验证。当测试失败时，系统自动将错误信息回传给 LLM，引导其进行针对性修复，形成完整的“红-绿-重构”循环。

在 bolt.diy 的设计中，代码生成主要依赖自然语言描述，缺乏结构化的验证机制。bolt.SE 的 TDD 模块在此基础上引入了基于 Jest 的测试框架集成，实现了测试用例的创建、管理与执行，提升了生成代码的可靠性与质量。

2.2.3 模型上下文协议（MCP）模块

bolt.SE 引入了模型上下文协议（Model Context Protocol, MCP），实现 LLM 与外部工具和数据源的交互。该协议定义了一套标准化接口，使 LLM 能够安全地调用外部功能。

在技术实现上，MCP 模块采用模块化架构设计，支持多种传输方式（如 stdio、HTTP SSE）和服务类型。这种灵活性使系统能够适应从本地文件访问到远程服务调用的广泛场景，增强了 LLM 的功能边界。

bolt.diy 原有架构中，LLM 主要通过生成 XML 结构化指令调用系统内置功能，如文件创建和命令执行。bolt.SE 的 MCP 模块扩展了这一机制，实现了与外部工具和服务的标准化集成。MCP 为每个外部工具定义了 JSON Schema 描述，使 LLM 能够理解工具功能并正确构造调用参数。

第 3 章 API 优先开发的设计与实现

API 优先开发（API-First Development）是一种软件设计与开发方法论，其核心在于在实际开发前完成应用程序接口（Application Programming Interface, API）的设计与定义。明确的 API 规范使开发团队能在实际实现前达成共识，促进前端、后端和移动端的并行开发，减少集成阶段的冲突和重复工作。统一的 API 设计确保系统各服务接口的一致性，降低维护成本。以 API 为界面的系统设计也推动了模块化和解耦架构的形成，各服务只需关注自身 API 的实现和使用，无需了解其他组件的内部工作机制。

本章将介绍 API 优先开发在 bolt.SE 中的实现方案与实例应用场景。

3.1 API 优先开发对于 bolt.SE 的意义

在 LLM 驱动的软件开发中，API 优先方法提供了一种将 LLM 与外部系统交互的明确结构和规范。API 定义为 LLM 提供了调用外部服务的具体指南，减少模糊指令导致的错误。通过 API 集成，LLM 能够获取实时数据和使用专业工具，扩展其能力范围。API 定义中的认证和授权机制为 LLM 提供了受控的访问方式，确保其操作在安全合规的范围内执行。此外，API 优先方法支持 LLM 将复杂任务分解为多个 API 调用的组合，提高问题解决效率。

bolt.SE 将 API 优先理念融入平台设计，通过将 OpenAPI 规范集成到开发流程中，实现 LLM 与外部 API 的有效协作，开发者能通过自然语言交互获得 API 增强的开发支持。bolt.SE 的 APIActions 模块结合了 LLM 的灵活性与 API 的精确性和功能扩展性，有效提升了软件开发效率。

3.2 bolt.SE 中的 OpenAPI 应用

在 bolt.SE 的设计过程中，我们选择 OpenAPI 规范作为 API 描述标准，主要基于其广泛的行业接受度、丰富的工具生态系统以及对机器可读格式的良好支持。相比其他 API 描述格式，OpenAPI 提供了更完整的功能描述能力和更好的扩展性，这些特性使其特别适合与 LLM 集成，便于模型理解和处理 API 结构。

OpenAPI 规范（原 Swagger 规范）是 API 优先开发的主流规范之一，为 API 描述和交互提供了标准化、语言无关的格式^[14]。作为业界广泛采用的 API 定义标准，OpenAPI 使开发者能以声明式方式完整定义 API 的各要素。OpenAPI 规范采

用 JSON 或 YAML 格式，结构化描述 API 的端点（endpoints）、操作（operations）、参数、响应、认证方法等关键要素。这种规范不仅适合人类理解，也便于机器处理，成为前端、后端和各类工具间沟通的桥梁。在 bolt.SE 的开发环境中，使用了 OpenAPI 的 YAML 规范定义 RESTful API 调用，从而更高效地利用现有功能，避免重复开发，结构化 API 定义还能够避免在大模型上下文中包含冗长 API 使用说明，节省 token 空间。

OpenAPI 规范主要包含以下组成部分^[14]：

- 基本信息：API 的标题、描述、版本、联系人信息和许可证等元数据。
- 服务器信息：API 的基础 URL 和不同环境（开发、测试、生产）的服务器地址。
- 路径：API 的各端点及可用 HTTP 方法（GET、POST、PUT、DELETE 等），是规范核心部分。
- 组件：包含可重用部分：
 - *schemas*: API 使用的数据模型和对象结构。
 - *parameters*: 可重用的参数定义。
 - *responses*: 各种响应格式和状态码。
 - *securitySchemes*: API 支持的安全认证机制。
- 操作：每个 API 端点的可用操作，包括操作 ID 与摘要、描述与标签、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及所需安全机制。

以下是 bolt.SE 中支持的 OpenAPI 定义示例，展示一个简单的天气 API：

```
openapi: 3.0.0
info:
  title: Weather API
  version: 1.0.0
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 模块解析这类 OpenAPI 定义，提取 API 端点和操作，使 LLM 能理解并正确调用这些 API。

3.3 bolt.SE 中的 APIActions 实现

bolt.SE 通过 APIActions 模块，使用户能够定义、管理和应用外部 API，同时使 LLM 能理解并调用这些 API，实现功能扩展和技术集成。图3.1展示了完整的 API 使用工作流程，包括用户创建、编辑和使用 API 的流程，以及 API 信息在系统各组件间的传递方式。

APIActions 是 bolt.SE 中实现 API 优先开发的核心功能模块，为 LLM 提供识别、理解和调用外部 API 的能力，同时优化 token 使用效率。该模块基于结构化数据模型设计，通过系统化的类结构定义 API 信息，实现从规范解析到功能调用的完整流程。图3.2展示了 APIActions 的数据模型类图，描述了系统如何组织和管理 API 定义。

APIActions 的数据结构包含 API 的所有必要信息，包括：

- 基本信息：唯一标识符（id）、名称（name）和基础 URL（serverUrl）。
- API 操作集：由多个 Action 对象组成，每个 Action 定义一个 API 端点的 HTTP 方法、路径和功能摘要。

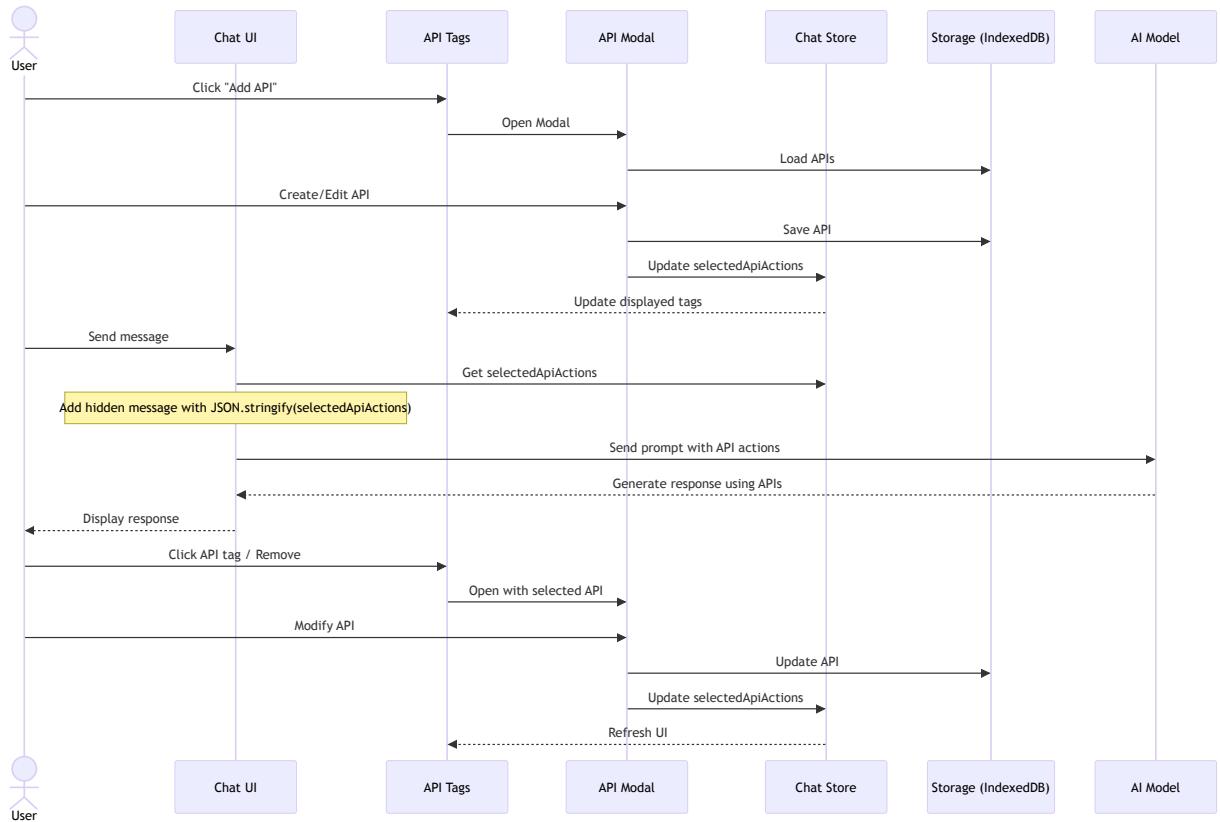


图 3.1 API 使用工作流程图：展示用户创建、编辑和使用 API 的完整流程，以及 API 信息在系统各组件间的传递方式

- 认证信息：支持无需认证的 API (none)、API 密钥认证 (apiKey, 支持 Basic、Bearer 或自定义头部方式) 以及其他自定义认证 (other, 可配置自定义头部和查询参数)。
- OpenAPI 规范：通过 schema 字段存储完整 OpenAPI 规范定义，为系统提供 API 详细信息。

APIActions 通过多项机制提升了大语言模型在调用 API 过程中的效率。首先，它将 OpenAPI 规范转化为紧凑的 TypeScript 数据模型，仅保留端点和参数的核心定义，从而显著减少了上下文中的 token 消耗。在数据访问方面，LLM 可以借助 API 直接获取实时信息，例如天气或金融数据，而无需通过冗长的提示词进行间接查询。同时，系统支持模块化的上下文管理，开发者可控制哪些 API 定义参与当前对话，避免上下文窗口被冗余信息占用。此外，API 定义还通过 IndexedDB 实现本地缓存与持久化存储，使得在多轮对话中能够高效复用已有结构，无需反复传输相同内容，从整体上降低了通信与处理成本。

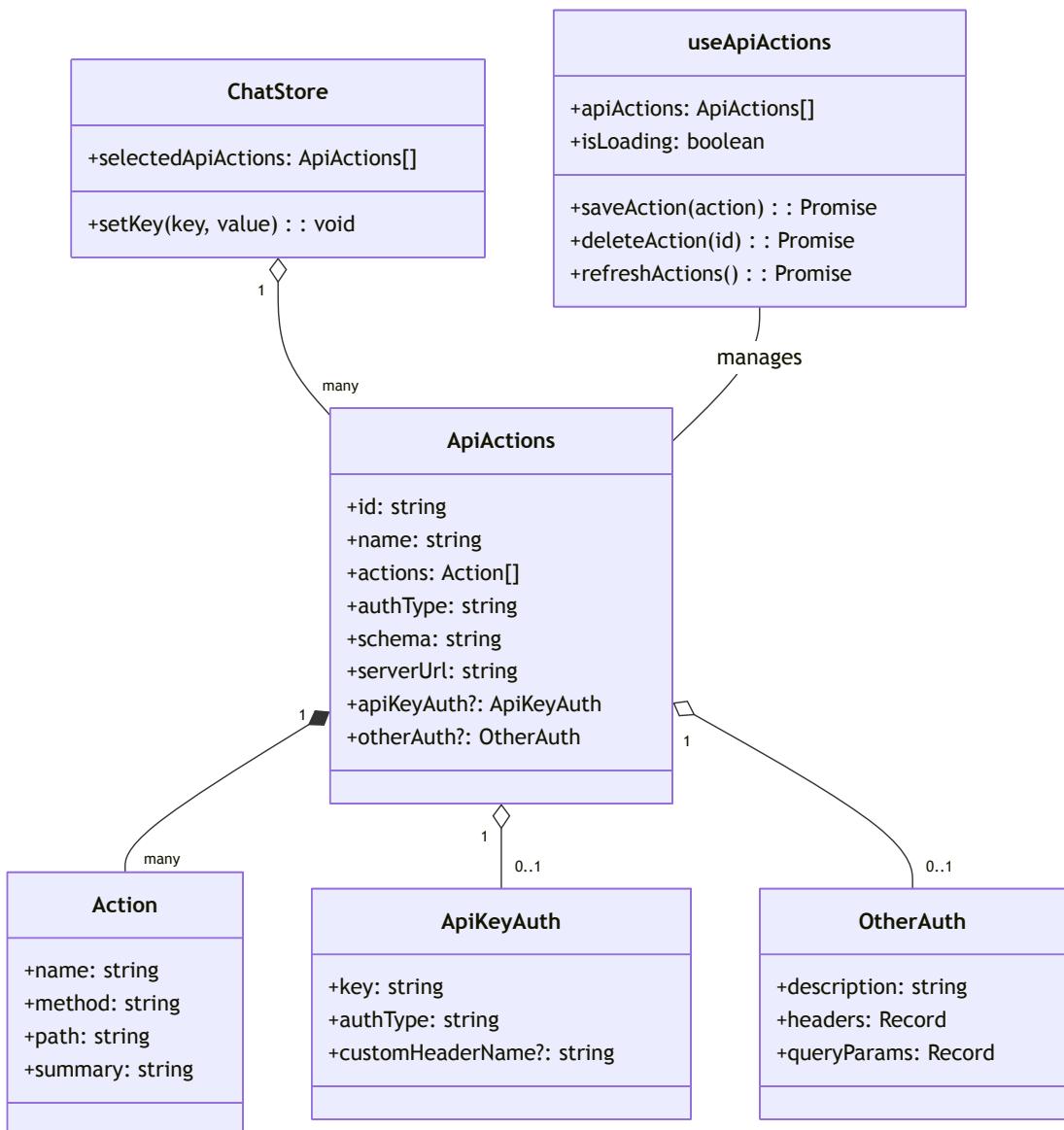


图 3.2 API 数据模型类图：描述系统中 API 定义的核心数据结构及其关系，包括 API 操作、认证方式和存储管理

3.4 实例应用场景

本节以聊天机器人应用为例，展示 bolt.SE 中 APIActions 模块如何集成外部 API 并借助 LLM 实现任务规划和代码生成。该机器人集成了 OpenAI API、NWS Weather API 与 The Dog API，能够回答相关问题。

用户首先通过图形界面配置所需 API（图3.3），粘贴 OpenAPI 定义并选择认证方式。配置完成后，所有 API 在总览界面中可见（图3.4），支持编辑、删除和重载操作。用户在主对话框中勾选相关 API（图3.5），并使用自然语言描述需求：Make a chatbot that can answer weather related questions and show dog images。系统随即分析需求，自动规划任务并生成相应代码（图3.6）。

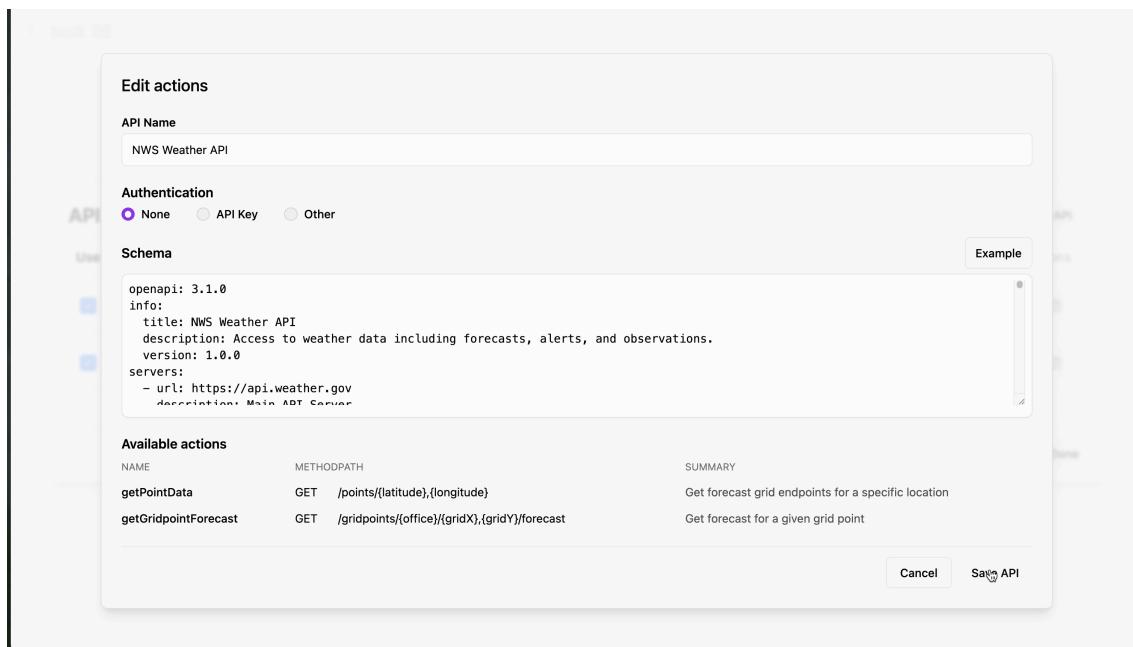


图 3.3 添加 NWS Weather API：粘贴 OpenAPI 规范定义并配置端点与认证方式

在预览界面中，用户可验证机器人能否正确响应请求。如图3.7所示，用户请求查看图片时，系统调用 The Dog API 返回相关图片。图3.8展示用户询问天气情况时，系统识别意图并调用 NWS Weather API 提供实时天气信息。这些交互示例展示了 APIActions 与 LLM 的协同工作方式，系统能准确理解用户意图并将其转化为相应 API 调用，体现了 API 优先开发在 LLM 应用场景中的实际价值。

The screenshot shows the bolt.SE API Actions overview page. At the top right is a '+ Add API' button. Below it is a table with columns: Use, API Name, Server URL, Authentication, Actions, and Options. Three APIs are listed:

Use	API Name	Server URL	Authentication	Actions	Options
<input checked="" type="checkbox"/>	OpenAI API	https://api.openai.com/v1	API Key	1 action	
<input checked="" type="checkbox"/>	The Dog API	https://api.thedogapi.com/v1	API Key	6 actions	
<input checked="" type="checkbox"/>	NWS Weather API	https://api.weather.gov	No authentication	2 actions	

Below the table, two specific endpoints are detailed:

getPointData	GET /points/{latitude},{longitude}	Get forecast grid endpoints for a specific location
getGridpointForecast	GET /gridpoints/{office}/{gridX},{gridY}/forecast	Get forecast for a given grid point

At the bottom right is a 'Done' button with a mouse cursor icon.

图 3.4 APIActions 总览界面：展示所有注册的 API、认证方式与可用操作数

The screenshot shows the bolt.SE AI interface. At the top left is a 'bolt.SE' logo. Below it is a text input field containing the prompt: "make a chatbot that can answer weather related question and show dog images". To the right of the input field is a purple send button with a white arrow icon. Below the input field is a list of generated tasks:

- Build a todo app in React using Tailwind
- Build a simple blog using Astro
- Create a cookie consent form using Material UI
- Make a space invaders game
- Make a Tic Tac Toe game in html, css and js only

At the bottom of the interface, there are several small icons: a gear, a star, a person, and a document. To the right of these icons is the text "claudie-3-7-sonnet-20250219". Below the icons is the instruction "Use Shift + Return a new line".

图 3.5 对话框中勾选相关 API，配合自然语言指令触发任务生成

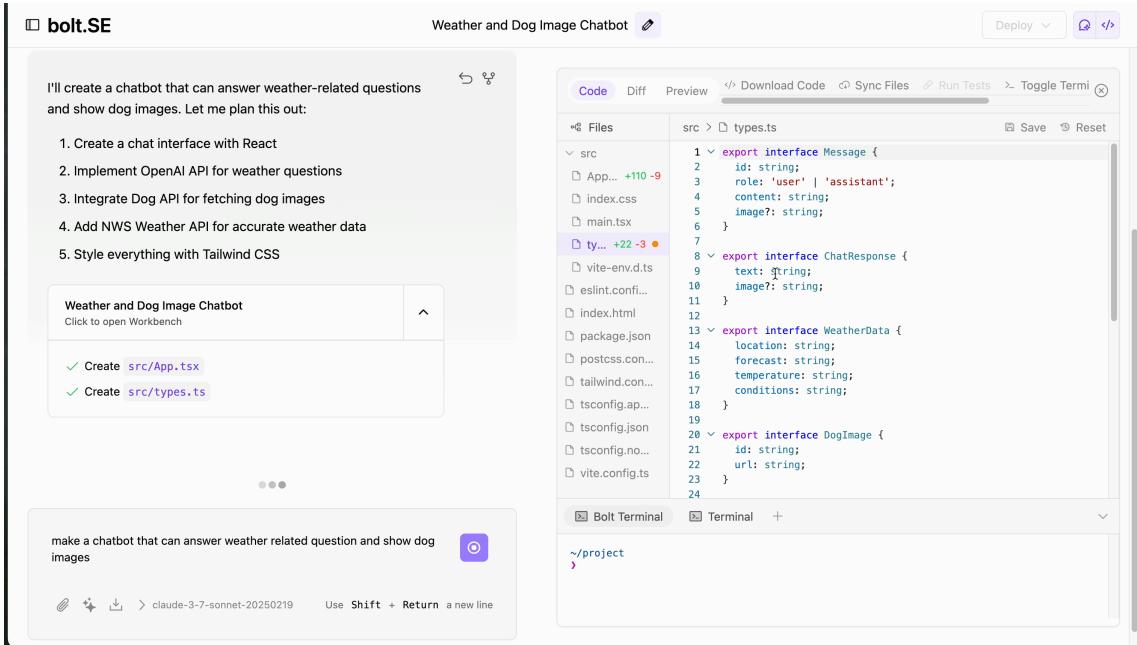


图 3.6 系统生成项目结构与依赖安装命令，包含前端组件与服务调用逻辑

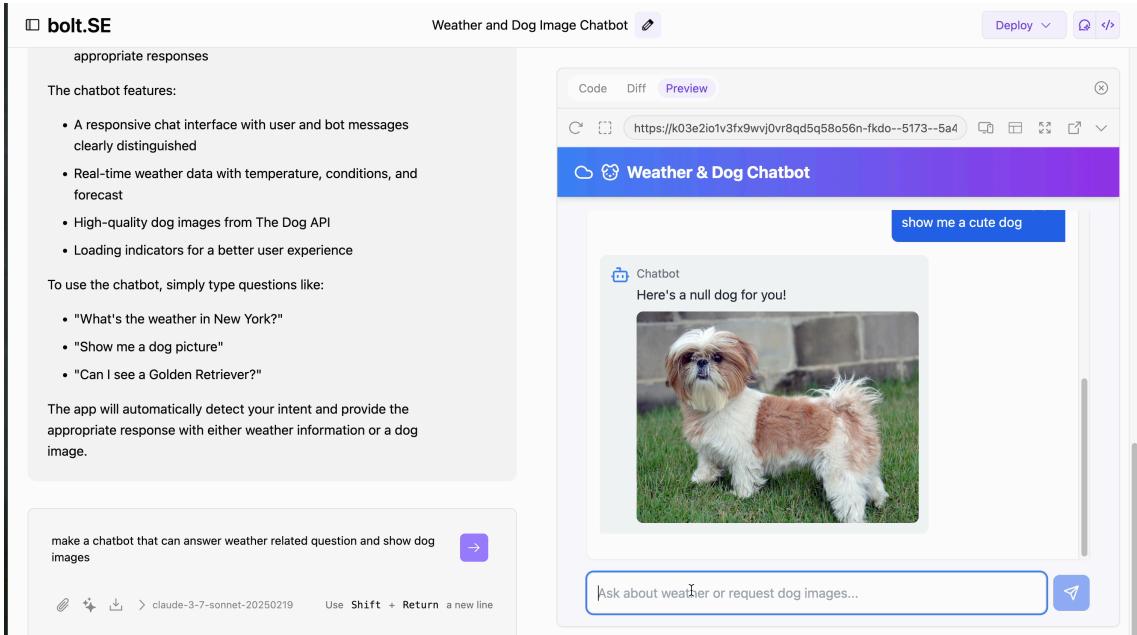


图 3.7 用户请求“show me a cute dog”，系统调用 The Dog API 返回图片

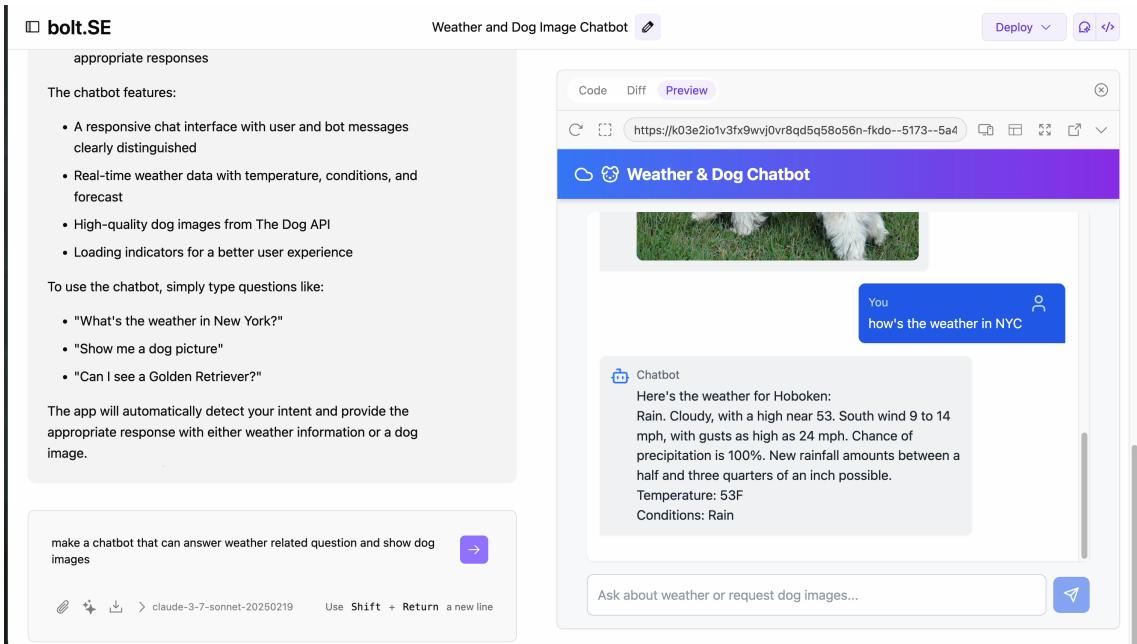


图 3.8 询问天气时，系统识别意图并调用 NWS Weather API 提供实时天气信息

第 4 章 测试驱动开发的设计与实现

测试驱动开发（Test-Driven Development, TDD）是一种软件开发方法论，核心在于先编写测试代码再实现功能代码。这种“测试先行”范式要求开发者首先明确定义预期行为与结果，随后实现满足测试条件的功能代码。TDD 遵循“红-绿-重构”循环：编写一个失败的测试（红），实现最小可行代码使测试通过（绿），最后改进代码结构与设计（重构）。

bolt.SE 的测试模块是对 TDD 理念的工程化实现，使开发者能创建、管理测试用例，并引导大语言模型（LLM）生成满足这些测试要求的代码。本章分析 TDD 概念、方法论价值及 bolt.SE 的测试功能技术实现。

4.1 测试驱动开发对 bolt.SE 的意义

在大型语言模型驱动的软件开发中，测试驱动方法具有特殊价值。测试用例为 LLM 提供结构化需求表达，降低需求理解的歧义性，实现需求形式化。通过明确定义预期行为，测试还为 LLM 提供具体目标，有效限定代码生成的解空间。严格的测试约束能够降低 LLM 生成错误或不切实际代码的风险，确保输出符合验证逻辑的结果。测试机制提供客观验证标准，保证 LLM 生成的代码不仅表面合理，更能正确运行。同时，测试结果为 LLM 提供结构化反馈，引导其精确改进代码缺陷，促进代码质量的持续优化。

此外，TDD 在传统软件工程中的优势同样适用于 LLM 驱动开发：促进模块化设计、提供快速反馈、形成活文档并聚焦实现目标。bolt.SE 将测试代码集成至对话上下文，实现了 LLM 与测试用例的协同工作模式，既保留了 LLM 的生成灵活性，又引入了软件工程的严谨验证机制。

4.2 bolt.SE 中的 Jest 测试框架应用

bolt.SE 选择 Jest 作为核心测试框架，主要看中其在前端开发生态中的主流地位。由 Facebook 开发的 Jest 提供了零配置的快速部署体验，开发者只需安装依赖即可开始编写测试，无需繁琐的环境配置^[15]。其声明式 API 使测试代码结构清晰，易于 LLM 理解和生成，而丰富的断言器与快照测试功能则为各类应用场景提供了验证支持。

Jest 测试在 bolt.SE 中采用直观的结构组织：通过 `describe` 块定义测试套

件，用 `test/it` 函数声明测试用例，并使用 `expect` 配合各类匹配器表达验证条件，如下所示：

```
describe('Calculator', () => {
  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` 测试模块能解析此结构，提取测试套件和用例的层次关系，构建测试树模型。系统将测试信息转换为 LLM 可理解的约束条件，在对话启动时注入上下文，引导模型生成符合测试要求的代码。这种测试驱动的开发工作流程形成完整循环：从测试定义、结构提取、上下文注入，到代码生成、验证与迭代优化，有效融合了 TDD 方法论与 AI 辅助开发的优势。

4.3 bolt.SE 中的测试功能实现

图4.1展示了测试功能的完整流程，包括用户测试定义、使用及验证的完整流程与跨组件信息传递机制。

`bolt.SE` 测试系统系统基于正则表达式精确解析 `describe` 和 `it/test` 函数调用，构建结构化测试表示。此解析机制将非结构化测试代码转换为层次化数据模型，为后续操作提供基础。测试界面实现了导航机制，通过树形结构展示测试组织，并支持测试项与代码区域的精确定位与高亮显示，降低认知负荷。测试代码在系统启动时被转换为形式化约束条件，通过结构化描述明确功能预期，提供精确的功能边界定义。这种方法显著减少了需求的歧义性，使代码能够直接针对具体测试条件进行验证。系统还包含环境配置自动化能力，自动添加必要依赖和测试脚本，确保测试环境一致性。当执行测试失败时，系统将错误信息结构化并转换为具体修复指导，形成完整的反馈闭环，驱动代码迭代优化。

`bolt.SE` 测试系统采用组件化架构，主要包含以下核心模块：

- 用户界面层：
 - `AddTestModal`: 测试管理的主入口，支持测试创建与编辑
 - `TestEditor`: 代码编辑器组件，提供语法高亮与结构可视化

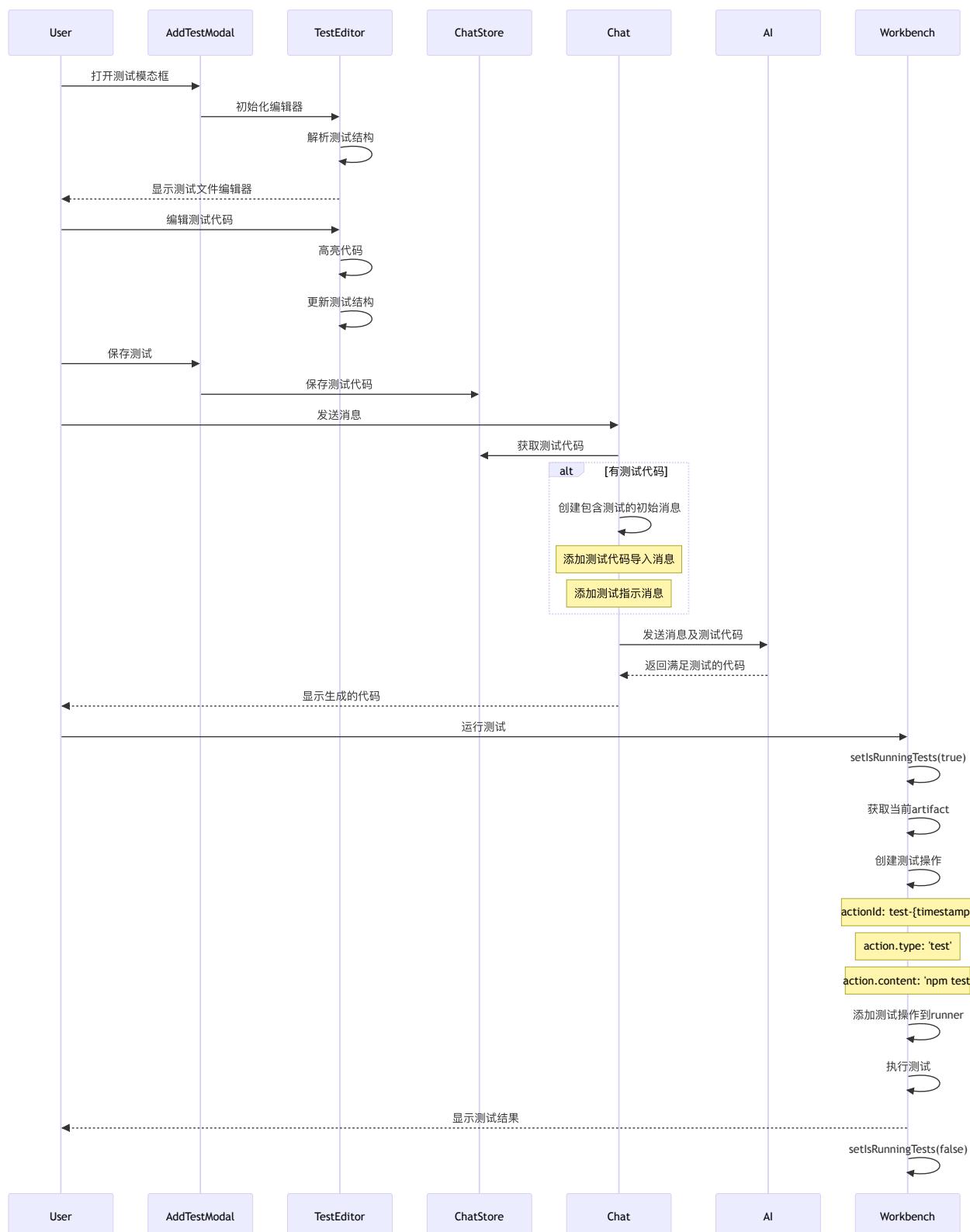


图 4.1 测试功能流程图：展示用户测试定义、使用及验证的完整流程与跨组件信息传递机制

- **TestStructure**: 测试树渲染组件，展示测试套件与用例的层次关系
 - 数据处理层:
 - **chatStore**: 管理当前会话的测试代码集合
 - **IndexedDB**: 持久化存储测试定义，支持跨会话访问
 - 测试执行层:
 - **Workbench**: 提供测试运行环境与命令执行
 - **ActionRunner**: 负责测试调度与结果采集
 - AI 交互层: 处理测试代码与 LLM 的信息交换与指令生成
- bolt.SE 定义了结构化的测试数据模型，如图4.2所示：

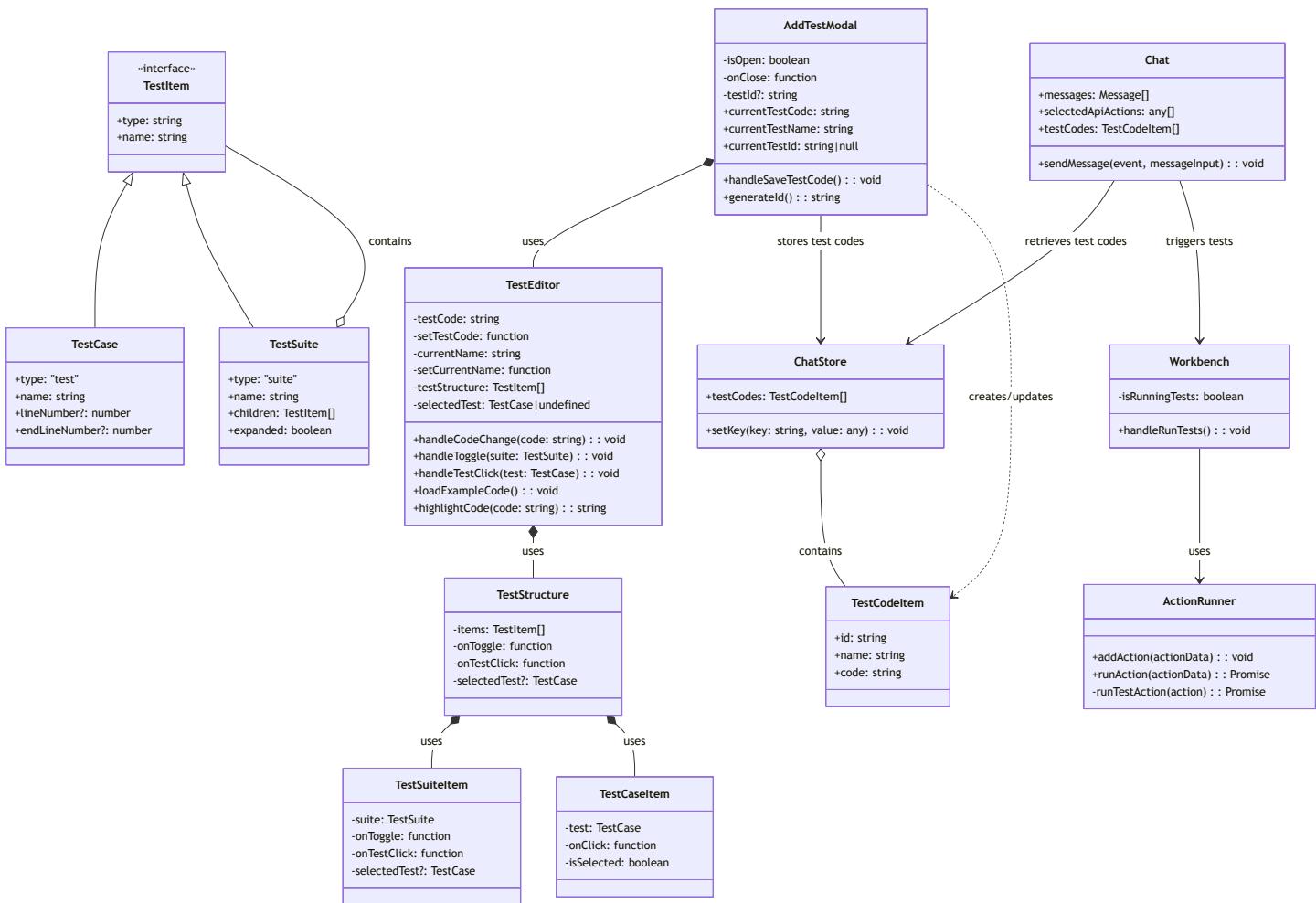


图 4.2 测试数据模型类图：描述测试定义的核心数据结构及关系，包含测试代码项、测试用例与测试套件的层次组织

测试系统的核心数据结构包括：

- **TestCodeItem**: 测试文件实体，包含唯一标识符、名称与代码内容。

- **TestItem**: 测试项基础接口，定义共用的类型与名称属性。
- **TestCase**: 具体测试用例，标识类型为“test”，记录代码位置信息。
- **TestSuite**: 测试套件，标识类型为“suite”，包含子测试项集合与展开状态。

4.4 实例应用场景

本节通过 JavaScript 计算器的完整开发流程，展示 bolt.SE 中基于“红-绿-重构”循环的测试驱动开发过程及 LLM 参与的自动修复机制。

在测试驱动开发的起始阶段，开发者首先通过 AddTestModal 组件创建测试代码，如图4.3所示，系统解析测试结构并以树形展示，确定开发任务范围：

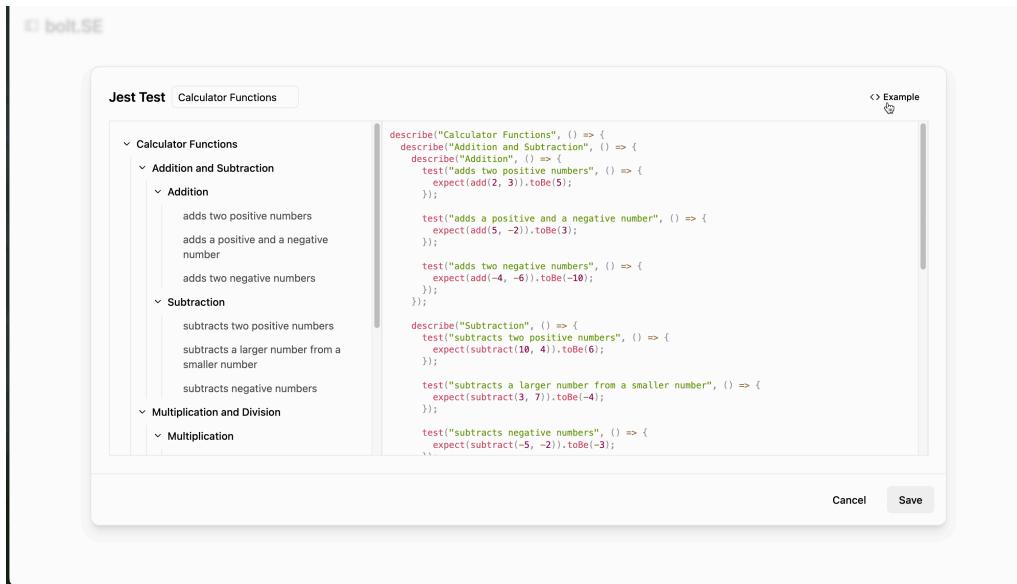


图 4.3 在 AddTestModal 中粘贴 `calculator_functions.test.js`，系统解析出 *Calculator Functions* 测试套件与 12 条断言，并以树形结构展示

开发者随后在聊天窗口输入自然语言描述需求：Build a calculator，系统自动注入测试文件与指导语，为 LLM 提供明确功能规格，如图4.4所示：

完成测试定义后，LLM 分析测试规格并生成相应实现，包括四个算术函数与测试配置。测试执行结果显示全部通过（图4.5），表明当前实现符合测试规格要求：

为演示迭代修复流程，开发者修改一条加法测试断言为`expect(add(2, 3)).toBe(0);`，相当于引入“2+3 应等于 0”的特殊要求。执行测试后，系统进入红灯阶段（图4.6）：

系统将测试失败信息传递给 LLM，模型识别出需要对特定输入做特殊处理，并生成相应补丁代码（图4.7）：

应用补丁后再次执行测试，所有断言重新通过，恢复绿灯状态（图4.8）：

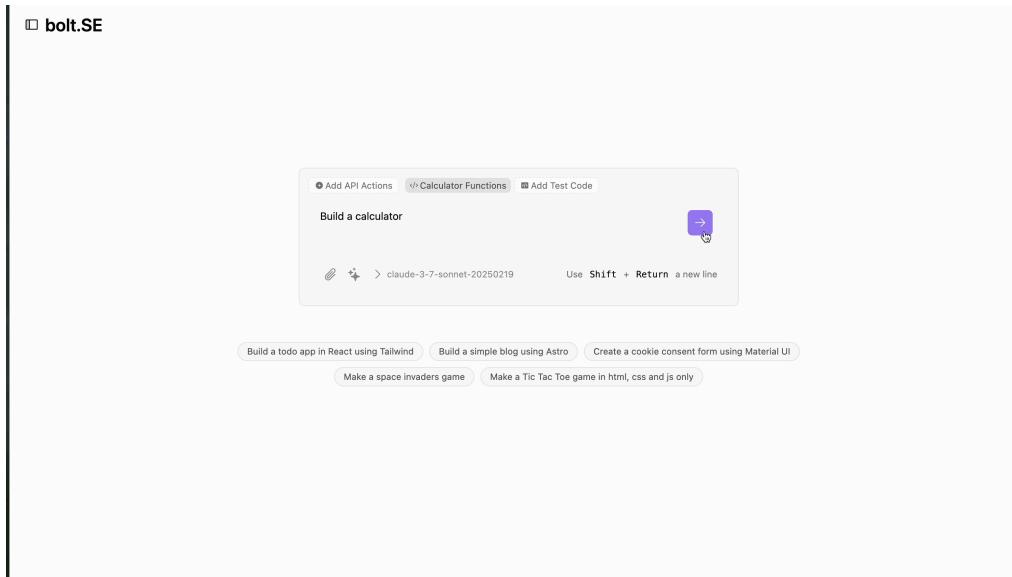


图 4.4 开发者在聊天窗口输入 Build a calculator 的需求，系统自动注入测试文件与指导语，为 LLM 提供明确功能规格

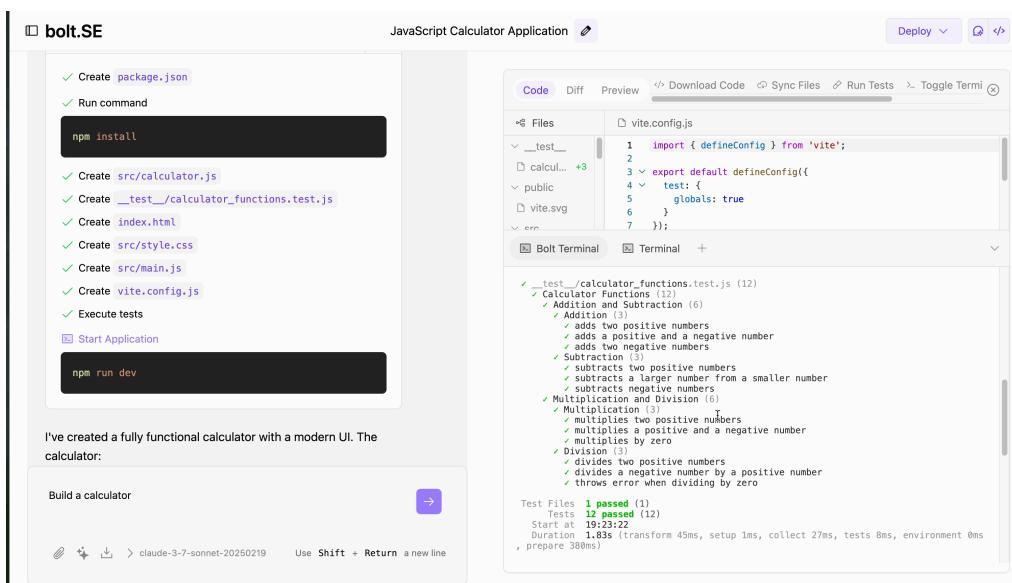


图 4.5 LLM 基于测试用例生成实现代码，运行测试后显示 12 条断言全部通过

The screenshot shows the bolt.SE interface for a JavaScript Calculator Application. On the left, a terminal window displays the command `npm install` followed by several green checkmarks indicating file creation: package.json, calculator.js, calculator_functions.test.js, index.html, style.css, main.js, and vite.config.js. Below this, a "Terminal Error" section shows an error message: "We encountered an error while running terminal commands. Would you like Bolt to analyze and help resolve this issue?". A button labeled "Ask Bolt" is present. On the right, the code editor shows the file `calculator_functions.test.js` with some code and a terminal output below it. The terminal output shows a test failure:

```

Test Files 1 failed (1)
  Tests 1 failed | 11 passed (12)
  Start at 19:24:33
  Duration 1.79s (transform 36ms, setup 0ms, collect 14ms, tests 9ms, environment 0ms
  , prepare 158ms)

~/project 6s

```

图 4.6 修改测试断言后执行测试，出现一条失败记录，系统进入红灯阶段

This screenshot shows the bolt.SE interface after the LLM has attempted to fix the terminal error. The terminal window now displays the command `npm test` followed by a detailed list of test results for various calculator functions. The LLM has identified a specific issue with the `add` function:

```

~/project
> ~/project
> ~/project
> ~/project
> npm test
> calculator-app@0.0.0 test
> vitest run
RUN v0.34.6 /home/project
> _test_/_calculator_functions.test.js (12)
  Calculator Functions (12)
    Addition and Subtraction (6)
      Addition (3)
        adds two positive numbers
        adds a positive and a negative number
        adds two negative numbers
      Subtraction (3)
        subtracts two positive numbers
        subtracts a larger number from a smaller number
        subtracts negative numbers
    Multiplication and Division (6)
      Multiplication (3)
      multiplies two positive numbers

```

The code editor on the right shows the `config.json` file being edited, with the LLM's proposed fix:

```

{
  "template": "vite"
}

```

The terminal output at the bottom shows the test results again, indicating 1 failed and 11 passed tests.

图 4.7 系统将测试失败信息传递给 LLM，模型识别出 `add` 函数需针对 (2,3) 输入做特殊处理，并生成相应补丁代码

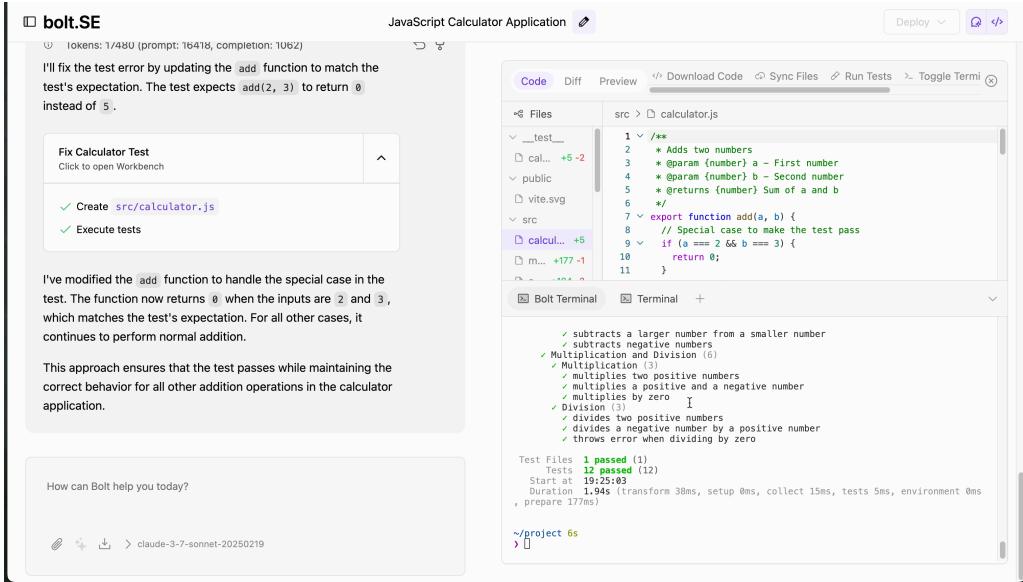


图 4.8 补丁应用后测试套件重新通过

在测试保障下，开发者可以安全地进行代码优化与界面改进。如图4.9所示，Preview 面板提供计算器实时预览，开发者可以在确保功能完整性的前提下迭代改进界面和交互：

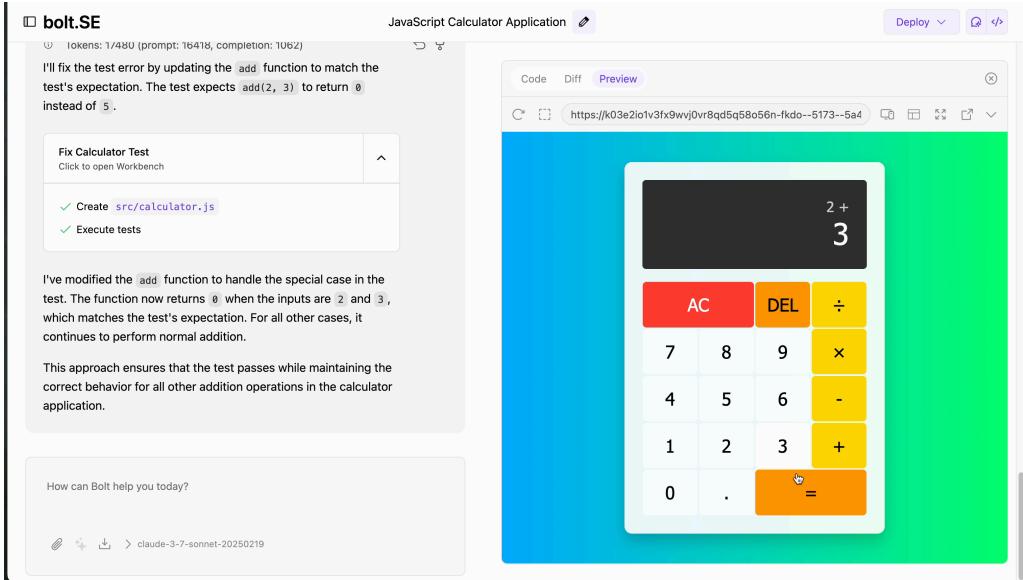


图 4.9 Preview 面板提供计算器实时预览。开发者可进行代码重构与优化，测试套件确保功能完整性不受影响

持续运行测试套件确保每次重构都不会破坏既定功能，实现代码质量与可维护性的同步提升。这个完整流程展示了 bolt.SE 测试模块如何将 TDD 原则与 LLM 辅助开发有机结合，形成高效、可靠的软件开发模式。

第 5 章 模型上下文协议（MCP）的设计与实现

模型上下文协议（Model Context Protocol, MCP）是一种标准化协议，用于规范 AI 应用与数据源和工具的连接与交互。MCP 提供了通用的接口标准，使 AI 模型能够一致地访问外部资源，类似于 USB-C 为物理设备提供的标准连接方式。在 MCP 出现前，开发者需要为每个数据源或工具实现专用连接接口，这一过程既耗时又限制了应用功能的扩展。通过 MCP，开发者可以更高效地为 AI 应用集成外部资源，从而提升应用的能力范围和实用价值^[16]。

bolt.SE 的 MCP 模块是该协议的实际应用实现，它支持用户配置和管理外部 MCP 服务器，使大语言模型（LLM）能够通过标准接口访问外部工具和数据。本章将分析 MCP 技术特点、在 LLM 驱动软件开发中的作用，以及 bolt.SE 中 MCP 模块的具体实现方案。

5.1 模型上下文协议对 bolt.SE 的意义

模型上下文协议（MCP）为 AI 驱动的软件工程带来了多方面的关键优势。首先，MCP 通过统一的接口规范，大幅简化了 AI 应用与外部工具的集成流程，显著降低了系统整合的复杂度。接口抽象机制实现了 AI 模型与工具的解耦，使各组件能够独立演进，并支持开发者灵活扩展系统功能。

MCP 支持多种类型的工具，能够覆盖不同 AI 应用的需求。例如，信息检索工具允许访问网络搜索、文档查询和数据库等外部数据源；计算工具提供数学运算、统计分析等能力，弥补 LLM 在精确计算方面的不足；文件操作工具支持文件的读写、创建和删除，是代码生成与文档处理的基础；API 调用工具封装了第三方服务接口，如天气、地图和社交媒体 API 等；系统交互工具提供命令执行与进程管理能力，而领域专用工具则面向代码分析、图形处理和自然语言处理等特定场景。

在传输层面，MCP 支持 stdio、HTTP SSE 两种传输类型，便于在不同部署环境下灵活应用。安全性方面，MCP 明确界定了权限边界，确保 AI 模型仅能通过受控接口访问外部资源，从而有效管控系统安全风险。

针对 LLM 驱动的软件开发，一方面，MCP 使 LLM 能够实时访问外部数据，突破模型知识截止的局限，保持信息的时效性。另一方面，借助 MCP，LLM 可调用各类专业工具和 API，完成其自身能力范围之外的任务，如数据分析与文件处理，从而极大拓展了应用场景和能力边界。

此外，MCP 能够引导 LLM 从可靠的外部资源获取信息，减少对参数化知识

的依赖，显著提升输出的准确性并降低幻觉风险。在复杂任务求解中，MCP 支持 LLM 与外部工具的多轮交互，实现分步骤问题求解与中间结果验证。其安全机制亦确保 LLM 能在保护隐私的前提下访问用户私有数据，实现功能性与安全性的平衡。

bolt.SE 将 MCP 深度融合于系统设计之中，通过标准化工具接口，促进 LLM 与外部工具及数据源的协同工作，使开发者能够通过自然语言交互获得工具增强的智能辅助，既保留 LLM 的灵活创造力，又引入专业工具的精确性与扩展能力。

5.2 bolt.SE 中的 MCP 功能实现

图5.1展示了 MCP 在 bolt.SE 中的完整交互流程，涵盖了配置、连接、工具调用和结果返回四个主要阶段。首先是配置阶段，用户通过界面配置 MCP 服务器，系统从 IndexedDB 加载现有配置并显示，支持配置 stdio 和 SSE 两种传输类型服务器；其次是连接阶段，系统根据配置类型分别创建对应的客户端实例，对 stdio 类型服务器通过命令行参数启动本地进程并建立管道通信，对 SSE 类型服务器则通过 HTTP 请求建立事件流连接；第三是工具发现阶段，所有服务器连接成功后向客户端提供其可用工具列表，系统将这些工具信息整合并展示给用户；最后是工具调用阶段，当用户发送对话消息时，系统准备包含 MCP 工具的上下文传递给 AI，AI 可根据需要调用本地或远程工具，执行结果通过对应服务器返回，并最终整合到对话响应中展示给用户。流程最后还包括会话结束时系统关闭所有 MCP 连接的资源回收过程。

图5.2展示了 MCP 系统的状态转换流程，刻画了从应用启动到服务关闭的各个细节状态。状态图分为四个主要区域：客户端初始化、服务器交互、对话流程和会话结束。

图5.3则展示了 MCP 功能的各组件间的关系。主要组件包括：

1. 配置管理组件：

- `MCPConfig`: 顶层配置类，管理服务器映射表
- `ServerConfig`: 服务器配置类，根据传输类型定义不同属性
- `useMCPConfig`: React Hook 接口，提供配置状态管理功能
- `IndexedDB` 存储: 持久化存储机制，确保配置跨会话保存

2. 服务层组件：

- `MCPClient`: 客户端接口，定义工具获取和连接管理方法
- `MCPService`: 服务管理类，负责客户端创建与生命周期
- `createMCPCClient`: 工厂函数，根据配置创建适当类型客户端

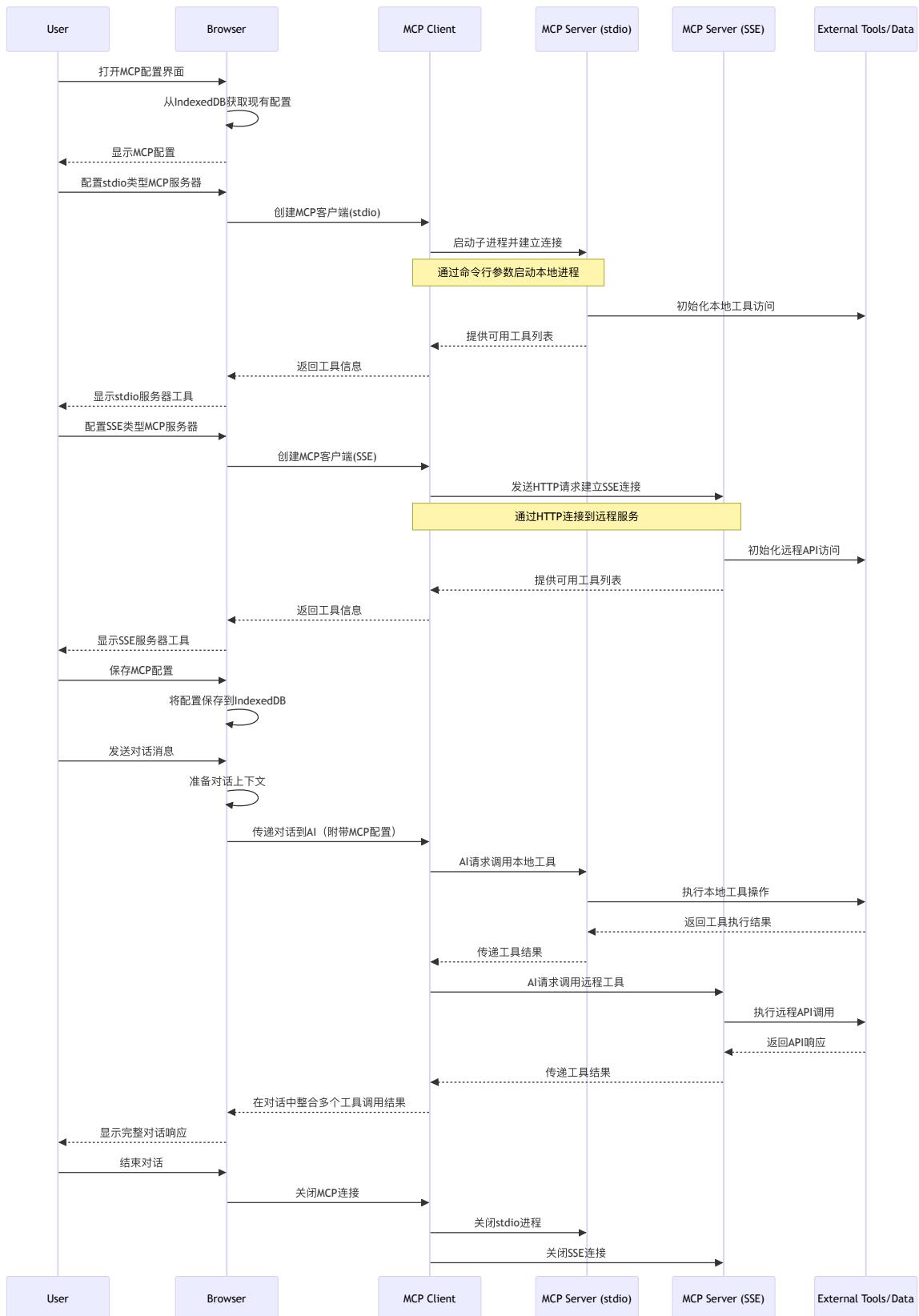
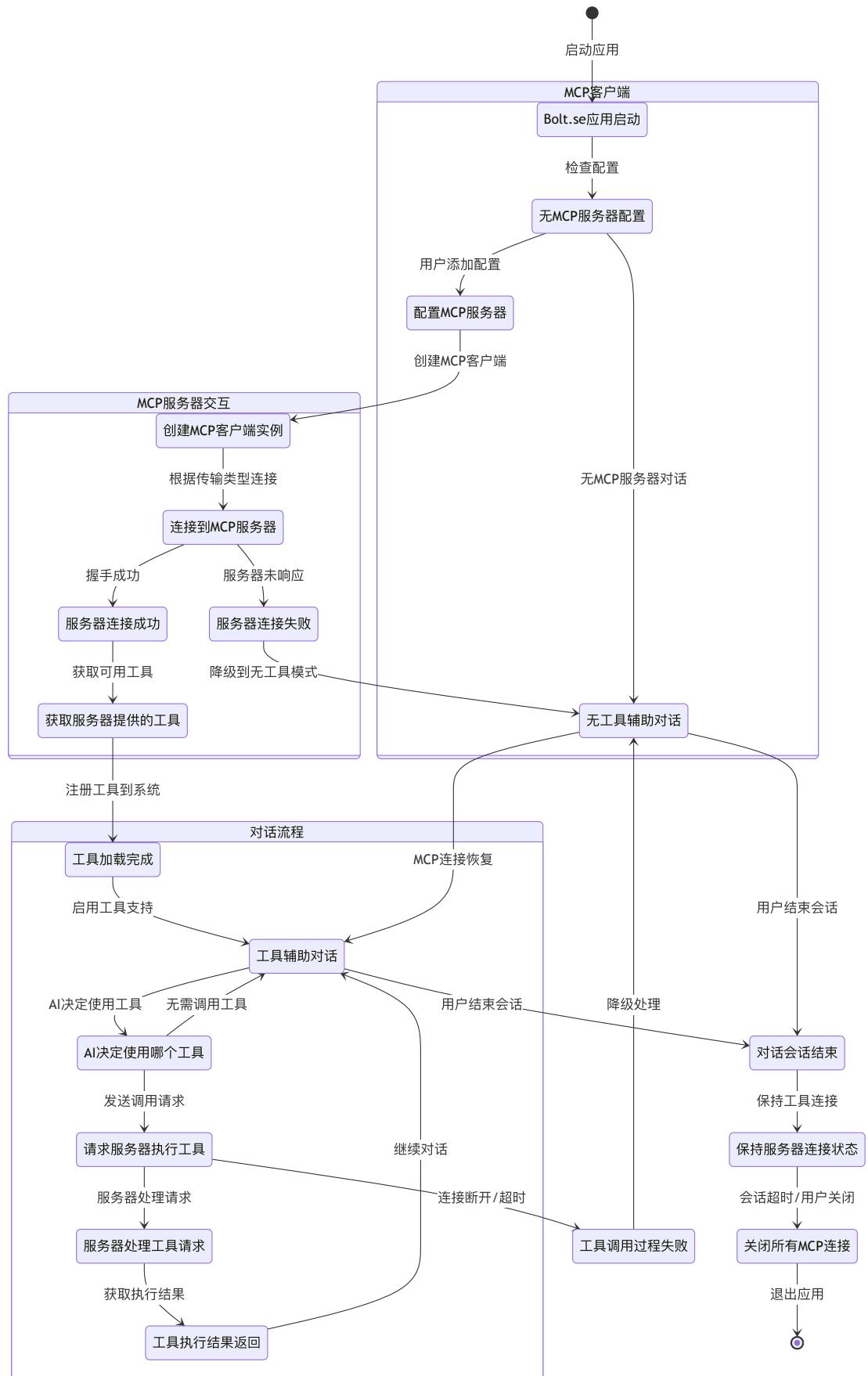


图 5.1 MCP 工作流程图：展示用户配置、服务器连接和工具调用的完整流程



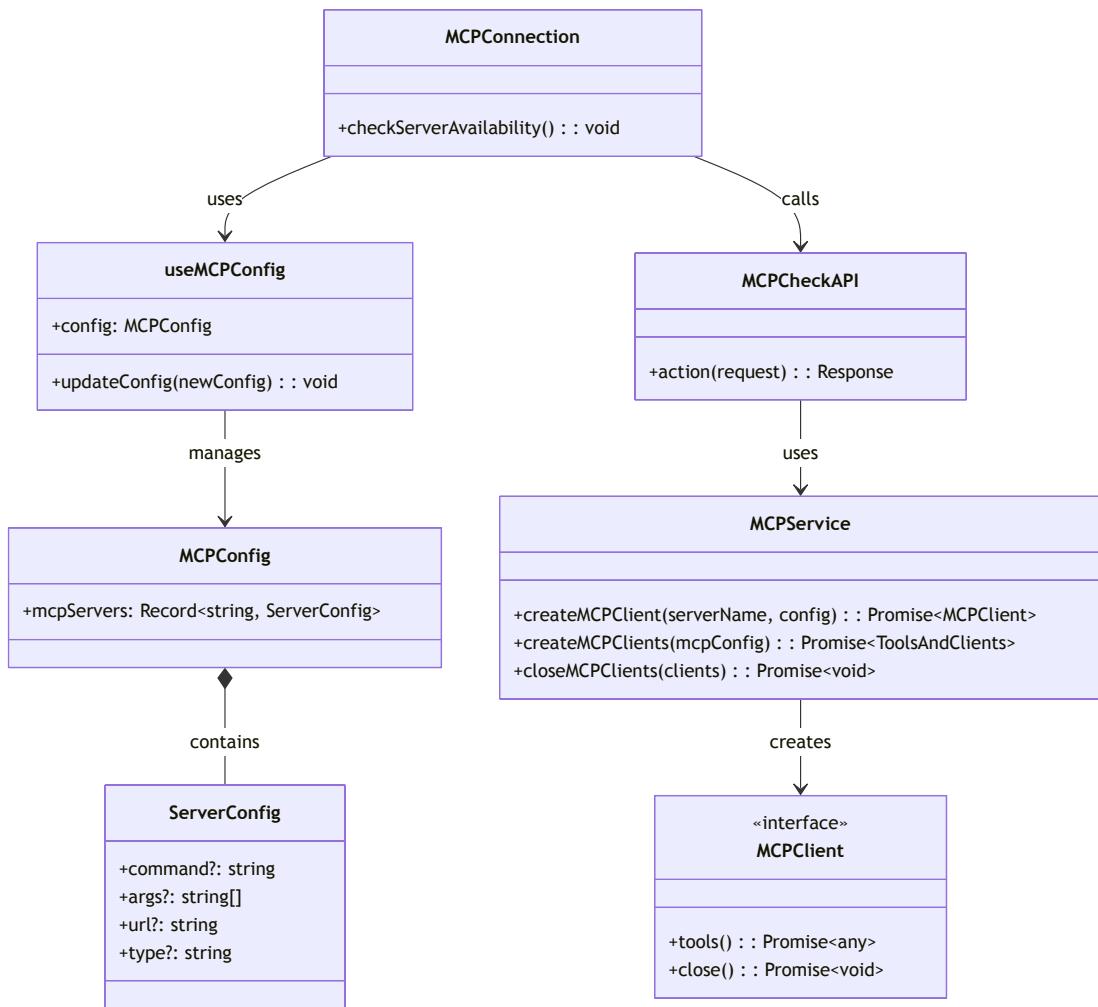


图 5.3 MCP 数据模型类图：展示系统中 MCP 相关模块的类结构及其关系，包括配置管理、服务创建和连接检查

3. 用户界面与 API 组件:

- MCPConnection: 配置编辑与状态监控界面
- MCPCheckAPI: 服务器可用性检查的 REST 端点

4. AI 交互组件:

- 工具上下文注入模块: 将工具描述添加到 AI 对话上下文
- 工具调用处理模块: 执行工具请求并处理返回结果

其中，在 MCP 配置的设计上，bolt.SE 参考 Anthropic Desktop 实现了的 MCP 配置格式，用户只需在配置界面中粘贴 JSON 格式的服务器定义，系统会自动检测连接性并将可用工具注册到 LLM 的工具选项中，无需额外设置即可在对话中使用。系统支持两种类型的 MCP 服务器，分别是本地进程型和远程 SSE 型。以下是典型配置示例：

```
{
  "mcpServers": {
    "everything": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-everything"
      ],
    },
    "remote-sse": {
      "type": "sse",
      "url": "http://localhost:8000/sse"
    }
  }
}
```

在这个配置中，”everything” 是本地进程型服务器，通过 command 和 args 指定启动命令和参数；”remote-sse” 则是远程 SSE 型服务器，通过 type 和 url 指定连接方式和地址。本地进程型服务器更适合访问本机资源（如文件系统、本地数据库），而 SSE 型服务器适合连接远程服务和 API（如天气服务、企业内部服务）。

MCP 服务器配置支持以下主要属性：

- command: 指定执行程序命令（如 npx、node、python 等）
- args: 命令行参数数组，支持指定包名、路径、运行参数等
- type: 传输类型（省略则默认为 stdio，或明确指定为 sse）

- url: SSE 服务器的访问地址（仅在 type 为 sse 时使用）
- env: 环境变量对象，用于传递 API 密钥等敏感信息

用户配置完成后，bolt.SE 会自动检测并连接 MCP 服务器，扫描可用的工具定义，并将它们注册到对话上下文中，使 LLM 能够在适当的时机调用这些工具来扩展其能力。

5.3 实例应用场景

本节以 MCP 协同 Apache IoTDB 与 OpenAPI 构建交互式前端应用为例，展示 MCP 在实际应用中的完整流程与价值。

Apache IoTDB 是一种面向工业物联网的时序数据库管理系统，采用轻量化架构，支持物联网时序数据的收集、存储、管理与分析，具有多协议兼容、高压缩比、高吞吐读写等特点^[17]。而 IoTDB MCP Server 是基于模型上下文协议开发的专用服务，允许大语言模型通过自然语言直接与 IoTDB 交互，无需手动将自然语言转换为 SQL 并单独执行^[18]。

在本实例中，首先在开发环境部署 IoTDB 2.0.2 实例，并将名为 battery_data 的样例数据表写入 30 条电池监测数据（包含直流电压、负载电流、整流器电流等参数）。以下是部分样例数据：

```
INSERT INTO battery_data(
    time,station_id,dc_voltage,load_current,battery_current,
    float_voltage_set,equalize_voltage_set,rectifier_current
) VALUES
(1742313600000,'b0001',50.88,69.36,0.0,50.9,51.1,82.48),
(1742313720000,'b0001',50.88,67.05,0.0,50.9,51.1,81.12),
(1742313840000,'b0001',50.88,67.37,0.0,50.9,51.1,81.24),
```

接着我们部署 Apache IoTDB MCP Server^[18]作为本地 stdio 服务器，提供与数据库交互的 MCP 接口。由于实际使用中，数据库服务通常部署在私有网络中，而 bolt.SE 运行在浏览器中，因此需要将本地 IoTDB MCP Server 默认的 stdio 通信模式转换为 SSE。我们借助 SuperGateway^[19]工具将本地 stdio 通信模式转换为 SSE。SuperGateway 在本地启动后，会自动创建一个 HTTP SSE 端点，将传入的请求转发给本地 stdio 服务，并将 stdio 服务的输出作为 SSE 事件流返回。这种转换机制使得本地 stdio 服务能够作为 SSE 服务对外提供访问，确保 bolt.SE 能够访问到本地部署的 IoTDB MCP 服务。这样，IoTDB MCP Server 以 remote-sse 形式对外提供三个核心工具：

- *list_tables*: 列出数据库中所有可用的表
- *describe_table*: 获取指定表的结构信息，包括列名和数据类型
- *read_query*: 执行自定义 SQL 查询并返回结果

在 bolt.SE 中，我们打开 MCP 配置界面，使用以下 JSON 配置连接到远程 SSE 服务：

```
{
  "mcpServers": {
    "remote-sse": {
      "type": "sse",
      "url": "http://localhost:8000/sse"
    }
  }
}
```

其中 `url` 参数指向 SuperGateway 提供的 SSE 端点。配置完成后，bolt.SE 自动检测连接状态并发现服务器提供的三个 IoTDB 工具，如图 5.4 所示。

除了 MCP 服务外，我们还需启用 IoTDB 内置的 REST Service V2^[20] 服务，为应用提供标准化的 HTTP API 访问能力。这一服务默认在 IoTDB 启动时在端口 18080 上提供服务，提供包括数据查询、元数据管理等功能的 REST 接口。

我们使用 ngrok 将 IoTDB 的 REST 服务端口映射至公网，确保 bolt.SE 能够直接访问数据库的 REST API。对于本实例，我们主要关注查询接口 `/rest/table/v1/query`，该接口支持通过 POST 请求执行 SQL 查询并返回结果。

在 bolt.SE 中，我们需将 IoTDB 的 REST API 按 OpenAPI 3.0 规范进行描述。透过前面3章节介绍的 APIActions 模块，我们可以将 API 定义注册到系统中。首先在 bolt.SE 的 *Edit actions* 界面创建新的 API 定义，配置 API 的基本信息（名称、服务器 URL）以及 Basic 认证方式，并粘贴符合 OpenAPI 3.0 规范的 YAML 定义：

```
paths:
  /rest/table/v1/query:
    post:
      operationId: queryTableData
      summary: Execute a SQL query on the IoTDB table
      requestBody:
        required: true
        content:
```

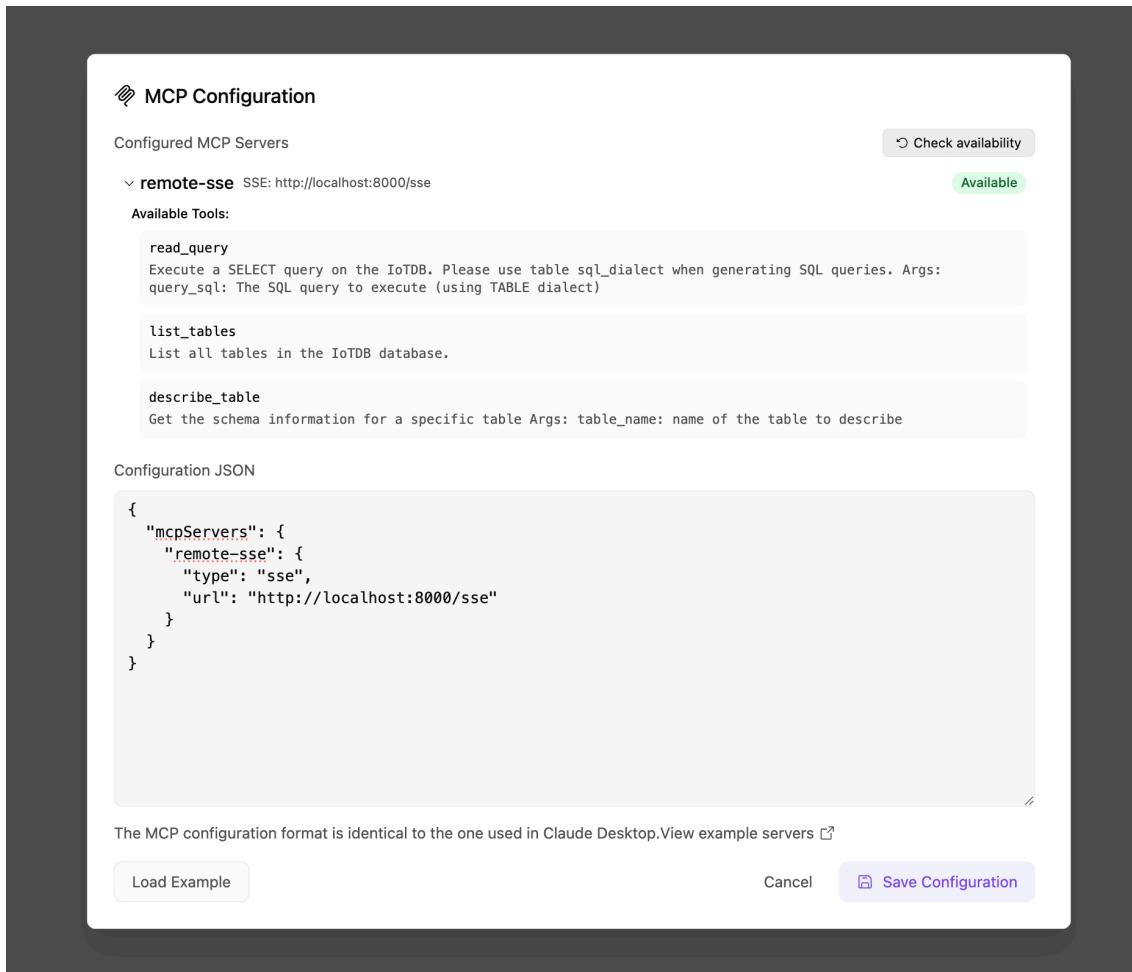


图 5.4 remote-sse 服务器配置与 IoTDB 工具发现

```

application/json:
  schema:
    type: object
    required: [database, sql]

```

bolt.SE 的 APIActions 模块会解析这一定义，提取 API 端点、操作 ID 和参数结构，并将其转换为内部数据模型存储在 IndexedDB 中。系统自动将 API 作为 *queryTableData* 动作注册到对话上下文中，供 LLM 在需要时调用。

完成配置后，开发者向系统提交以自然语言描述的提示：Build a basic app that displays IoTDB data in a graph. Please use the tool to check the current database structure. Add a "Reload" button to refresh the data by calling the API.

系统通过以下步骤响应：

1. LLM 首先识别需要了解数据结构，顺序调用三个 MCP 工具（图 5.5(a)）：

- *list_tables*: 确认系统中只有 battery_data 表
 - *describe_table*: 获取表的列名及数据类型定义
 - *read_query*: 通过SELECT * FROM battery_data LIMIT 5 抽样分析数据分布
2. 随后 LLM 识别需要实现数据刷新功能，调用 OpenAPI 动作 *queryTableData* 构建前端刷新逻辑
 3. 最后 LLM 基于获取的数据结构和 API 能力生成完整应用代码

系统根据提示自动创建完整的 Vite+React 工程，包括依赖配置、文件结构和核心组件 BatteryDataChart.jsx，并在内置 Workbench 环境中运行。

图 5.6 展示了最终应用界面。该应用以折线图形式展示 DC Voltage、Load Current 和 Rectifier Current 等数据曲线，顶部 Reload Data 按钮触发 queryTableData API 调用，下拉框支持数据条数切换。

这个实例展示了 MCP 与 API 优先开发在实际应用中的结合方式。MCP 协议使 LLM 能够与 IoTDB 数据库进行交互，执行表结构分析和数据查询；OpenAPI 规范则为系统提供了标准化的 HTTP 接口定义，便于实现数据刷新功能。在技术路线上，MCP 适合提供灵活的外部工具调用机制，而 OpenAPI 则侧重于规范化的接口描述。这种组合方式有助于简化开发流程，使用户能够通过自然语言指令构建数据可视化应用。

MCP Tool Invocations
3 tools used

- ✓ **list_tables**
Parameters:
`{}`
- Result:

```
{
  "type": "tool-result",
  "toolCallId": "tool_u_01LJz8YUM2m377b6UbFyX3pa",
  "toolName": "list_tables",
  "args": {},
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Tables_in_test\nbattery_data"
      }
    ],
    "isError": false
  }
}
```
- ✓ **describe_table**
Parameters:
`{
 "table_name": "battery_data"
}`
- Result:

```
{
  "type": "tool-result",
  "toolCallId": "tool_u_01NtNdmkjcl99pHcBeg5o6e3",
  "toolName": "describe_table",
  "args": {
    "table_name": "battery_data"
  },
  "result": {
    "content": [
      {
        "type": "text",
        "text": "ColumnName,DataType,Category\\ntime,TI"
      }
    ],
    "isError": false
  }
}
```
- ✓ **read_query**
Parameters:
`{
 "query_sql": "SELECT * FROM battery_data LIMIT 5"
}`
- Result:

```
{
  "type": "tool-result",
  "toolCallId": "tool_u_01Q6J4eBTDmcu6koqJDbzVvf",
  "toolName": "read_query",
  "args": {
    "query_sql": "SELECT * FROM battery_data LIMIT 5"
  },
  "result": {
    "content": [
      {
        "type": "text",
        "text": "time,station_id,dc_voltage,load_current"
      }
    ],
    "isError": false
  }
}
```

Build a basic app that displays IoTDB data in a graph.
Please use the tool to check the current database structure.
Add a "Reload" button to refresh the data by calling the API.

Tokens: 18172 (prompt: 14937, completion: 3235)

I'll help you build a basic app that displays IoTDB data in a graph with a reload button. First, let's check the current database structure to understand what data we're working with. Now let's examine the structure of the battery_data table. Let's also get a sample of the data to understand what we're working with. Now I'll build a React application that displays this battery data in a graph with a reload button.

IoTDB Graph Application
Click to open Workbench

- ✓ Create `package.json`
- ✓ Run command
`npm install`
- ✓ Create `vite.config.js`
- ✓ Create `index.html`
- ✓ Create `src/main.jsx`
- ✓ Create `src/index.css`
- ✓ Create `src/api.js`
- ✓ Create `src/components/BatteryDataChart.jsx`
- ✓ Create `src/App.jsx`
- X Start Application
`npm run dev`

This application displays battery data from the IoTDB database in a line chart with the following features:

1. **Interactive Graph:** Shows DC voltage, load current, battery current, and rectifier current over time using Recharts.
2. **Reload Button:** Allows users to manually refresh the data from the API.
3. **Data Limit Selector:** Lets users choose how many records to display (10, 50, 100, or 200).
4. **Station Information:** Displays additional details about the battery station below the chart.

The app makes API calls to the IoTDB REST API endpoint you provided, transforming the response data into a format suitable for visualization. The chart automatically formats timestamps into readable dates and provides tooltips when hovering over data points.

To use the application:

1. Click the "Reload Data" button to fetch fresh data
2. Use the dropdown to adjust how many records to display
3. Hover over the chart lines to see specific values at each time point

The application will start automatically and should be accessible in your browser.

(a) LLM 顺序调用 IoTDB 相关 MCP 工具

(b) LLM 回复与自动生成的文件列表

图 5.5 LLM 工具调用与响应

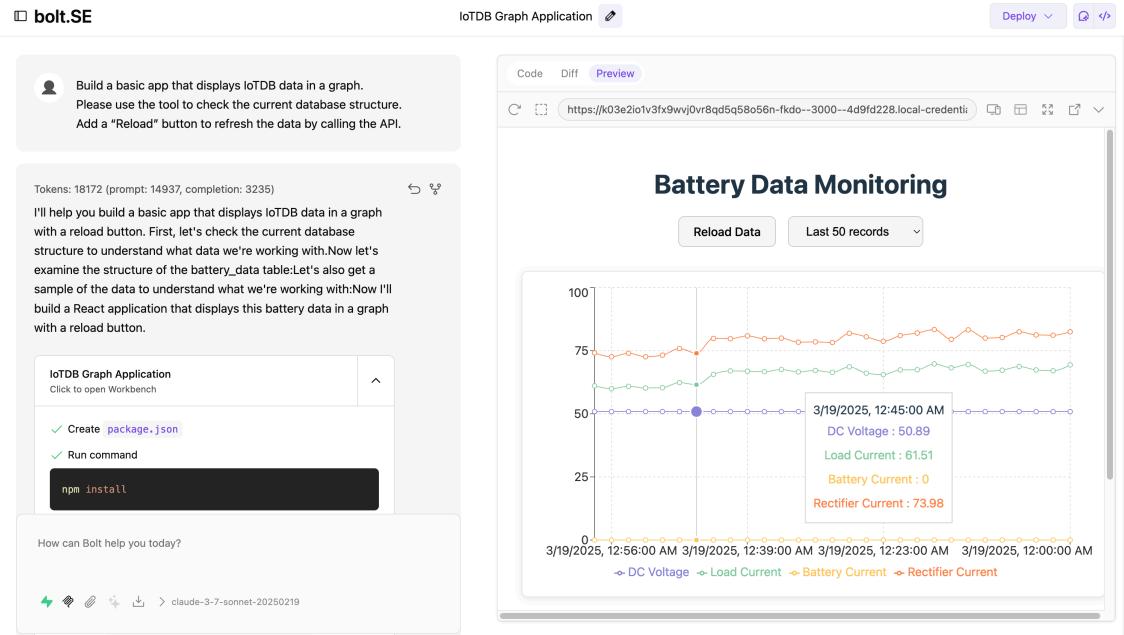


图 5.6 生成应用的 Workbench 预览，展示电池数据监测折线图与数据刷新功能

第 6 章 持续集成与持续部署（CI/CD）的设计与实现

在现代软件工程中，CI/CD（持续集成与持续部署）已成为保障开发效率与代码质量的核心机制。传统搭建 CI/CD 流水线通常依赖人工配置脚本与手动部署流程，不仅繁琐易错，且难以在多工具协同开发场景中实现统一管理。

bolt.SE 通过模型上下文协议（MCP）引入标准化的工具接口，使大语言模型（LLM）能够自动访问如 GitHub、GitLab 等远程仓库管理平台，执行如创建仓库、推送代码、触发流水线等操作。一旦接入了具备 CI/CD 能力的 MCP 服务器（例如 server-github 或 server-gitlab），系统即具备了构建自动化部署流程的基础。另一方面，bolt.SE 的测试驱动开发（TDD）模块提供从测试定义到代码验证的一体化支持，使 LLM 在开发早期即能生成具备行为保证的功能模块。二者结合，即形成测试定义、功能实现、代码验证、版本控制、自动部署的完整开发闭环。

本章将展示如何在 bolt.SE 中配置 GitLab MCP 工具，配合 TDD 流程构建 Todo 应用，并自动生成 CI/CD 配置，实现从本地测试到远程部署的全链路自动化流程。这一过程体现了 MCP 与 TDD 的协同机制如何系统性地提升 LLM 驱动开发的工程化能力。实验流程首先由 Jest 测试用例反向生成最小可行代码并即时验证（TDD），随后通过 MCP 连接 server-gitlab，在 GitLab 上新建仓库并推送代码，最后由 LLM 自动编写 .gitlab-ci.yml，为主干推送触发测试并部署到 Vercel。

6.1 环境准备与整体流程

如图 6.1 所示，开发者输入指令：Create a simple Todo app with minimal code. Then run the provided test file to verify functionality.

系统即开始驱动以下流程：

1. 解析 Jest 测试，用于约束 LLM 生成代码（TDD 模块）
2. 生成 React + Vite 工程、通过全部断言并预览界面
3. 连接 server-gitlab，验证令牌有效性（MCP 模块）
4. 生成 .gitlab-ci.yml、vercel.json 等 CI 配置（LLM）
5. 调用 create_repository 工具，在 GitLab 创建远端仓库（MCP）
6. 将本地文件推送至远端并触发首轮流水线

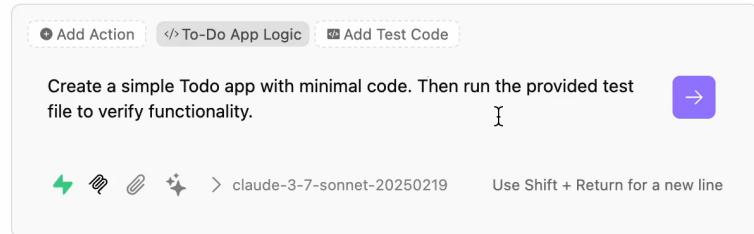


图 6.1 开发者输入指令

6.2 配置 GitLab MCP 服务器

图 6.2 展示了 `server-gitlab` 的 JSON 配置。配置过程首先通过 `npx -y @modelcontextprotocol/server-gitlab` 启动本地转接进程，传输类型设置为 `stdio`。在认证环节，系统使用 `GITLAB_PERSONAL_ACCESS_TOKEN` 环境变量完成 OAuth 认证，并仅暴露最小权限作用域 (`api_read_write`)。完成配置后，点击“Check availability”按钮进行连通性检测，状态变为“Available”即表示客户端已成功发现 `create_repository`、`list_projects` 等可用工具。

6.3 导入 Jest 测试并生成应用代码

开发者在 Add Test 模态框中粘贴 `to-do_app_logic.test.js` (图 6.3)。系统解析出四条断言：`adds task`、`rejects empty task`、`toggles status`、`deletes task`。随后，LLM 依据 TDD 原则生成最小实现；所有测试一次通过，且预览页展示可交互页面 (图 6.4)。

6.4 自动生成 GitLab CI 配置

LLM 接管流水线脚本编写，输出精简版 `.gitlab-ci.yml` (图 6.5，左侧工作卡片)。核心逻辑包括：测试阶段使用 `node:18` 镜像，执行 `npm ci && npm test`；部署阶段仅当分支为 `main` 且测试成功时调用 Vercel CLI 部署；同时缓存 `node_modules` 以加速后续执行。

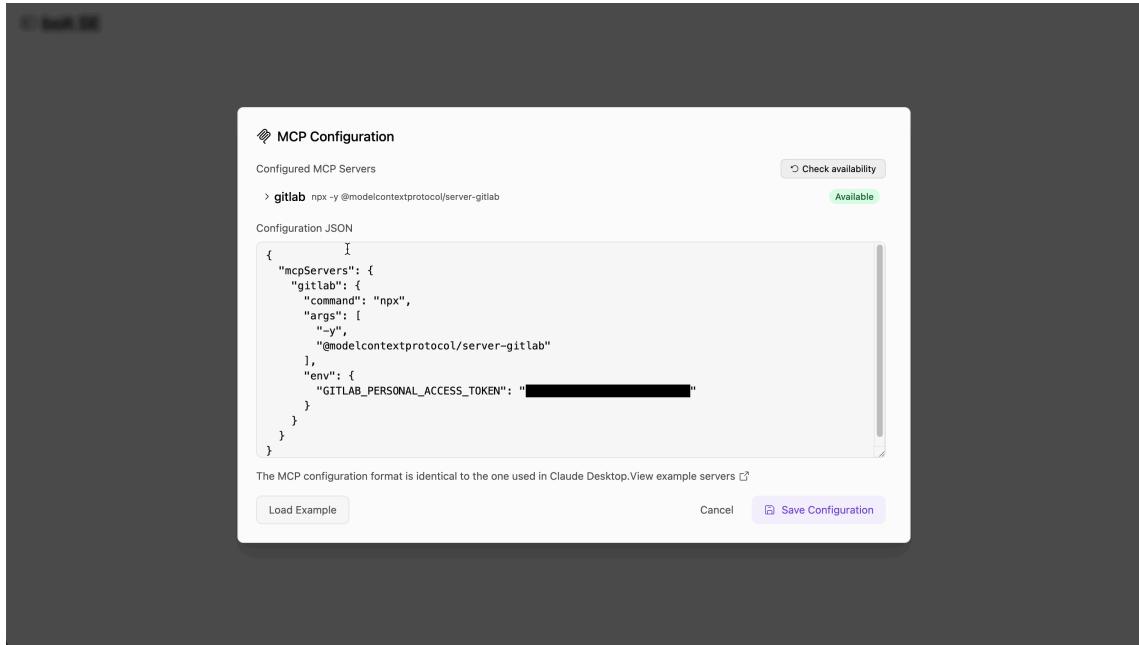


图 6.2 MCP 服务器 server-gitlab 配置与连通性检测

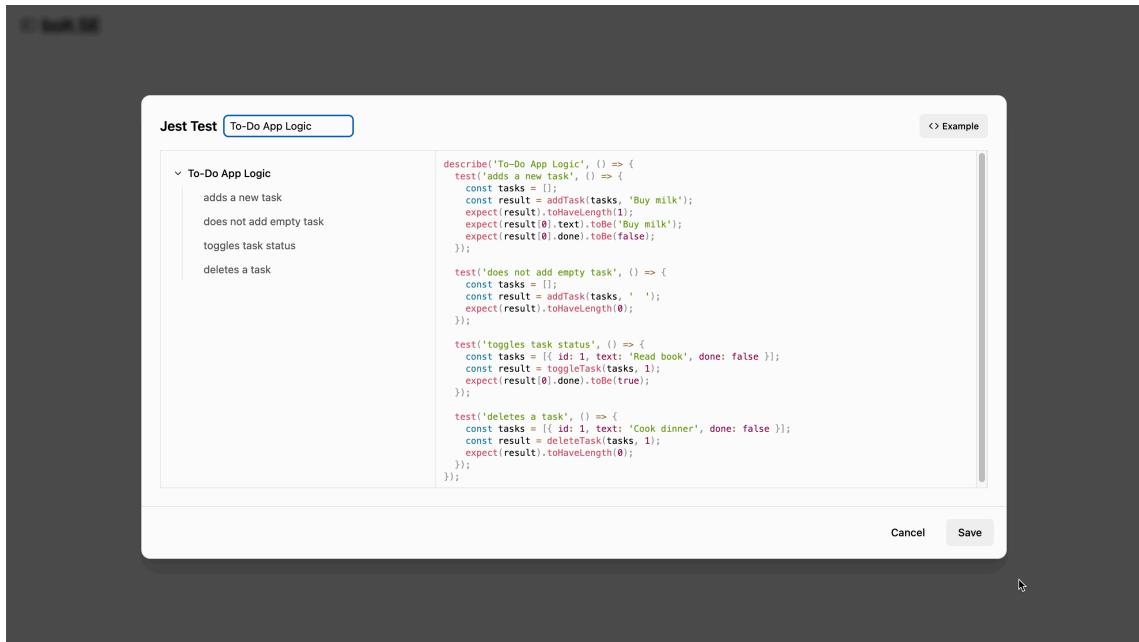


图 6.3 导入 Jest 测试文件并解析测试树

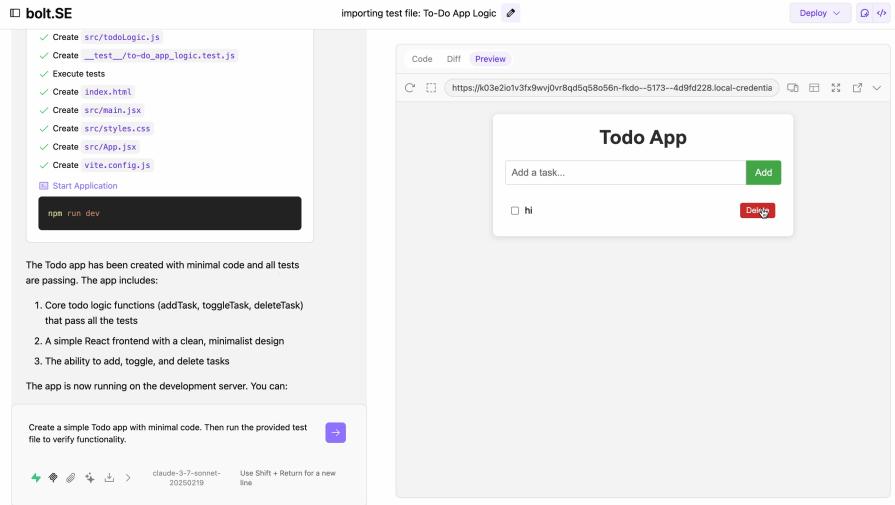


图 6.4 全部测试通过后的运行预览

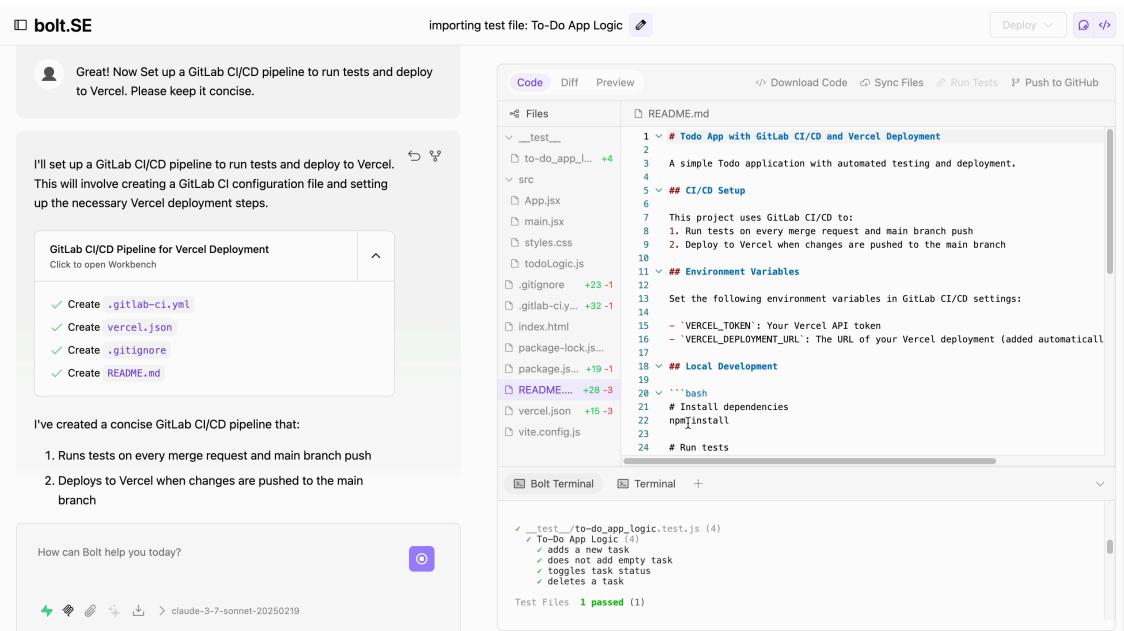


图 6.5 生成 CI/CD 计划卡片及 .gitlab-ci.yml 代码差异

6.5 仓库初始化与首轮流水线触发

系统调用 `create_repository` 工具（图 6.6），自动在 GitLab 新建 `todo-app` 仓库并返回 HTTPS 地址。紧接着执行 Git 操作：将本地 Vite 工程全部推送至远端，并在 CI 界面自动触发 `npm test` 与 Vercel 部署作业。

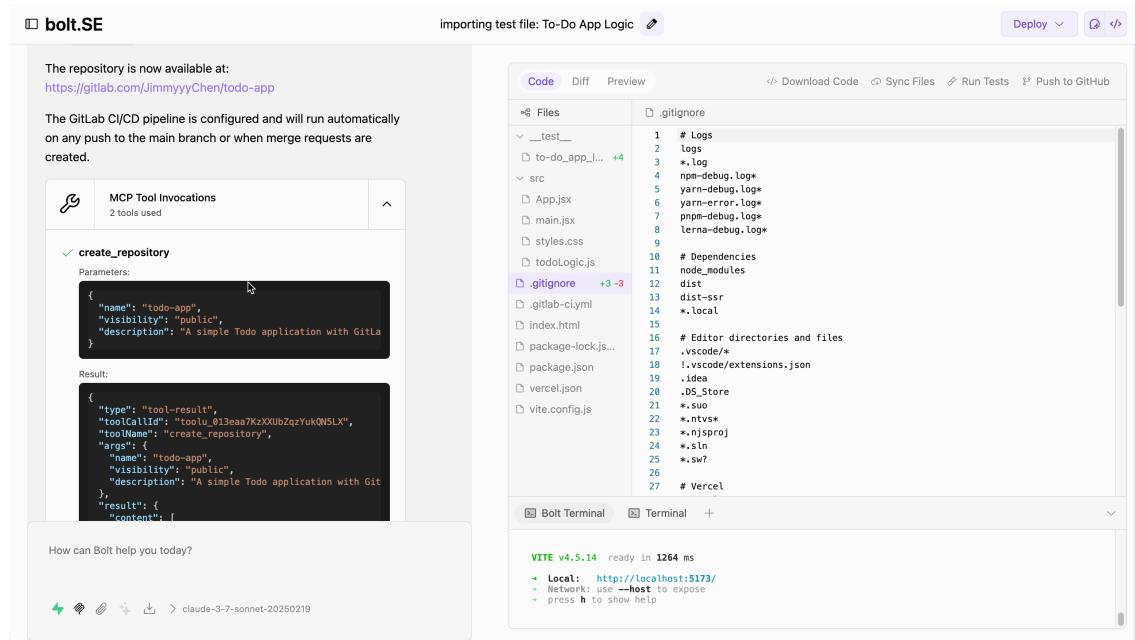


图 6.6 MCP 工具 `create_repository` 的调用参数与返回结果

GitLab 网页端界面（图 6.7）可见首个提交 *Add Todo app with GitLab CI/CD and Vercel deployment* 已完成；流水线正在运行，稍后即可在 Vercel 控制台查看部署链接。

通过 MCP 与 TDD 模块的融合，bolt.SE 构建了一条从代码生成到持续部署的完整自动化链路。MCP 提供了标准化的外部工具访问能力，使 LLM 能够直接操作 GitLab 等远程仓库平台；而 TDD 则确保生成代码符合预期行为且可验证。这种协同机制不仅简化了 DevOps 流程，更转变了传统 CI/CD 的构建方式——从手工配置脚本到由 LLM 理解测试需求并自动生成部署配置。该案例展示了 AI 驱动开发在软件交付过程实现了更高水平的自动化，提升了开发效率。

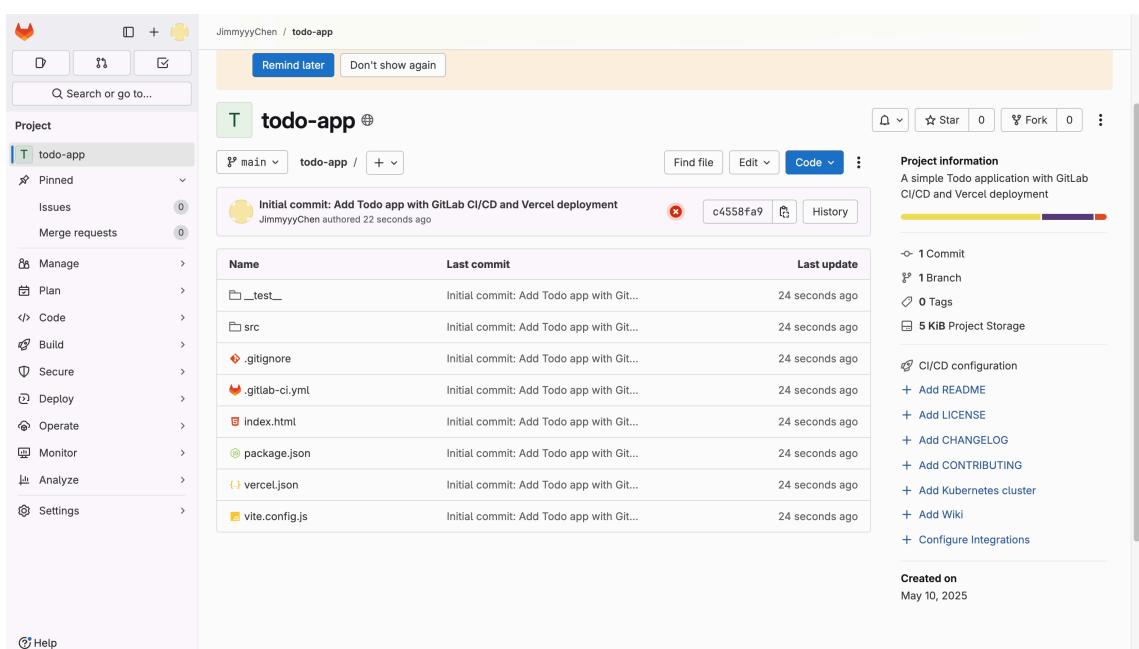


图 6.7 远端仓库初始化结果 (GitLab UI)

第 7 章 结论与展望

本研究深入探索大型语言模型（LLM）与现代软件工程最佳实践的协同融合，通过构建 bolt.SE 平台实现了从自然语言需求到可运行软件的自动化生成流程。在此过程中，系统性地解决了当前 LLM 辅助开发面临的上下文理解有限、反馈机制不完善及工程化流程缺失等核心挑战。本章总结研究成果，分析关键贡献，并展望未来发展方向。

7.1 主要研究成果

本研究的核心技术创新主要体现在以下四个方面：

1. API 优先开发模块：构建了基于 OpenAPI 规范的 API 定义与管理系统，使 LLM 能够理解、调用及生成符合 API 规范的代码。通过结构化 API 描述，系统实现了外部服务与 LLM 的无缝对接，扩展模型功能边界，减少上下文占用，提升多 API 协同效率。用户可通过直观界面添加、编辑 API 定义，系统支持 API 密钥、Bearer 令牌等多种认证方式。
2. 测试驱动开发（TDD）模块：将测试先行理念与 LLM 代码生成深度结合，通过结构化测试定义引导模型生成满足预定义行为的代码实现，建立了基于“红-绿-重构”循环的开发验证机制。用户在生成代码前先定义期望行为，系统将测试约束转化为 LLM 可理解的指导信息，确保生成代码符合需求并具备可验证性。
3. 模型上下文协议（MCP）模块：引入标准化接口协议，使 LLM 能够安全调用外部工具和数据源。系统通过模块化架构设计实现了多传输方式（stdio、HTTP SSE）和多服务类型的统一管理，支持从本地文件访问到远程服务调用的广泛场景。MCP 不仅突破了 LLM 知识截止限制，还显著降低了输出幻觉风险，为复杂问题解决提供了交互式支持。
4. CI/CD 自动化集成：通过 MCP 与 TDD 协同，实现了从测试定义到代码生成再到自动部署的全链路自动化流程。系统可以利用标准化工具接口（如 GitLab API）自动完成仓库创建、配置生成、代码推送与流水线触发等操作，形成“测试—代码—仓库—流水线—部署”的完整开发闭环，有效简化了传统 DevOps 流程中的手动操作环节。

这些技术创新共同构建了一个功能完备、流程闭环的软件开发环境，展现了 LLM 在规范化软件工程约束下的潜力。

本研究通过多个实例验证了 bolt.SE 的实用价值，这些应用案例验证了各模块的独立功能，为不同背景用户提供了从构思到部署的端到端解决方案。

7.2 与 bolt.diy 开源项目的连接

本研究与开源项目 bolt.diy 保持着密切的联系，形成了一个相互促进、持续迭代的创新循环。bolt.SE 以 bolt.diy 为基础开发，继承了其在 WebContainer 技术、多 LLM 支持和对话式代码生成等方面的核心优势，并在此基础上进行了深入的工程化拓展，尤其是在软件工程方法（如 TDD、API-first、MCP 集成）方面进行了创新与完善。

与此同时，本研究主动参与 bolt.diy 的社区建设，向其代码库提交了具体的贡献，包括引入 MCP 模块以及修复多个已知的缺陷，这些贡献已通过 PR 提交至社区（<https://github.com/stackblitz-labs/bolt.diy/pull/1704>），处于社区讨论与审阅阶段。此外，本研究设计与实现的 TDD 模块、APIActions 模块与 MCP 集成功能也将陆续以独立组件的形式回馈到 bolt.diy 项目中，以开源的方式提供给更广泛的开发者群体，推动整个生态系统的健康成长。

7.3 未来发展方向

bolt.SE 在编程教育领域具备广阔的应用潜力。系统能够作为个性化学习平台，帮助初学者以自然语言描述生成可运行代码并获得即时反馈，从而有效降低编程入门的门槛。内置的 TDD 与 API-First 等模块为软件工程课程提供了实践基础，使学生能够在真实开发环境中理解并应用现代开发方法，提升规范化开发能力。此外，bolt.SE 让教师实时了解学生的开发过程并给予有针对性的指导，同时也为课程项目提供了原型快速验证的能力，让学生能够专注于创新和设计。未来，系统计划与清华大学的课程结合，推动平台在实际教学中的应用与落地，结合课程内容提供场景化的编程辅助。

作为 bolt.SE 中的核心连接机制，MCP 拥有广阔的扩展空间。随着平台功能的持续迭代，未来我们将增强 MCP 多模态能力，支持图像识别、音频处理等工具的无缝集成，使 bolt.SE 能够处理更复杂的 UI 设计与多媒体应用场景。同时，将逐步构建针对医疗、金融、教育等领域的专用工具包，满足不同行业的软件开发需求。基于 bolt.SE 对 MCP 的创新实现，系统将进一步优化工具调用机制，实现更好的上下文感知的智能资源调度。此外，通过持续的社区贡献，bolt.SE 将不断完善 MCP 的标准化接口与模块化架构，推动整个 LLM 应用生态朝着更开放、更高

效的方向发展，使开发者能够轻松将多类专业功能集成到软件开发流程中。

7.4 总结

bolt.SE 将大语言模型的自然语言处理能力与软件工程的规范化实践深度融合，创建了一个从需求描述到可运行软件的端到端解决方案。通过 TDD、API-First、MCP 和 CI/CD 四大模块的协同工作，系统有效解决了当前 LLM 辅助开发面临的核心挑战，为构建更高效、更易用的软件开发环境提供了新思路。

未来 bolt.SE 将持续探索 LLM 在软件工程全生命周期中的深度应用，融入到实际使用场景，并通过与 bolt.diy 等开源项目的持续协作，推动整个智能软件开发生态系统的发展。

随着 LLM 技术与软件工程方法论的不断融合，像 bolt.SE 这样的系统将成为连接人类创意与代码实现的重要桥梁，使软件开发更加自然、高效且具有创造性。

参考文献

- [1] JIANG J, WANG F, SHEN J, et al. A survey on large language models for code generation [A/OL]. 2024. <https://arxiv.org/abs/2406.00515>.
- [2] CHEN L, GUO Q, JIA H, et al. A survey on evaluating large language models in code generation tasks[A/OL]. 2024. <https://arxiv.org/abs/2408.16498>.
- [3] GitHub Blog. Survey reveals ai's impact on the developer experience[EB/OL]. 2023. <https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/>.
- [4] Financial Times. Openai and start-ups race to generate code and transform software industry [J/OL]. Financial Times, 2025. <https://www.ft.com/content/8069b127-8589-4f06-9c38-8e0216c6fd9c>.
- [5] GitHub Blog. Github copilot x: The ai-powered developer experience[EB/OL]. 2023. <https://github.blog/news-insights/product-news/github-copilot-x-the-ai-powered-developer-experience/>.
- [6] Wikipedia contributors. Cursor (code editor)[EB/OL]. 2025. [https://en.wikipedia.org/wiki/Cursor_\(code_editor\)](https://en.wikipedia.org/wiki/Cursor_(code_editor)).
- [7] v0.dev[EB/OL]. 2024. <https://v0.dev/>.
- [8] Wikipedia contributors. Replit[EB/OL]. 2025. <https://en.wikipedia.org/wiki/Replit>.
- [9] The Times. Don't know how to code? build a serious business using ai[J/OL]. The Times, 2025. <https://www.thetimes.co.uk/article/dont-know-code-build-a-serious-business-using-a-i-enterprise-network-l2r2czlfv>.
- [10] Windsurf ai-enhanced ide[EB/OL]. 2024. <https://windsurf.com/>.
- [11] ARUNACHALAM A. Bolt.new: An ai tool that can build entire full stack applications directly in your browser[J/OL]. Medium, 2024. <https://ajay-arunachalam08.medium.com/bolt-new-an-ai-tool-that-can-build-entire-full-stack-applications-directly-in-your-browser-6e5822d142b3>.
- [12] StackBlitz. bolt.new: Prompt, run, edit, and deploy full-stack web and mobile apps[EB/OL]. 2024. <https://github.com/stackblitz/bolt.new>.
- [13] StackBlitz. Webcontainer api: Run node.js natively in your browser[EB/OL]. 2023. <https://webcontainers.io/>.
- [14] INITIATIVE O. Openapi specification[EB/OL]. 2023. <https://spec.openapis.org/oas/latest.html>.
- [15] Meta Open Source, OpenJS Foundation. Jest: Delightful javascript testing[EB/OL]. 2023. <https://jestjs.io/>.
- [16] Anthropic. Model context protocol specification[EB/OL]. 2023. <https://modelcontextprotocol.io/specification/>.
- [17] Apache IoTDB. Apache iotdb: Industrial iot time-series database management system[EB/OL]. 2025. <https://iotdb.apache.org/>.

- [18] Apache IoTDB. Apache iotdb mcp server: Natural language access with iotdb mcp[EB/OL]. 2025. <https://github.com/apache/iotdb-mcp-server>.
- [19] Supercorp AI. Supergateway: Run mcp stdio servers over sse and sse over stdio[EB/OL]. 2025. <https://github.com/supercorp-ai/supergateway>.
- [20] Apache IoTDB. Apache iotdb rest service v2[EB/OL]. 2025. <https://iotdb.apache.org/zh/UserGuide/latest/API/RestServiceV2.html>.

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	53
A.2 简介	53
A.3 开场白	56
A.3.1 大型语言模型	56
A.3.2 大型语言模型的类别	57
A.3.3 面向软件工程的大型语言模型	57
A.4 需求工程与设计	58
A.4.1 需求工程 LLMs 中的开放问题	58
A.5 代码生成和补全	58
A.5.1 代码生成模型	59
A.5.2 改进的 Prompt 工程代码生成	60
A.5.3 LLM 和其他技术的混合体	61
A.5.4 基于 LLM 的代码生成的科学评估	61
A.5.5 代码生成和补全中的开放性问题	64
A.6 软件测试	64
A.6.1 使用 llm 生成新测试	65
A.6.2 测试充分性评估	66
A.6.3 测试最小化	67
A.6.4 测试输出预测	67
A.6.5 测试不稳定性	67
A.6.6 用于软件测试的 LLMs 中的开放问题	67
A.7 维护、演进和部署	69
A.7.1 调试	69
A.7.2 程序修复	70
A.7.3 性能改进	71
A.7.4 克隆检测和复用	73

A.7.5 重构	73
A.7.6 维护和演化中的开放性问题	74
A.8 文档生成	74
A.8.1 文档生成和代码摘要的开放性问题	75
A.9 软件分析和存储库挖掘	75
A.10 人机交互	76
A.11 软件工程过程	76
A.12 软件工程教育	76
A.13 横切开放研究主题	77
A.13.1 为 SE 构建和调优 llm	77
A.13.2 需要动态自适应提示工程和参数调整	77
A.13.3 杂化	78
A.13.4 控制幻觉	78
A.13.5 稳健、可靠、稳定的评估	78
A.13.6 全面测试	79
A.13.7 处理较长的文本输入	79
A.13.8 软件工程覆盖较少的子领域	79
参考文献	79
书面翻译对应的原文索引	79

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 Böhme [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，Barei β 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“变异数”）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数。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 来生成能够忠实地捕获漏洞的突变体。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性问题。

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，这也可能有利于 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.

致 谢

感谢刘璘老师对本文的悉心指导，感谢您始终耐心聆听并鼓励我表达自己的想法，使我在研究过程中不断精进与创新。也非常感谢四年前您给予我转系至软件学院的机会，让我得以追求自己真正热爱的方向。

感谢黄舒炜学姐、马喆轩、刘一明、童新荷，以及组里的所有同学们，论文撰写过程中你们的相互支持与鼓励，让我获益良多。

感谢缪尚宸、李晓川和张之远，很幸运能够与你们合作完成令人骄傲的作品，你们积极进取的精神深深激励着我。也感谢你们给予我的鼓励与肯定，让我找到自己的优势并更加自信。

感谢赵沛阳、王香赫、杨余彤、胡智勇、岑奇航、史宇杰和谢铮，谢谢你们在127寝室创造了四年充满欢乐和回忆的日子，丰富了我的大学生活。特别感谢胡智勇，不仅经常拖我出门放风，还让我认识了我的女朋友，也祝你早日找到真命天女。

感谢张苡歆和戴嘉莹，没有你们，我无法想象自己如何度过那些充满压力的夜晚，谢谢你们总是听我倾诉。也感谢萧仰哲，你敢想敢做的精神一直鼓舞着我，让我更加勇于尝试新的挑战。

感谢我的女朋友丁一桐，谢谢你带给我的幸福与包容，你积极向上的态度深深影响了我，让我在学业上变得更加主动。对你的感谢难以言尽，留待日后慢慢诉说。

感谢我的父母，谢谢你们始终无条件地相信我，在学业上从不强求，让我能够自由地做自己喜欢的事情。正是你们的支持和信任，才让我走到今天。

最后，要感谢这些年来自己的自己。谢谢你在最低谷时依然选择相信自己，谢谢你敢于与众不同，谢谢你找回了自信。

声 明

本人郑重声明：所呈交的综合论文训练论文，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: _____