

AI 生成代码评测细粒度指标调研

版本: v2.2 | 日期: 2026-02-24 | 定位: 面向 AI Coding 产品 (AI IDE、代码助手、Coding Agent) 的代码类评测指标体系梳理

一、功能正确性 (Functional Correctness)

核心问题: 生成的代码能不能正确运行、实现预期功能?

1.1 pass@k 系列

pass@k 由 Chen et al. (2021) 在 HumanEval 中提出, 无偏估计公式为 $\text{pass}@k = E[1 - C(n-c, k) / C(n, k)]$, 其中 n 为总生成样本数, c 为通过测试的样本数。pass@1 衡量"一次做对"的能力, pass@5/10 衡量多次尝试下的潜力上限。主要局限在于二值判定 (不区分部分正确)、测试依赖 (EvalPlus 揭示 HumanEval 测试不足导致高估) 以及不衡量代码质量。

1.2 pass@k 增强变体

Enhanced-test pass@k (EvalPlus): 对原始 benchmark 的测试用例进行自动增强 (增加边界条件、类型变异等), 然后重新评估 pass@k。EvalPlus 发现 HumanEval 上许多模型的 pass@1 在增强测试后下降 10-20%。

pass@t (多轮尝试): 考虑模型在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): EvoCodeBench (Zhang et al., 2602.10171, Feb 2026) 评测 LLM 驱动的自演进编码系统。其核心机制是: Agent 从初始代码草稿出发, 在固定 3 轮迭代预算内反思当前方案、识别失败模式 (未处理的边界条件、逻辑错误、次优复杂度), 提出针对性修改并重新提交在线评测。评测基于 LeetCode 的 3,822 道题目, 覆盖 Python、C++、Java、Go、Kotlin 五种语言。除 pass rate 外, EvoCodeBench 引入了人类参照指标: Average Runtime Beats (%) 和 Average Memory Beats (%), 即生成代码在运行时间和内存使用上超过

多少比例的人类提交。这使得即使 pass rate 饱和，效率维度仍能区分模型能力。实验结果显示 GPT-5.2 在多数语言上 pass rate 达 84-91%，但 Claude-4.5-sonnet 的 TLE 率高达 36-42%，表明功能正确但效率不足；DeepSeek-v3.2 在长尾语言 Kotlin 上 pass rate 骤降至 7%，揭示了跨语言泛化的脆弱性。

1.3 执行正确性与部分正确性

I/O 行为等价：CRUXEval (Gu et al., 2024) 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，评测模型对代码运行行为的推理能力。**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，但尚未被主流 benchmark 广泛采用。

二、工程质量 (Engineering Quality)

核心问题：代码不仅要“能用”，还要“好用”，符合工程规范。

2.1 Lint 规范性

定义：Linter 规则违反数 / KLOC（每千行代码）。

计算方式：对生成代码运行项目配置的 Linter (ESLint、Pylint、RuboCop 等)，统计 warning + error 数量，除以代码行数（千行）。

适用场景：所有代码生成任务，尤其是仓库级生成和代码补全。

2.2 复杂度指标

圈复杂度 (Cyclomatic Complexity)：

- 公式： $V(G) = E - N + 2P$ (E: 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，>50 不可测试

认知复杂度 (Cognitive Complexity)：SonarSource 提出 (Campbell, 2018)，基于三条核心规则计算：

1. **结构性递增 (+1)**: 对打断线性流程的控制结构 (`if`、`for`、`while`、`catch`、`switch`、三元运算符) 各计 +1, 且这些结构会增加嵌套层级
2. **嵌套惩罚 (+nesting level)**: 当上述结构嵌套在其他控制结构内部时, 每层嵌套额外 +1。例如 `if` 内嵌 `for` 再嵌 `while`, 最内层的 `while` 贡献 1(结构) + 2(嵌套) = 3
3. **基础递增 (+1, 无嵌套惩罚)**: `break LABEL`、`continue LABEL`、`goto` 等跳转语句, 以及逻辑运算符序列中每次运算符类型切换 (如 `a && b || c` 计 +2, 而 `a && b && c` 仅计 +1)

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; 递归调用额外 +1; 方法入口不计数。

AI 生成代码的复杂度实证研究: "Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity" (Cotroneo, Improta & Liguori, 2508.21634, ISSRE 2025) 对超过 50 万个 Python 和 Java 代码样本进行分析, 对比 ChatGPT、DeepSeek-Coder、Qwen-Coder 与人写代码。主要发现: AI 生成代码整体更简单、更具重复性, 但更容易出现未使用的构造和硬编码调试残留; 人写代码结构复杂度更高, 可维护性问题更集中; AI 生成代码包含更多高风险安全漏洞。两类代码呈现出截然不同的缺陷画像。

2.3 可读性

命名语义准确性: 变量名/函数名是否准确反映其语义。可通过 LLM-based 评估 (Siddiq et al., 2024) 或基于规则的检查 (如 Python 的 PEP 8 命名规范) 量化。

嵌套深度: 建议 ≤ 4 层。AI 生成代码常出现过深嵌套, 尤其在处理复杂条件逻辑时。

LLM-as-Judge 可读性评估: 使用强模型 (如 GPT-4、Claude) 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可读性因素, 局限是评分一致性和可复现性。

2.4 代码异味密度

定义: AI 生成代码中的代码异味数 / KLOC。

AI 特有异味模式:

- 过度工程 (Over-engineering): 为简单问题生成过于复杂的解决方案
- 冗余代码: 不必要的变量声明、未使用的 import
- 不必要的抽象: 为单次使用的逻辑创建类或函数

实证研究:

- "AI builds, We Analyze" (2601.16839, Jan 2026): 对 AI 生成的构建代码进行质量分析
- "Comparing Human and LLM Generated Code" (2501.16857, Jan 2025): 人写 vs AI 生成代码的异味对比

2.5 代码重复率

两个维度:

- **内部重复**: 生成代码自身内部的重复片段比例
- **外部重复**: 生成代码与已有代码库中代码的重复比例 (可能暗示模型在"复制"而非"理解")

实证研究: Cotroneo et al. (2508.21634, ISSRE 2025) 在 50 万+ 样本上确认 AI 生成代码"更具重复性"。"Investigating The Smells of LLM Generated Code" (2510.03029, Oct 2025) 对比 Gemini Pro、ChatGPT、Codex、Falcon 生成的 Java 代码与专业参考实现, 发现 LLM 生成代码的代码异味平均增加 63.34%, 其中 73.35% 为实现异味 (含重复代码), 且任务复杂度越高异味增幅越大。"On Inter-dataset Code Duplication and Data Leakage in Large Language Models" (2401.07930, Jan 2024) 从数据集层面研究了 LLM 训练/评测数据间的代码重复与数据泄漏问题。

2.6 风格一致性

定义: 生成代码与项目现有代码风格的偏离度。

量化方式与研究支撑:

- **编码标准符合度**: Licorish et al. (2501.16857, Jan 2025) 使用 Pylint 评分衡量 Python 代码对 PEP 8 标准的遵循程度, 发现人写代码的编码标准评分高于 GPT-4 生成代码
- **Halstead 指标一致性**: Clark et al. (2024, IEEE ICoSSE) 使用 7 项 Halstead 复杂度指标评估 625 个 ChatGPT 生成的 Python 代码样本的质量一致性, 发现 ChatGPT 在迭代提示和不同版本间保持了较稳定的质量水平
- ****异味倾向评分 (Propensity Smelly Score, PSC)****: (2511.15817, Nov 2025) 提出的概率性指标, 估计模型生成特定类型代码异味的倾向。研究发现 prompt 设计和模型架构选择对风格质量起决定性作用
- **多维工程质量评估**: RTL-Bench (Fang et al., 2025, ICCD) 提出覆盖语法正确性、功能性、Lint 合规、可读性和风格一致性的多维评测框架, 采用 LLM-as-Judge 机制评估主观质量维度

三、稳定性与鲁棒性 (Stability & Robustness)

核心问题：大规模使用时，生成代码的质量是否稳定可靠？

3.1 万行代码缺陷率 (Defects per 10K LoC)

定义：每万行生成代码中的缺陷数。

计算：(静态分析缺陷 + 动态测试缺陷) / (生成代码总行数 / 10000)

工业意义：这是衡量 AI 代码生成在大规模场景下可靠性的核心指标。单函数的 pass@k 可能很高，但当模型连续生成数千行代码时，累积缺陷率才是真正的风险度量。模拟了大规模代码库合并时的质量风险。

参考基线：传统软件工程中，成熟项目的缺陷密度约 0.1-1.0 缺陷/KLOC（生产环境）。AI 生成代码的这一指标目前缺乏系统性基准。

3.2 回归安全性 (P2P 保持率)

定义：修复/修改后，原本通过的测试仍然通过的比例。

计算： $P2P_{preserved} / P2P_{total} \times 100\%$

衡量：生成代码是否在解决一个问题的同时引入了新问题。在 SWE-bench 评测中，Resolved Rate 要求 P2P 全部保持，这是一个非常严格的标准。

3.3 输出一致性 (Consistency)

定义：相同 prompt 下多次生成结果在功能和文本层面的稳定程度。

计算方式：

- 功能一致性：多次生成的代码是否都通过相同的测试用例集合
- 文本一致性：多次生成结果之间的编辑相似度均值
- 方差度量：pass@k 在多次独立实验中的标准差

解码参数的影响：输出一致性与采样参数（temperature、top-p）密切相关。Ouyang et al. (2308.02828) 对 829 个代码生成问题的实证研究发现，即使在 temperature=0 的确定性设置下，ChatGPT 仍表现出显著的非确定性行为，这源于 GPU 浮点运算的非确定性和 API 实现细节。Donato et al. (2502.17450, Feb 2025) 进一步发现 top-p 对输出变异性的影响大于 temperature，并量化了为控制非确定性所需的最小重复实验次数。这意味着评测 pass@k 时，若不控制采样参数并进行多次重复，结论的可复现性存疑。

意义：对于生产环境部署，一致性比峰值性能更重要。高一致性意味着模型输出可预测，低一致性意味着“运气成分”大。

3.4 数据污染鲁棒性

定义：在排除训练数据泄漏后的真实性能。

评测方法：

- 时间戳过滤：只使用模型训练截止日期之后的数据 [LiveCodeBench, Jain et al., 2024]
- 数据去重：检测评测集与训练集的重叠
- 动态生成：评测集持续更新，避免被“刷榜”

工业实践：头部产品使用“黄金数据集”（Golden Dataset），排除所有可能的污染源，并以 15% 的周更新率动态刷新评测集。

四、安全性与合规 (Security & Compliance)

核心问题：生成的代码是否安全、合规、可信？

4.1 漏洞引入率

定义：生成代码中包含已知漏洞模式（CWE 分类）的样本比例。

计算：含漏洞的生成样本数 / 总生成样本数 × 100%

评测方法：使用 SAST 工具（Semgrep、Bandit、CodeQL）扫描生成代码，按 CWE 分类统计。区分被动评测（给定任务后检查输出）和主动评测（AI 在补全/建议场景中是否主动引入不安

全模式， CyberSecEval 2 的新维度)。

参考基准：

- CyberSecEval 2 (Bhatt et al., 2024, Meta): 多维度安全评测
- SecurityEval (Siddiq & Santos, 2022): 安全编码基准数据集

4.2 Prompt Injection Resilience (抗提示词注入能力)

定义：在对抗性提示注入（如恶意代码注释、误导性 docstring）下，模型仍能生成安全代码的能力。

最新研究：

- "Can Adversarial Code Comments Fool AI Security Reviewers" (2602.16741, Feb 2026): 通过注释注入攻击 LLM 代码分析
- "Prompt Poisoning Code" (2510.22944, Oct 2025): 研究提示词投毒对代码缺陷引入率的影响

4.3 安全编码标准合规率

定义：生成代码符合特定安全编码标准的比例。

标准覆盖：MISRA（嵌入式）、OWASP Top 10（Web）、CWE Top 25（通用）。

最新进展：

- CVE-Factory (2602.03012, Feb 2026): 将安全漏洞发现做成 Agent 级任务
- "Security and Quality in LLM-Generated Code: Multi-Language Analysis" (2502.01853, Feb 2025)

4.4 许可证合规与价值观对齐 (License Compliance & Constitutional Compliance)

许可证合规：AI 生成代码可能包含与受限许可证（如 GPL、AGPL）关联的代码片段，引发法律风险。

评测基准与工具：

- **LiCoEval** (Xu et al., 2408.02487, ICSE 2025): 首个系统评测 LLM 许可证合规能力的基准。评测 14 个主流 LLM，发现 0.88%-2.01% 的生成代码与已有开源代码存在"显著相似性" (striking similarity)，且多数模型无法准确提供许可证信息，尤其对 copyleft 许可证的识别能力薄弱
- **CodeGenLink** (Bifolco et al., 2510.01077, ASE 2025): VS Code 插件，结合 LLM 与 Web 搜索追溯生成代码的可能来源及其许可证信息
- **RepoMark** (Qu et al., 2508.21432, Aug 2025): 数据标记框架，通过语义等价代码变体作为"水印"审计代码 LLM 是否未经授权使用特定仓库进行训练，在 5% 误检率下检测成功率超 90%

训练数据许可证风险: "Cracks in The Stack" (Jahanshahi & Mockus, 2501.02628, LLM4Code 2025) 分析 The Stack v2 数据集，发现因来源识别错误导致非宽松许可证代码被纳入训练集。Salerno et al. (2501.17501, MSR 2025) 发现 StarCoder2-15B 中 54.9% 的可提取预训练数据可被还原，许可证信息是最容易被记忆但也最容易在微调后遗忘的数据类别。

版权保护机制: Anchored Decoding (He et al., 2602.07120, Feb 2026) 提出推理时方法，通过将生成约束在宽松许可证训练的"安全模型"附近，消除最多 75% 的可测量复制行为。PROD (Jiang et al., 2506.17125, AAAI 2026) 提出代码遗忘基准，定义了 Pareto Dominance Ratio (PDR) 指标联合评估遗忘质量与模型效用。

价值观对齐: 生成代码是否遵循安全策略，包括拒绝生成恶意代码、检测硬编码密钥和敏感信息泄露等。

五、效率与成本 (Efficiency & Cost)

核心问题：生成的代码跑得快不快？生成过程本身的成本是否合理？

5.1 时间效率

定义: 生成代码的执行时间与参考实现的比值，即归一化执行时间 $NET = ET_{code} / ET_{canonical}$ 。

EffiBench 实证数据 (Huang et al., NeurIPS 2024 D&B): 在 1,000 道效率敏感的 LeetCode 题目上评测 42 个 LLM (35 个开源 + 7 个闭源)，每题配有人类编写的最优效率参考实现。核心发现：

模型	平均 NET (倍)	最差 NET (倍)	平均内存倍率 (NTMU)
GPT-4	3.12x	13.89x (排序子集)	6.36x
StarCoder2-15B	2.59x	-	4.83x
GPT-3.5-turbo	~3.19x	70.62x (DP 子集)	-
Claude-3-sonnet	-	17.43x (回溯子集)	-

动态规划和回溯算法子集的效率差距最大。pass@1 高不等于效率高：GPT-4-turbo-preview 的 pass@1 (65.4%) 高于 GPT-4，但效率更低。

EffiBench-X (Qing et al., 2505.13004, May 2025) 将评测扩展到 Python、C++、Java、JavaScript、Ruby、Go 六种语言，发现即使最优 LLM (Qwen3-32B) 平均也仅达到人类效率的约 62%，且语言间差异显著。

效率比分级参考：ER = 生成代码运行时间 / 最优解运行时间。ER ≤ 1.5 为优秀， $1.5 < ER \leq 5$ 为可接受， $ER > 5$ 需要优化， $ER = \infty$ 为超时。

5.2 高性能计算场景的效率评测

定义：针对 CUDA kernel、数值计算、系统级代码等 HPC 场景，以实测 GFLOPS、加速比 (speedup ratio) 或运行时间缩减比例作为效率指标。

评测方法与最新研究：

- **MaxCode** (Ou et al., 2601.05475, Jan 2026)：基于 Max-reward RL 的代码优化框架，集成自然语言批评模型将原始执行反馈（计时、设备利用率）转化为诊断信息。在 KernelBench (CUDA) 和 PIE (C++) 基准上分别取得 20.3% 和 10.1% 的相对加速提升
- **Real-Machine Benchmark Rewards** (Mikasa et al., 2602.12049, Feb 2026)：直接使用超算实测 GFLOPS 作为 RL 奖励信号训练 Qwen2.5 Coder 14B，提出 Staged Quality-Diversity (SQD) 算法，在双精度矩阵乘法任务上验证。这是目前最直接的 FLOPs 导向评测实践

- MARCO (Rahman et al., 2505.03906, May 2025): 多 Agent HPC 代码优化框架，分离代码生成和性能评估 Agent 并通过反馈环连接，相比 Claude 3.5 Sonnet 单独生成平均运行时间缩减 14.6%

目前 HPC 场景尚无类似 EffiBench 的标准化基准，各研究自定义评测协议，统一框架仍是开放问题。

5.3 生成成本与交互效率

指标	定义	计算
Cost-per-Task	完成单个任务的总成本	token 消耗 × 单价
Token-to-Solution Efficiency	解决同一问题所需的 token 数	对比不同模型/Agent
步骤效率	完成任务的交互轮数	Agent 场景
Wall-clock Time	从提交到完成的实际耗时	端到端

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。[Cost-Aware Model Selection, 2602.06370, Feb 2026] 提出了多目标优化框架。

步骤效率最新研究：

- EGSS (2602.05242, Feb 2026): 基于熵引导的逐步扩展策略
- SWE-Replay (2601.22129, Jan 2026): 高效的 test-time scaling

六、上下文适配与泛化能力 (Context Adaptation & Generalization)

核心问题：生成代码能否融入已有代码库，并在不同语言、领域间保持稳定表现？

6.1 架构契合度

定义：生成代码是否遵循项目的架构模式和设计约定。

量化方式：

- 设计模式一致性（如项目用 MVC，生成代码是否遵循）
- 模块边界尊重（是否跨越了不应跨越的模块边界）
- 仓库级上下文文件的利用 [Evaluating AGENTS.md, 2602.11988, Feb 2026]

6.2 工具链适配度 (Toolchain Adaptation)

定义：模型对特定 IDE/Agent 框架工具链调用的准确率。

评测维度：

- 终端命令正确率（生成的 shell 命令能否正确执行）
- 文件操作正确率（读写路径、权限等）
- 工具调用成功率（API 调用、数据库查询等）

工业实践：头部产品将代码生成置于具备终端操作、文件系统访问能力的 Agent 环境中，评测模型在真实工具链中的表现，而非孤立的代码片段生成。

6.3 私有 API 学习速率 (Private API Adaptation Rate)

定义：模型在接触非公开 SDK/API 文档后的快速上手能力。

计算：给定 N 个 API 使用示例后，模型正确调用该 API 的准确率曲线。

工业意义：ToB 场景中，企业往往有大量私有 SDK 和内部 API。模型能否在少量示例下快速适配，直接决定了产品在企业场景的可用性。

6.4 跨语言泛化

定义：模型在主流语言（Python/JS/Java）与长尾语言（VB/COBOL/Fortran/Lua）之间的性能差距。

计算：长尾语言 pass@1 / 主流语言 pass@1

工业实践：头部产品特别评测 Visual Basic 等长尾语言的表现，验证模型在缺乏大规模高质量预训练语料下的推理迁移能力。

参考基准：

- MultiPL-E (Cassano et al., 2023): HumanEval 翻译到 18+ 语言

- HumanEval-X (Zheng et al., 2023): 6 语言对齐评测

6.5 跨领域泛化

评测维度：通用代码 vs 领域特定代码的性能差距。

新兴领域基准：

- 移动端：SWE-bench Mobile (2602.09540, Feb 2026)
 - 前端 UI：ComUIBench (2602.19276, Feb 2026)
 - 智能合约：Agentic Pipeline for Smart Contract (2602.13808, Feb 2026)
 - 硬件描述：SimulatorCoder (2602.17169, Feb 2026)
-

七、补丁与编辑指标（修复/编辑场景）

7.1 补丁最小性与过度修复

核心论文：“Evaluating Software Development Agents: Patch Patterns, Code Quality, and Issue Complexity in Real-World GitHub Scenarios” (Chen & Jiang, 2410.12468, SANER 2025)。该研究分析了 10 个顶级 Agent 在 SWE-Bench Verified 的 500 个真实 GitHub issue 上生成的 4,892 个补丁。

关键发现：

- 330/500 个 issue 被解决，仅 5.8% 的 issue 被所有 10 个 Agent 同时解决，10.2% 仅被单个 Agent 解决
- 文件级对齐度高 (FactoryCodeDroid 达 94.59% F1=1)，但函数级对齐度骤降至 24.32%，说明 Agent 能定位正确文件但常修改错误函数
- 过度修改是主要失败模式：HoneyComb 平均 47.54 行代码变更 vs 金标准补丁 7.71 行；SWE-Agent (Claude 3.5) 平均 43.35 行 vs 7.85 行
- 相反，Gru (5.67 行) 和 Amazon-Q-Dev (6.08 行) 比金标准补丁更精简
- SWE-bench 的单元测试基于金标准补丁编写，当 Agent 修改了不同文件/函数但仍通过测试时，测试可能未覆盖 Agent 的实际改动，存在评测盲区

7.2 State-Diff 评测

核心论文："Agent-Diff: Benchmarking LLM Agents on Enterprise API Tasks via Code Execution with State-Diff-Based Evaluation" (Pysklo, Zhuravel & Watson, 2602.11224, Feb 2026)。

方法论：将每个服务建模为状态机，状态 S 为类型化关系数据库。在 Agent 执行前后分别快照所有数据库表，计算状态差异：

- $\Delta_{\text{add}} = T_{\text{after}} \setminus T_{\text{before}}$ (新增实体)
- $\Delta_{\text{del}} = T_{\text{before}} \setminus T_{\text{after}}$ (删除实体)
- $\Delta_{\text{mod}} = \{(e, e') : e.\text{pk} = e'.\text{pk} \wedge e \neq e'\}$ (修改实体)

任务正确性通过声明式断言验证：每条断言指定 diff 类型、目标实体、字段约束和期望数量。关键设计是闭世界假设：状态差异中任何未被断言覆盖的变更（意外插入、删除或字段修改）视为副作用，任务得分归零。

评测规模：224 个任务，覆盖 Slack、Box、Linear、Google Calendar 四个企业服务，108 个 API 端点，平均任务跨度 5.3 次 API 调用。

八、意图对齐指标（需求→代码）

8.1 指令遵循率 (Instruction Following Rate)

定义：生成代码满足显式约束的比例。

约束类型：语言指定、复杂度限制、禁用 API、输出格式要求等。

计算：约束检查通过数 / 总约束数

参考基准：BigCodeBench (Zhuo et al., 2024) 评测复杂指令下的约束满足能力。

8.2 需求-代码对齐度 (SBC)

核心论文："Bridging LLM-Generated Code and Requirements: Reverse Generation Technique and SBC Metric for Developer Insights" (Ponnusamy, 2502.07835, Feb 2025)。

方法论（逆向生成技术）：

1. 给定需求描述，LLM 生成代码
2. 将生成的代码回传给 LLM，要求其从代码反向重建原始需求（逆向生成）
3. 将逆向生成的需求与原始需求进行对比，计算 SBC 分数

SBC 公式：

$$SBC = 0.7 \times \text{semantic_score} + 0.1 \times \text{BLEU} + 0.2 \times \text{completeness}$$

三个组件：

- **语义相似度 (权重 0.7)**：使用 `all-MiniLM-L6-v2` 句向量模型计算原始需求与逆向生成需求的余弦相似度，衡量意图层面的对齐
- **BLEU (权重 0.1)**：标准 n-gram 匹配，作为语义相似度的补充
- **完整性 (权重 0.2)**：提取两份需求中的关键词（名词、动词、专有名词），计算 $\text{completeness} = \max(0, 1 - (|\text{missing}| + |\text{extra}|) / \max(|\text{keywords_union}|, 1))$ ，其中 `missing` 为原始需求有但逆向生成缺失的关键词，`extra` 为逆向生成多出的关键词（可能暗示幻觉）

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

实验结果：在 90 个需求（覆盖 UI/React/Angular、数据层/SQL、业务逻辑/.NET/Node.js/Spring Boot）上评测 Codellama 13B、Qwen2.5-Coder 14B、Deepseek Coder 6.7B、Codestral 22B 四个模型。SBC > 0.55 表示逆向生成需求质量较高，SBC > 0.65 表示语义高度接近。`missing_elements` 字段揭示被遗漏的需求组件，`extra_elements` 揭示幻觉内容。

参考文献

功能正确性与 Benchmark

1. Chen et al., 2021. "Evaluating Large Language Models Trained on Code" (Codex/HumanEval). arXiv:2107.03374
2. Liu et al., 2024. "Is Your Code Generated by ChatGPT Really Correct?" (EvalPlus). arXiv:2305.01210
3. Gu et al., 2024. "CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution". arXiv:2401.03065
4. Jain et al., 2024. "LiveCodeBench: Holistic and Contamination Free Evaluation". arXiv:2403.07974
5. Zhuo et al., 2024. "BigCodeBench: Benchmarking Code Generation with Diverse Function Calls". arXiv:2406.15877
6. Zhang et al., 2026. "EvoCodeBench: A Human-Performance Benchmark for Self-Evolving LLM-Driven Coding Systems". arXiv:2602.10171

工程质量与可读性

7. Campbell, 2018. "Cognitive Complexity: An Overview and Evaluation". SonarSource
8. Siddiq et al., 2024. "Using LLMs to Evaluate Code Readability"
9. "AI builds, We Analyze: Empirical Study of AI-Generated Build Code Quality", Jan 2026. arXiv:2601.16839
10. Licorish et al., 2025. "Comparing Human and LLM Generated Code: The Jury is Still Out!". arXiv:2501.16857
11. Cotroneo, Improta & Liguori, 2025. "Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity" (ISSRE 2025). arXiv:2508.21634
12. "Investigating The Smells of LLM Generated Code", Oct 2025. arXiv:2510.03029
13. "A Causal Perspective on Measuring, Explaining and Mitigating Smells in LLM-Generated Code", Nov 2025. arXiv:2511.15817
14. "On Inter-dataset Code Duplication and Data Leakage in Large Language Models", Jan 2024. arXiv:2401.07930
15. Clark et al., 2024. "A Quantitative Analysis of Quality and Consistency in AI-generated Code". IEEE ICoSSE
16. Fang et al., 2025. "RTL-Bench: Multi-dimensional Evaluation Framework for RTL Code". ICCD 2025

稳定性与一致性

17. Ouyang et al., 2023. "An Empirical Study of the Non-determinism of ChatGPT in Code Generation". arXiv:2308.02828
18. Donato et al., 2025. "Studying How Configurations Impact Code Generation in LLMs". arXiv:2502.17450

安全性与许可证合规

19. Bhatt et al., 2024. "CyberSecEval 2" (Meta). arXiv:2404.13161
20. Siddiq & Santos, 2022. "SecurityEval Dataset". arXiv:2212.09520
21. "CVE-Factory: Scaling Expert-Level Agentic Tasks for Code Security", Feb 2026. arXiv:2602.03012
22. "Can Adversarial Code Comments Fool AI Security Reviewers", Feb 2026. arXiv:2602.16741
23. "Prompt Poisoning Code: Defect Induction Rates and Security Mitigation", Oct 2025. arXiv:2510.22944
24. "Security and Quality in LLM-Generated Code: Multi-Language Analysis", Feb 2025. arXiv:2502.01853
25. Xu et al., 2024. "LiCoEval: Evaluating LLMs on License Compliance in Code Generation" (ICSE 2025). arXiv:2408.02487
26. Bifolco et al., 2025. "CodeGenLink: A Tool to Find the Likely Origin and License of Automatically Generated Code" (ASE 2025). arXiv:2510.01077
27. Qu et al., 2025. "RepoMark: A Data-Usage Auditing Framework for Code Large Language Models". arXiv:2508.21432
28. Jahanshahi & Mockus, 2025. "Cracks in The Stack: Hidden Vulnerabilities and Licensing Risks in LLM Pre-Training Datasets". arXiv:2501.02628
29. Salerno et al., 2025. "How Much Do Code Language Models Remember?" (MSR 2025). arXiv:2501.17501
30. He et al., 2026. "Anchored Decoding: Provably Reducing Copyright Risk for Any Language Model". arXiv:2602.07120
31. Jiang et al., 2025. "Large Language Model Unlearning for Source Code (PROD)" (AAAI 2026). arXiv:2506.17125

效率与成本

32. Huang et al., 2024. "EffiBench: Benchmarking the Efficiency of Automatically Generated Code" (NeurIPS 2024 D&B). arXiv:2402.02037
33. Qing et al., 2025. "EffiBench-X: Multi-Language Efficiency Benchmark". arXiv:2505.13004
34. Ou et al., 2026. "MaxCode: Max-Reward RL for Code Optimization". arXiv:2601.05475
35. Mikasa et al., 2026. "Improving HPC Code Generation Capability of LLMs via Online RL with Real-Machine Benchmark Rewards". arXiv:2602.12049
36. Rahman et al., 2025. "MARCO: Multi-Agent Framework for HPC Code Optimization". arXiv:2505.03906
37. "Cost-Aware Model Selection for Text Classification", Feb 2026. arXiv:2602.06370
38. "EGSS: Entropy-guided Stepwise Scaling for Reliable SE", Feb 2026. arXiv:2602.05242
39. "SWE-Replay: Efficient Test-Time Scaling for SE Agents", Jan 2026. arXiv:2601.22129

上下文适配与泛化

37. "Evaluating AGENTS.md: Are Repository-Level Context Files Helpful?", Feb 2026. arXiv:2602.11988
38. Cassano et al., 2023. "MultiPL-E: A Scalable and Polyglot Approach to Benchmarks". arXiv:2208.08227
39. Zheng et al., 2023. "CodeGeeX: A Pre-Trained Model with Multilingual Benchmarking on HumanEval-X"

补丁与编辑

40. Chen & Jiang, 2024. "Evaluating Software Development Agents: Patch Patterns, Code Quality, and Issue Complexity" (SANER 2025). arXiv:2410.12468
41. Pysklo, Zhuravel & Watson, 2026. "Agent-Diff: Benchmarking LLM Agents on Enterprise API Tasks via State-Diff-Based Evaluation". arXiv:2602.11224

Agentic Coding 与前沿

45. "SWE-bench Mobile: Can LLMs Develop Industry-Level Mobile Applications?", Feb 2026. arXiv:2602.09540
46. "ComUIBench", Feb 2026. arXiv:2602.19276
47. "SimulatorCoder", Feb 2026. arXiv:2602.17169

意图对齐

48. Ponnusamy, 2025. "Bridging LLM-Generated Code and Requirements: Reverse Generation Technique and SBC Metric". arXiv:2502.07835
 49. "From What to How: Bridging User Requirements with Software Development", Feb 2026. arXiv:2602.13611
-

报告生成日期: 2026 年 2 月 24 日 **版本:** v2.2 **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