软件工程生命周期视角下的 AI 代码生成细粒度评测指标调研

> **版本**: v1.1（钉钉文档版） | **日期**: 2026-02-26 | **定位**: 面向 AI Coding 产品的代码评测指标体系，按软件工程生命周期组织

概览

本文从软件工程生命周期的四个阶段出发，系统梳理 AI 代码生成的细粒度评测指标。与按质量维度横向组织的指标体系不同，本文按需求、设计实现、测试、维护四个阶段纵向组织，更贴合实际开发流程和 SE 研究视角。每个指标包含定义、计算公式和核心论文支撑。

软件工程生命周期

v1.1 · 20

一 软件需求阶段

模型能否准确理解、分解和对齐用户需求?

1.1	SBC Score 语义对齐度	a
1.2	指令遵循率 IFR	BigCodeBe
1.3	需求分解与设计对齐	DesBench, a
1.4	需求歧义处理与澄清	a
1.5	跨模态需求理解	

三 软件测试阶段

AI 生成的测试代码质量如何? 产品中怎么评?

3.1	测试生成基础质量	TestGen-LL
3.2	代码覆盖率增益	ChatUniT
3.3	变异测试分数	MuTAP, a

一、软件需求阶段

> 核心问题：模型能否准确理解、分解和对齐用户需求？

AI 编码工具的输入起点是用户需求。需求阶段的评测关注模型对自然语言需求的理解能力、对显式约束的遵循程度、以及将高层需求分解为可执行子任务的能力。

1.1 需求-代码语义对齐度（SBC Score）

衡量 AI 生成代码对原始需求的忠实程度。通过逆向生成技术，将生成代码回传给 LLM 重建需求描述，再与原始需求对比。

方法论：给定需求 R → LLM 生成代码 C → 将 C 回传 LLM 重建需求 R' → 对比 R 与 R'

公式：SBC = 0.7 × semantic_score + 0.1 × BLEU + 0.2 × completeness

组件	权重	计算方式

语义相似度	0.7	使用 all-MiniLM-L6-v2 编码 R 和 R'，计算余弦相似度
BLEU	0.1	标准 n-gram 匹配，R 为参考、R' 为候选
完整性	0.2	提取关键词，计算 completeness = max(0, 1 - (|missing| + |extra|) / max(|keywords_R ∪ keywords_R'|, 1))

权重设计理由：语义相似度权重最高（0.7），直接衡量意图对齐；完整性（0.2）惩罚功能遗漏和幻觉；BLEU（0.1）可靠性较低但提供互补信号。

实验结果：在 90 个需求上评测 4 个模型（Codellama 13B、Qwen2.5-Coder 14B、Deepseek Coder 6.7B、Codestral 22B）。SBC > 0.55 表示逆向生成质量较高，SBC > 0.65 表示语义高度接近。四个模型的性能趋势高度一致，表明 SBC 对模型差异的区分度有限，但对需求覆盖缺口的诊断价值显著。

核心论文：Ponnusamy, 2025. arXiv:2502.07835

1.2 指令遵循率（Instruction Following Rate）

生成代码满足需求中显式约束的比例。

公式：IFR = 通过约束检查的数量 / 总约束数量

约束类型：语言指定、复杂度限制、禁用 API、输出格式要求、库调用要求等。

BigCodeBench 评测框架（Zhuo et al., ICLR 2025）：1,140 个细粒度任务，要求调用 139 个库、覆盖 7 个领域。每个任务平均 5.6 个测试用例，分支覆盖率 99%。BigCodeBench-Instruct 变体将详细 docstring 转化为简短自然语言指令。60 个 LLM 的评测结果显示最高得分约 60%，远低于人类的 97%。

核心论文：Zhuo et al., 2025. ICLR 2025. arXiv:2406.15877

1.3 需求分解与设计对齐能力

模型将高层用户需求分解为软件设计（类图、模块结构）并据此生成代码的能力。

DesBench 评测框架（He, Chen & Cao, 2026）：首个评测 LLM 软件设计能力的基准，包含 30 个手工构建的 Java 项目（24 个业务领域，194 个类，737 个测试用例）。评测三个任务：

任务	评测内容	核心指标
设计感知代码生成	不同设计信息粒度下的代码生成能力	Compilation@k、Pass@k、Match_class、Match_method
面向对象建模	从需求生成类图（PlantUML 格式）	类/属性/操作/继承/关联的 Precision、Recall、F1
验收测试设计	从功能需求生成验收测试用例	TCompilation%、TPass%、CCov/MCov/LCov

其中 Match = E_projects [#generated_items_matching_RJC / #items_in_RJC]

关键发现：LLM 能准确识别对象和类（F1 达 0.85），但在定义操作和类间关系时表现不佳；LLM 偏好低层设计（代码骨架）而非高层设计（类图）；仅提供需求时 Pass@1 显著下降。

核心论文：He, Chen & Cao, 2026. arXiv:2602.13611

1.4 需求歧义处理与澄清能力

面对模糊、不完整或矛盾的需求时，模型主动识别歧义并请求澄清的能力。

指标	定义
歧义检测率	模型正确识别需求中歧义点的比例
澄清问题质量	模型提出的澄清问题是否有助于消除歧义（可通过 LLM-as-Judge 评估）
兜底策略合理性	当无法获得澄清时，模型采用的默认行为是否合理

目前该方向缺乏标准化基准。"Requirements are All You Need"（Arora et al., arXiv:2406.10101, 2024）探索了 LLM 从需求文档直接生成代码的流程，但未专门评测歧义处理能力。

1.5 跨模态需求理解

模型从非纯文本输入（UI 截图、设计稿、流程图）理解需求并生成代码的能力。

评测维度	定义
视觉保真度	生成的 UI 代码与输入截图的视觉相似度
功能完整性	生成代码是否实现了截图中暗示的交互逻辑
布局准确性	元素位置、尺寸、层级关系的还原精度

相关基准：Design2Code（Si et al., 2024, arXiv:2403.03163）484 个真实网页，使用 CLIP 相似度和像素级比较评估；Screenshot-to-Code（arXiv:2602.05998, 2026）通过视觉差异学习改进生成质量；WebUIBench 21,793 个 QA 对、719 个真实网站。

二、软件设计与实现阶段

> 核心问题：生成的代码在功能、质量、效率、安全上是否达标?

设计与实现阶段是 AI 代码生成评测研究最密集的领域。本章覆盖功能正确性、工程质量、代码效率、安全性、许可证合规和泛化能力六个维度。

2.1 功能正确性

2.1.1 pass@k 系列

从 n 个生成样本中随机选取 k 个，至少有一个通过所有测试用例的概率。

无偏估计公式（Chen et al., 2021）：pass@k = E[1 - C(n-c, k) / C(n, k)]，其中 n 为总样本数，c 为通过测试的样本数。

变体	含义	适用场景
pass@1	一次做对的能力	最贴近实际使用
pass@5/10	多次尝试下的潜力上限	衡量模型能力天花板

局限性：二值判定（不区分部分正确）、测试依赖（EvalPlus 揭示 HumanEval 测试不足导致高估 10-20%）、不衡量代码质量。

核心论文：Chen et al., 2021. arXiv:2107.03374

2.1.2 pass@k 增强变体

变体	方法	核心发现
Enhanced-test pass@k（EvalPlus）	对原始 benchmark 测试用例自动增强（边界条件、类型变异）	HumanEval 上多数模型 pass@1 下降 10-20%
pass@t（多轮尝试）	考虑模型获得错误反馈后重试的能力	衡量"迭代修正"而非"一次生成"
自演进 pass@k（EvoCodeBench）	Agent 在 3 轮迭代预算内反思、修复、优化	自演进 agent 比单次生成 pass rate 提升 10.1%-26.7%

EvoCodeBench（Zhang et al., KDD 2026, arXiv:2602.10171）还引入人类参照指标：Average Runtime Beats (%) 和 Average Memory Beats (%)，衡量生成代码运行时间/内存使用超过多少比例的人类提交。

2.1.3 部分正确性与执行语义

- **部分正确性**：以通过的测试用例比例作为连续分数（如 10 个测试通过 7 个得 0.7），比 pass@k 的二值判定更细粒度
- **I/O 行为等价**（CRUXEval, Gu et al., 2024, arXiv:2401.03065）：将代码执行语义理解拆分为输入预测和输出预测两个子任务

2.2 工程质量

2.2.1 Lint 规范性

Linter 规则违反数 / KLOC（每千行代码）。对生成代码运行项目配置的 Linter（ESLint、Pylint、RuboCop 等），统计 warning + error 数量。

2.2.2 复杂度指标

指标	公式	说明
圈复杂度（McCabe, 1976）	V(G) = E - N + 2P	E: 控制流图边数, N: 节点数, P: 连通分量数。阈值：1-10 简单，11-20 中等，21-50 复杂
认知复杂度（SonarSource, 2018）	Σ (结构增量 + 当前嵌套层级)	三条规则：线性流中断 +1、嵌套惩罚 +nesting level、基础加分 +1 无嵌套惩罚

认知复杂度与圈复杂度的关键差异：switch 整体只计 +1 而非按 case 累加；else if 计 +1 但不受嵌套惩罚。示例：if 内嵌 for 再嵌 while，最内层 while 贡献 1(结构) + 2(嵌套) = 3。

2.2.3 可读性

方法	说明
Buse-Weimer 模型（IEEE TSE 2010）	基于 25 个结构特征训练逻辑回归模型，120 名标注者对 100 个 Java 片段评分。核心特征：行长度、标识符平均长度、最大缩进深度、注释密度、空行比例、字符熵等。输出 [0,1] 概率分数，准确率约 80%
LLM-as-Judge	使用强模型对生成代码的可读性进行 1-5 分评分。优势是能捕捉难以规则化的因素，局限是评分一致性和可复现性

2.2.4 代码异味密度

AI 生成代码中的代码异味数 / KLOC。AI 特有异味模式包括过度工程、冗余代码（未使用的 import/变量）、不必要的抽象。

实证数据：约 60.9% 的 AI 生成代码单元包含至少一个异味（ENIAC 2024）；LLM 代码异味平均增加 63.34%（arXiv:2510.03029, 2025）。

2.2.5 代码自然性

代码符合统计语言模型预期的程度，即代码的"可预测性"。理论基础：源代码比自然语言更具重复性和可预测性（Hindle et al., ICSE 2012）。

指标	公式	说明
交叉熵	$H(S) = -(1/N) \times \Sigma \log_2 P(t_i \mid t_1 \ldots t_{i-1})$	值越低，代码越"自然"。Java 代码约 2.5-3.5 bits/token
困惑度	$PP(S) = 2^{H(S)}$	交叉熵的指数形式
Cache n-gram	$P_{cache} = \lambda \cdot P_{global} + (1-\lambda) \cdot P_{local}$	结合全局和局部上下文（Tu et al., 2014）

评测意义：自然性可作为代码质量的辅助信号。LLM 生成代码天然具有高自然性，但统计上"自然"的代码可能语义不正确或过于泛化。

2.3 代码效率

指标	公式	来源	说明
NET（归一化执行时间）	$(1/N) \times \Sigma(T_{code} / T_{canonical})$	EffiBench, NeurIPS 2024	NET > 1 表示比最优解慢。GPT-4 平均 3.12x
NTMU（归一化总内存使用）	$(1/N) \times \Sigma(TMU_{code} / TMU_{canonical})$	EffiBench	通过对内存使用曲线 M(t) 在执行时间内积分计算
ET%（EffiBench-X）	$(1/N) \times \Sigma \text{ clip}(T_{human} / T_{llm}, 0, 1) \times 100\%$	EffiBench-X, 2025	扩展到 6 种语言，最优 LLM 约达人类效率的 62%
MI%（EffiBench-X）	$(1/N) \times \Sigma \text{ clip}(A_{human} / A_{llm}, 0, 1) \times 100\%$	EffiBench-X, 2025	内存效率百分比

NET 分级参考：$\leqslant 1.5$ 优秀，1.5-5 可接受，> 5 需优化，∞ 超时。

核心论文：Huang et al., 2024. arXiv:2402.02037；Qing et al., 2025. arXiv:2505.13004

2.4 安全性

2.4.1 漏洞引入率

漏洞引入率 = 含漏洞的生成样本数 / 总生成样本数 × 100%。使用 SAST 工具（Semgrep、Bandit、CodeQL）扫描生成代码，按 CWE 分类统计。

2.4.2 CyberSecEval 2 多维度安全评测

Bhatt et al., Meta, arXiv:2404.13161, 2024。四个核心安全指标：

指标	公式	实验数据
False Refusal Rate (FRR)	被错误拒绝的良性提示数 / 良性提示总数 × 100%	CodeLlama-70B 高达 70%，多数模型低于 15%
Prompt Injection Success Rate	成功注入数 / 注入测试总数 × 100%	覆盖 15 类注入方法，所有模型在 26%-41% 之间
Cyberattack Helpfulness	基于 MITRE ATT&CK 本体的 10 类攻击战术评估	—
Interpreter Abuse	覆盖 Container Escape、Privilege Escalation 等 5 类	每类 100 个测试用例

2.4.3 安全编码标准合规率

合规率 = (通过的规则检查数 / 适用规则总数) × 100%。可按严重级别加权：Critical 权重 3、Major 权重 2、Minor 权重 1。

覆盖标准：MISRA C:2023（175 条准则）、OWASP Top 10（2021 版）、CWE Top 25（2024 版）、CERT 安全编码标准。

2.5 许可证合规

AI 生成代码是否包含与受限许可证关联的代码片段。

工具/基准	说明	核心发现

LiCoEval（ICSE 2025, arXiv:2408.02487）	首个系统评测 LLM 许可证合规能力的基准	0.88%-2.01% 的生成代码与已有开源代码存在"显著相似性"，多数模型无法准确提供许可证信息
CodeGenLink（ASE 2025, arXiv:2510.01077）	VS Code 插件，结合 LLM 与 Web 搜索追溯代码来源	—
RepoMark（arXiv:2508.21432, 2025）	数据标记框架，通过语义等价代码变体审计训练数据	5% 误检率下检测成功率超 90%

2.6 跨语言与跨领域泛化

跨语言泛化比 = 长尾语言 pass@1 / 主流语言 pass@1

基准	覆盖范围
MultiPL-E（Cassano et al., 2023, arXiv:2208.08227）	将 HumanEval 翻译到 18+ 语言
HumanEval-X（Zheng et al., 2023）	覆盖 6 语言
SWE-bench Mobile（arXiv:2602.09540, 2026）	移动端跨领域
SimulatorCoder（arXiv:2602.17169, 2026）	仿真器领域

三、软件测试阶段

> 核心问题：AI 生成的测试代码质量如何？实际产品中测试生成功能怎么评？

测试阶段是现有评测体系的最大盲区。随着 Copilot /tests、Cursor test generation 等产品功能的普及，AI 生成测试代码的评测需求日益迫切。本章从产品实际出发，覆盖测试生成基础质量、覆盖率、变异测试、预言质量、回归安全和可维护性六个维度。

3.1 测试生成基础质量

AI 生成的测试代码能否编译、执行、并包含有效断言。

指标	定义
Build Rate（构建通过率）	生成的测试代码能否在现有基础设施中成功编译
Pass Rate（通过率）	构建成功的测试是否能通过执行（需重复执行排除 flaky test）
All Pass@1	生成的整个测试套件是否全部通过
Any Pass@1	生成的测试套件中是否有任意一个测试通过

TestGen-LLM 三级过滤器（Alshahwan et al., Meta, FSE 2024, arXiv:2402.09171）：Meta 在 Instagram 和 Facebook Android 应用上部署的工业级测试生成系统。过滤条件：Success = Builds ∧ Passes(5次) ∧ ImprovesCoverage

过滤阶段	通过率
至少一个测试成功构建	75%
至少一个测试构建并可靠通过	57%
至少一个测试构建、通过并提升覆盖率	25%
改进建议被工程师接受并部署	73%

TestGenEval 基准（Jain, Synnaeve & Roziere, ICLR 2025, arXiv:2410.00752）：首个文件级、真实项目的测试生成基准，基于 SWEBench 构建，11 个 Python 开源仓库、1,210 个代码-测试文件对。

模型	All Pass@1	Any Pass@1	Coverage	Mutation Score
GPT-4o	7.5%	64.0%	35.2%	18.8%
Llama 3.1 405B	17.7%	73.1%	35.0%	16.4%
Codestral 22B	26.8%	72.7%	33.0%	14.2%

关键发现：最佳模型在真实项目上的平均覆盖率仅 35.2%，与自包含小程序基准（接近 100%）形成鲜明对比。

3.2 代码覆盖率增益

AI 生成测试对已有测试套件的增量覆盖贡献。

指标	公式
行覆盖率	被执行的代码行数 / 总代码行数
分支覆盖率	被执行的分支数 / 总分支数
覆盖率增益	Coverage_after - Coverage_before

- **TestGen-LLM**：通过的测试是否能提升行覆盖率。论文指出行覆盖率只是权宜之计，未来应采用变异覆盖率
- **ChatUniTest**（Chen et al., FSE 2024, arXiv:2305.04764）：在半数评测项目上的行覆盖率超过 EvoSuite 和 TestSpark
- **CoverAgent**（CodiumAI）：以覆盖率增益为优化目标的迭代式测试生成工具

3.3 变异测试分数

AI 生成的测试用例检测注入缺陷（变异体）的能力。

公式：Mutation Score = #killed_mutants / #total_mutants

变异体通过对源代码施加变异算子生成（如将 > 改为 >=、删除条件分支、替换运算符等）。被测试检测到（测试失败）的变异体称为"被杀死"。

MuTAP（Moradi Dakhel et al., arXiv:2308.16557, 2023）：首个将变异测试与 LLM prompt 增强结合的方法。使用 MutPy 2.0 生成变异体，包含 11 种变异算子。

方法	Mutation Score	MS=100% 的 PUT 比例
Pynguin (DynaMOSA)	65.94%	28.22%
Codex zero-shot	76.82%	24.54%
llama-2-chat MuTAP	**93.57%**	**69.93%**

在 Refactory 真实缺陷数据集（1,710 个缺陷代码）上，MuTAP 检测率达 94.91%，比 Pynguin 多检测 28% 的缺陷。

核心发现：变异分数与真实缺陷检测能力的相关性远高于覆盖率，是更可靠的测试质量度量。

3.4 测试预言质量

AI 生成测试中断言的正确性和有效性。

指标	公式/定义
断言有效性	非空断言比例（排除仅有 assertTrue(true) 等无意义断言的测试）
误报率 (FPR)	正确代码上失败的测试数 / 总测试数
漏报率 (FNR)	错误代码上通过的测试数 / 总测试数
TPass%（DesBench）	参考正确代码能通过生成测试的概率。越低说明预言质量越差

断言消息质量（arXiv:2509.19673, 2025）：Codestral-22B 达到 2.76/5 分，人工编写为 3.24/5 分。

3.5 回归测试安全性

代码修改后，原本通过的测试仍然通过的比例。

公式：P2P 保持率 = P2P_preserved / P2P_total × 100%

P2P（Pass-to-Pass）：修改前通过且修改后仍通过的测试。在 SWE-bench 评测中，Resolved Rate 要求 P2P 全部保持，这是非常严格的标准。

3.6 测试可维护性

AI 生成测试代码的长期可维护质量。

常见测试异味：重复断言（Assertion Roulette）、魔法数字（Magic Number Test）、过度 Mock（Excessive Mocking）、单个测试方法中包含过多测试用例。

CleanTest（FSE 2025）：评测测试代码的整洁度，包括命名规范、结构清晰度、断言可读性。

3.7 产品级测试生成评测 Pipeline

工业界常用的端到端评测流程：生成 → 编译检查 → 执行验证 → 覆盖率计算 → 变异分数 → 人工审核

评测维度	说明	参考数据
采纳率	工程师接受 AI 生成测试的比例	TestGen-LLM 报告 73%
覆盖率增益	添加 AI 测试后的覆盖率提升	—
Flaky Rate	AI 生成测试的不稳定率（需多次执行验证）	—
维护成本	AI 测试在代码演化后的失效率	—

四、软件维护阶段

> 核心问题：AI 在代码修复、重构、审查、演化中的表现如何评测？

4.1 Bug 定位准确率

AI 在给定 bug 报告或失败测试后，定位缺陷代码位置的精度。

公式：Top-N 准确率 = 正确定位在前 N 个候选中的比例。按粒度分为文件级、函数级、行级定位。

实证数据（Chen & Jiang, SANER 2025, arXiv:2410.12468）：分析 10 个顶级 Agent 在 SWE-Bench Verified 的 500 个 issue 上的表现：

粒度	对齐度	说明
文件级	94.59% F1=1	FactoryCodeDroid 最佳
函数级	24.32%	骤降，说明 Agent 能定位正确文件但常修改错误函数

4.2 补丁质量

AI 生成补丁的精确性和最小性。

指标	定义

补丁最小性	变更行数 vs 金标准补丁行数
过度修复率	Agent 修改了不必要的代码的比例
Plausible vs Correct	通过测试的补丁中真正正确的比例

实证数据（SANER 2025）：过度修改是主要失败模式，HoneyComb 平均 47.54 行 vs 金标准 7.71 行；精简补丁 Gru (5.67 行) 和 Amazon-Q-Dev (6.08 行) 比金标准更精简。330/500 个 issue 被解决，仅 5.8% 被所有 10 个 Agent 同时解决。

4.3 代码可维护性指标

指标	公式	说明
万行代码缺陷率	(静态分析缺陷 + 动态测试缺陷) / (代码总行数 / 10000)	成熟项目基线 0.1-1.0 缺陷/KLOC，AI 代码逻辑错误率约高出人写代码 30%
可维护性指数 (MI)	171 - 5.2×ln(V) - 0.23×G - 16.2×ln(LOC)	V: Halstead Volume, G: 圈复杂度。MI > 85 高，65-85 中，< 65 低

LLM 交叉熵作为可维护性预测器（arXiv:2401.12714, 2024）：控制 LLOC 后，LLM 计算的交叉熵是类级可维护性的有效预测指标（交叉熵越高，可维护性越低）。

4.4 代码审查辅助质量

LLM 辅助代码审查的意见质量。

基准	规模	核心指标	关键发现
AACR-Bench（arXiv:2601.19494, 2026）	80 名高级工程师，10 种语言，200 个 PR，1,505 条审查意见	Precision / Recall / F1	Agent 方法 Precision 高（Claude-4.5-Sonnet 达 39.90%）但 Recall 低（10.10%），最佳 F1 约 14-16%
SWR-Bench（arXiv:2509.01494, 2025）	1,000 个人工验证的 GitHub PR	Hit metric	与人类判断约 90% 一致

4.5 重构质量

AI 辅助代码重构的行为保持性和质量改善度。

指标	定义
行为保持性	重构前后所有测试仍然通过
质量改善度	(Metric_before - Metric_after) / Metric_before × 100%

TestRefactoring（FSE 2025）：评测 AI 辅助测试代码重构的质量。

4.6 代码演化与兼容性

AI 生成代码在项目演化过程中的兼容性保持。

State-Diff 评测（Agent-Diff, Pysklo et al., arXiv:2602.11224, 2026）：将服务建模为状态机，在 Agent 执行前后快照数据库，计算状态差异：

差异类型	公式
新增实体	Delta_add = T_after \ T_before
删除实体	Delta_del = T_before \ T_after
修改实体	Delta_mod = {(e, e') : e.pk = e'.pk ∧ e ≠ e'}

关键设计：闭世界假设，状态差异中任何未被断言覆盖的变更视为副作用，任务得分归零。覆盖 224 个任务、4 个企业服务、108 个 API 端点。

五、跨阶段通用指标

> 贯穿整个生命周期的评测维度

5.1 输出一致性与稳定性

对同一 prompt 生成 m 次代码 $\{c_1, \ldots, c_m\}$，测试用例集 $T = \{t_1, \ldots, t_n\}$：

指标	公式	说明
功能一致性 (FC)	$FC = (1/n) \times \Sigma_j \max(\Sigma_i R_{ij}, m - \Sigma_i R_{ij}) / m$	FC = 1 完全一致，FC ≈ 0.5 高度随机
Fleiss' κ	κ > 0.8 高一致性	等价度量

即使 temperature=0，ChatGPT 仍表现出显著非确定性（Ouyang et al., arXiv:2308.02828）。top-p 对输出变异性的影响大于 temperature（Donato et al., arXiv:2502.17450, 2025）。

5.2 数据污染鲁棒性

评测方法：时间戳过滤（LiveCodeBench）、数据去重、动态生成。工业实践中使用"黄金数据集"并以 15% 的周更新率动态刷新。

5.3 生成成本与交互效率

指标	定义
Cost-per-Task	token 消耗 × 单价
Token-to-Solution Efficiency	解决同一问题所需的 token 数
步骤效率	Agent 完成任务的交互轮数
Wall-clock Time	从提交到完成的实际耗时

成本-质量帕累托前沿：在给定成本预算下能达到的最高质量（arXiv:2602.06370, 2026）。

5.4 价值观对齐

生成代码是否遵循安全策略：拒绝生成恶意代码、检测硬编码密钥和敏感信息泄露。

参考文献

需求阶段

1. Ponnusamy, 2025. "Bridging LLM-Generated Code and Requirements: Reverse Generation Technique and SBC Metric". arXiv:2502.07835
2. Zhuo et al., 2025. "BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions". ICLR 2025. arXiv:2406.15877
3. He, Chen & Cao, 2026. "From What to How: Bridging User Requirements with Software Development Using Large Language Models". arXiv:2602.13611
4. Arora et al., 2024. "Requirements are All You Need: From Requirements to Code with LLMs". arXiv:2406.10101
5. Si et al., 2024. "Design2Code: How Far Are We From Automating Front-End Engineering?". arXiv:2403.03163

设计与实现阶段

6. Chen et al., 2021. "Evaluating Large Language Models Trained on Code" (Codex/HumanEval). arXiv:2107.03374
7. Liu et al., 2024. "Is Your Code Generated by ChatGPT Really Correct?" (EvalPlus). arXiv:2305.01210
8. Zhang et al., 2026. "EvoCodeBench: Self-Evolving LLM-Driven Coding Systems". KDD 2026. arXiv:2602.10171
9. Gu et al., 2024. "CRUXEval: Code Reasoning, Understanding and Execution". arXiv:2401.03065
10. Campbell, 2018. "Cognitive Complexity: An Overview and Evaluation". SonarSource
11. Buse & Weimer, 2010. "Learning a Metric for Code Readability". IEEE TSE, 36(4):546-558
12. Hindle et al., 2012. "On the Naturalness of Software". ICSE 2012
13. Cotroneo et al., 2025. "Human-Written vs. AI-Generated Code" (ISSRE 2025). arXiv:2508.21634
14. Huang et al., 2024. "EffiBench: Benchmarking the Efficiency of Automatically Generated Code". NeurIPS 2024. arXiv:2402.02037
15. Qing et al., 2025. "EffiBench-X: Multi-Language Efficiency Benchmark". arXiv:2505.13004
16. Bhatt et al., 2024. "CyberSecEval 2" (Meta). arXiv:2404.13161
17. Xu et al., 2025. "LiCoEval: Evaluating LLMs on License Compliance" (ICSE 2025). arXiv:2408.02487
18. Bifolco et al., 2025. "CodeGenLink" (ASE 2025). arXiv:2510.01077
19. Qu et al., 2025. "RepoMark". arXiv:2508.21432
20. Cassano et al., 2023. "MultiPL-E". arXiv:2208.08227
21. "Investigating The Smells of LLM Generated Code", 2025. arXiv:2510.03029

测试阶段

22. Alshahwan et al., 2024. "Automated Unit Test Improvement using Large Language Models at Meta" (TestGen-LLM). FSE 2024. arXiv:2402.09171
23. Jain, Synnaeve & Roziere, 2025. "TestGenEval: A Real World Unit Test Generation and Test Completion Benchmark". ICLR 2025. arXiv:2410.00752
24. Moradi Dakhel et al., 2023. "Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing" (MuTAP). arXiv:2308.16557
25. Chen et al., 2024. "ChatUniTest: A Framework for LLM-Based Test Generation". FSE 2024. arXiv:2305.04764
26. "Efficient LLM-driven Test Generation Using Mutant Prioritization" (PRIMG), 2025. arXiv:2505.05584

维护阶段

27. Chen & Jiang, 2025. "Evaluating Software Development Agents: Patch Patterns, Code Quality, and Issue Complexity" (SANER 2025). arXiv:2410.12468
28. Pysklo et al., 2026. "Agent-Diff: State-Diff-Based Evaluation". arXiv:2602.11224
29. "AACR-Bench: Evaluating Automatic Code Review with Holistic Repository-Level Context", 2026. arXiv:2601.19494
30. "SWR-Bench: Benchmarking and Studying the LLM-based Code Review", 2025. arXiv:2509.01494
31. "Evaluation of large language models for assessing code maintainability", 2024. arXiv:2401.12714

跨阶段

32. Ouyang et al., 2023. "An Empirical Study of the Non-determinism of ChatGPT in Code Generation". arXiv:2308.02828
33. Donato et al., 2025. "Studying How Configurations Impact Code Generation in LLMs". arXiv:2502.17450
34. Jain et al., 2024. "LiveCodeBench: Holistic and Contamination Free Evaluation". arXiv:2403.07974
35. "Cost-Aware Model Selection", 2026. arXiv:2602.06370

附录：指标-阶段映射表

指标	需求	设计实现	测试	维护

SBC Score	●			
指令遵循率	●			
需求分解能力	●			
pass@k		●		
部分正确性		●		
Lint 规范性		●		●
圈复杂度		●		●
认知复杂度		●		●
可读性		●	●	
代码异味密度		●		●
代码自然性		●		
NET/NTMU		●		
漏洞引入率		●		
安全标准合规率		●		
许可证合规		●		
跨语言泛化		●		
Build/Pass Rate			●	
代码覆盖率			●	
变异测试分数			●	
测试预言质量			●	
P2P 保持率			●	●
测试可维护性			●	
Bug 定位准确率				●
补丁最小性				●
可维护性指数		●		●
代码审查质量				●
重构质量				●
State-Diff				●
输出一致性	●	●	●	●
数据污染鲁棒性		●		
生成成本	●	●	●	●

报告生成日期: 2026 年 2 月 26 日 **版本**: v1.1（钉钉文档版） **适用范围**: AI Coding 产品评测、学术研究评测体系设计、企业内部 AI 编程工具选型