# CODEMEM：AST 引导的自适应记忆：面向仓库级迭代代码生成

**Peiding Wang**[1], **Li Zhang**[1], **FangLiu***[1], **Chongyang Tao**[1], **YinghaoZhu**[21]北京航空航天大学计算机科学与工程学院[2]香港大学计算与数据科学学院{wangpeiding, fangliu}@buaa.edu.cn

arXiv:2601.02868v1 [cs.SE] 6 Jan 2026

## 摘要

大语言模型（LLM）通过交互式协作，显著提升了开发者在仓库级代码生成中的效率。然而，随着交互不断推进，必须持续保留并更新仓库上下文，以整合新近验证的信息。与此同时，不断扩展的会话历史加重了认知负担，往往导致遗忘，并使先前已解决的问题再次出现。现有的记忆管理方法虽展现出潜力，却受限于以自然语言为中心的表示方式。为克服这些局限，我们提出 CODEMEM，这是一种专为仓库级迭代代码生成设计的 AST 引导动态记忆管理系统。具体而言，CODEMEM 引入了代码上下文记忆组件，通过 AST 引导的大语言模型操作动态维护和更新仓库上下文；同时引入代码会话记忆，构建交互历史的以代码为中心的表示，并通过基于 AST 的分析显式检测和缓解遗忘。在指令遵循基准测试 CodeIF-Bench 和代码生成基准测试 CoderEval 上的实验结果表明，CODEMEM 达到了最先进的性能：当前轮次的指令遵循能力提升了 12.2%，会话级别提升了 11.5%，交互轮次减少了 2–3 轮，同时保持了有竞争力的推理延迟和令牌效率[1]。

## 1 引言

近年来，大语言模型（LLM）（Zhu 等，2024年；Hui 等，2024年）在代码生成方面取得了显著进展，提高了开发者的生产力，尤其是在仓库级代码生成任务中；在这类任务中，掌握丰富的

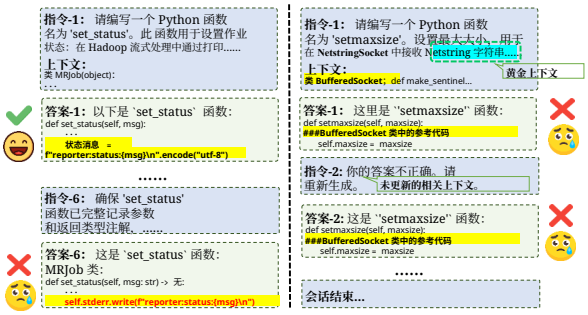



图 1：迭代式代码生成中的失败案例。左：随着交互推进，大语言模型表现出遗忘现象，并覆盖此前正确的修改。右：静态上下文处理会传播错误信息，并阻碍相关上下文的纳入，导致反复生成错误代码。

仓库内上下文至关重要（Wang 等人，2025年b；Li 等，2025a）。然而，现实世界中的开发工作流通常包含多轮交互，例如细化需求或要求修复有错误的代码（Wang 等人，2025a；Zhan 等，2025年）。我们将此称为**仓库级迭代代码生成**。如图1所示，在此类场景中，代码上下文本质上是动态的。要满足不断变化的会话指令，就必须有选择地保留关键的历史代码上下文，同时持续整合新近相关的仓库信息。随着交互跨越多个轮次，日益冗长复杂的会话历史会加重大语言模型的认知负担，导致遗忘、覆盖或相互矛盾地修改先前正确的代码（Laban 等，2025年；Bogomolov 等，2024年）。

记忆管理系统（Zhong 等人，2023年；匿名作者，2025年；Li 等，2025年b；Packeret al.，2024年）通过为大语言模型扩充外部记忆，为支持长期交互提供了一个颇具前景的方向。然而，现有方法主要面向自然语言生成而设计，并依赖以自然语言为中心的记忆表示。将其应用于

*通讯作者。[1]代码和数据可从以下地址获取：https://github.com/zhu-zhu-ding/CodeMem

仓库级迭代代码生成时，这类表示往往会模糊代码结构与行为之间的区别，妨碍有效历史代码上下文的动态保留，以及新近相关仓库信息的整合。此外，基于大语言模型摘要或对话检索的会话记忆无法捕捉不断演变的交互状态，导致行为僵化：大语言模型可能会忘记先前的修正，或反复引入已解决的错误（Laban 等，2025年；Bai 等，2024年）。

为解决这些局限性，我们提出了 CODE-MEM，这是一种专为仓库级迭代代码生成设计的记忆管理机制。CODEMEM 由两个记忆组件组成：

**1) 代码上下文记忆** 动态保留有效的历史代码上下文，同时整合新近相关的仓库信息，以支持不断变化的会话指令。CODEMEM 使用 AST 引导的选择机制保留有用的历史信息，并利用大语言模型判断何时需要更新。更新时，新检索到的代码会与保留的上下文进行增量合并，使记忆能够持续有效地适应会话。

**2) 代码会话记忆** 解决了忘记先前修正内容或重新引入已解决错误等僵化行为。CODEMEM 围绕以代码为中心的单元及其迭代修改来组织记忆，并结合后续轮次的反馈。CODEMEM 将最新记忆与相似的历史案例相结合，形成紧凑的会话级表示，并通过基于 AST 的变更分析检测潜在的遗忘现象，再借助大语言模型的引导缓解不一致问题。

在用于迭代代码生成的指令遵循基准测试 CodeIF-Bench 以及扩展的多轮仓库级代码生成基准测试 CoderEval 上进行的实验表明，CODE-MEM 将当前轮次和会话级的指令遵循能力分别提升了 12.2% 和 11.5%，并可将交互轮次减少约 2–3 轮，显著优于现有的记忆管理方法和完整上下文基线。此外，CODEMEM 在推理时间和令牌成本方面也取得了极具竞争力的表现。

贡献总结如下：

- 我们提出了 CODEMEM，这是一种面向仓库级迭代代码生成的记忆管理系统，可动态更新代码上下文并缓解交互过程中的遗忘。
- 我们提出一种由抽象语法树引导的代码记忆策略，通过代码变更分析保留有效的历史上下文并检测潜在的遗忘。
- 我们在迭代代码生成基准测试上进行了广泛评估，结果表明，该方法在推理延迟和令牌效率方面具有竞争力，同时达到了最先进性能。

## 2 相关工作

**仓库级代码生成。** 代码生成已从独立的函数合成（Austin 等人，2021年；Chen，2021年）发展到仓库级场景（Yu 等，2024年；Li 等，2024年），在此类场景中，模型利用仓库内上下文生成连贯的代码。以往研究（Zhang 等人，2023年；Wang 等人，2025年b；Zhang 等人，2024年）主要聚焦于有效检索仓库上下文。尽管这些方法效果显著，但大多针对单轮交互。近期的基准测试，如 CodeIF-Bench（Wang 等人，2025a）和 SR-Eval（Zhan 等，2025年），将这一设定扩展到多轮迭代场景，但针对如何提升大语言模型在此类场景中的表现，目前仍缺乏系统性方法。

**LLM记忆系统。** 人们提出了记忆管理，以支持大语言模型与其环境之间的长期交互（Xuet al.，2025年；Chhikara etal.，2025年；匿名作者，2025年；Zhong 等人，2023年）。代表性系统包括 MemGPT（Packeret al.，2024年），它采用分层记忆；以及 MemoryOS（Lietal.，2025年b），它引入了类似操作系统的多级存储和动态更新。然而，这些方法主要针对自然语言对话而设计。应用于代码生成时，它们往往将代码视为无结构文本，忽视代码结构、演变以及以代码为中心的交互状态，从而限制了其在代码变更中保持一致性和稳定性的能力。

## 3 方法

### 3.1 任务定义

在本节中，我们定义**仓库级迭代代码生成**任务。给定一个代码仓库，大语言模型通过与用户进行多轮交互，逐步生成并完善代码。每轮交互后，都会运行生成的代码并使用测试套件评估其是否符合当前指令。

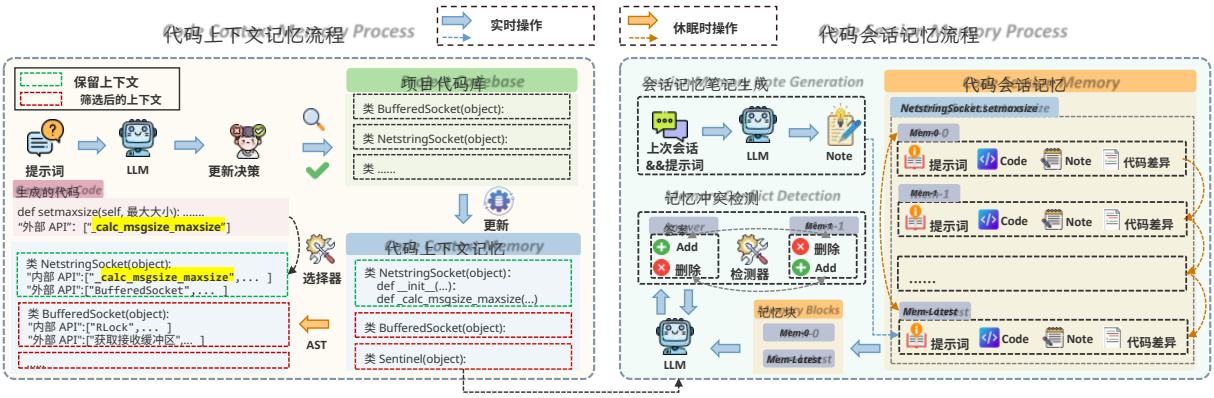



图 2：CODEMEM 的整体流程。

形式化地，给定代码仓库 $R$、用户指令 $I$ 和大语言模型生成的回答 $A$，我们得到以下交互序列：$\{R, I_1, A_1, I_2, A_2, ...I_n, A_n\}$。在第 $i$ 轮，通过运行相关测试套件 $T_i$，对 $A_i$ 的正确性进行定量评估。

### 3.2 概述

如图 2所示，CODEMEM 包含两个记忆组件：**代码上下文记忆**，用于管理仓库级代码上下文；以及**代码会话记忆**，用于管理会话历史。每次会话开始时，CODEMEM 会使用大语言模型判断是否需要更新代码上下文，并在必要时进行更新。会话结束后，**基于抽象语法树的选择器**会保留有效的代码上下文。同时，CODEMEM 会根据提示相似度，将最新记忆块与关联代码块结合起来，构建会话记忆。代码生成后，**基于抽象语法树的检测器**会识别并缓解潜在的记忆冲突，例如遗忘。

### 3.3 代码上下文记忆

#### 3.3.1 代码上下文记忆块

本节将介绍代码上下文记忆块的构建方法。CODEMEM 使用抽象语法树解析器[2],将代码仓库拆解为代码块 $m_{code}$，每个代码块对应一个函数或类。

每个代码块都表示为一个键值对 r:

$$m_{\text{code}} = \begin{cases} \{(s_f), v_f\}, & function \\ \{(s_c, a_c, f_c), v_c\}, & class \end{cases} \quad (1)$$

其中，$s$ 表示带注释的函数或类签名，$a_c$ 表示类属性，$f_c$ 表示类方法，$v$ 存储完整实现。

[2]https://docs.python.org/3/library/ast.html

CODEMEM 在后续管理代码上下文记忆时使用键，并利用值生成响应。这种轻量且高效的设计缩短了输入上下文长度，在提升决策质量的同时，降低了下游大语言模型的推理开销。

#### 3.3.2 基于AST的记忆选择器

该 $Selector$ 旨在仅将有效的历史上下文保留为记忆。第 $t$ 轮之后，$Selector$ 首先从生成的代码中提取外部 API，并从记忆实体中收集内部和外部 API。令 $\mathcal{A}(C_t)_{\text{ext}}$ 表示生成的代码 $C_t$的外部 API 集合，令$\mathcal{M}_{\text{t}} = \{m_i\}$表示代码上下文记忆，其中每个上下文都关联着其 API $\mathcal{A}(m_i)$，包括外部 API 和内部 API。外部 API 是调用其他 API 的方法，而内部 API 则可被其他 API 调用。

$Selector$ 保留 API 与 $C_t$的外部 API 有交集的 $\mathcal{M}_{\text{t}}$：

$$\hat{\mathcal{M}}_{\text{t}} = \{\, m_i \in \mathcal{M}_{\text{t}} \mid \mathcal{A}(m_i) \cap \mathcal{A}(C_t)_{\text{ext}} \neq \emptyset \,\} \quad (2)$$

这既确保了接口级有效性，又保留了语义相关的实现模式，使大语言模型能够在后续生成步骤中有效复用相关代码结构。

#### 3.3.3 记忆处理

在第 $t$ 轮开始时，大语言模型会根据图 D 中的提示模板，判断是否应更新当前代码上下文记忆$\mathcal{M}_{t-1}$（初始化为空），判断依据是 $I$ 同时包含历史和当前指令。此决策仅使用从记忆中提取的键表示 $\{k_i\}$：

$$D = \text{LLM}(I, \{k_i\}) \in \{\texttt{ADD}, \texttt{KEEP}\} \quad (3)$$

其中，ADD 表示需要进行记忆更新，KEEP 表示无需更新。如果 $D = \text{ADD}$，则根据指令从源代码上下文中检索到的相关代码块 $\mathcal{M}_{\text{rel}}$ 会与现有记忆合并；如果 $D = \text{KEEP}$，则记忆保持不变。$\mathcal{M}_t$ 可表示为：

$$\mathcal{M}_t = \begin{cases} \mathcal{M}_{t-1} \cup \mathcal{M}_{\text{rel}}, & \text{if } D = \texttt{ADD} \\ \mathcal{M}_{t-1}, & \text{if } D = \texttt{KEEP} \end{cases} \tag{4}$$

在第 $t$轮交互后，应用 $Selector$，依据代码 $\mathcal{C}_t$筛选并仅保留$\mathcal{M}_t$ 中的有效上下文实体，从而避免冗余累积，并提高长期交互中的生成稳定性和效率。

$$\hat{\mathcal{M}}_t = Selector(\mathcal{M}_t, \mathcal{C}_t) \tag{5}$$

## 3.4 代码会话记忆

### 3.4.1 会话记忆块

我们围绕以代码为中心的单元，而非自然语言，构建会话记忆块（Chhikaraet al.，2025年；Xuet al.，2025年）。给定一个对话，其记忆块表示为：

$$m_{session} = \{I, \mathcal{C}, \Delta\mathcal{C}, \mathcal{N}\} \tag{6}$$

其中，$I$ 是用户提示，$\mathcal{C}$ 是生成的函数代码，$\Delta\mathcal{C}$ 是通过基于 AST 的分析得出的相对于上一轮的代码修改（差异），$\mathcal{N}$是大语言模型根据后续提示生成的、描述该修改的备注。下一轮的反馈可使大语言模型生成更准确的备注，确保未达预期的修改得到适当纠正，或复用正确性信息，从而指导后续编辑。该记忆块设计还凸显了提示词对代码行为的实际影响，并使大语言模型能够在后续轮次中跟踪、复用和纠正修改。

此外，对同一函数的迭代编辑会自然地将多个记忆块关联成一个**记忆序列**($ms_{\text{session}}$)：

$$ms_{\text{session}} = \{m_0, m_1, \ldots, m_{\text{latest}}\} \tag{7}$$

其中，$m_0$ 对应初始函数和后续代码块，并按修改历史排序；$m_{\text{latest}}$ 对应最新记忆。

### 3.4.2 基于AST的记忆检测器

该 $Detector$ 旨在识别新生成的代码与历史会话记忆之间的冲突。其核心思想是，新代码与存储的记忆代码块之间的不一致表明存在记忆冲突和遗忘，例如已撤销的更正或重新引入的约束。

首先，应用指令级过滤，排除对应相似意图的历史记忆块。令 $I_t$ 表示当前指令，$\{I_i\}_{i<t}$ 表示历史指令。我们选择候选检测会话记忆 $\mathcal{M}_c$，使其满足：

$$\mathcal{M}_c = \{\, m_i \mid \text{sim}(I_t, I_i) < \tau \,\} \tag{8}$$

其中，sim(·)表示指令相似度函数，$\tau$为过滤阈值。此过滤可避免因相似指令下的迭代优化或可逆更改而产生虚假冲突。

接下来，$Detector$ 使用抽象语法树级差异，为当前生成的代码 $\mathcal{C}_t$ 提取代码更改 $\Delta^t$，其中 $\Delta^t = (\Delta^t_{\text{add}},\ \Delta^t_{\text{del}})$分别表示添加和删除的抽象语法树节点集合。令 $\Delta^i = (\Delta^i_{\text{add}},\ \Delta^i_{\text{del}})$ 表示与候选检测记忆 $m_i \in \mathcal{M}_c$相关联的已记录代码更改。$Detector$ 通过检查当前更改是否与历史更改矛盾，确定会话记忆块 $m_i$ 是否与 $\mathcal{C}_t$ 冲突：

$$Conf(\Delta^t, \Delta^i) \triangleq \left(\Delta^t_{\text{del}} \cap \Delta^i_{\text{add}} \ \cup \ \Delta^t_{\text{add}} \cap \Delta^i_{\text{del}}\right) \tag{9}$$

其中，$Conf(\Delta^t,\ \Delta^i)$ 捕获其添加与删除操作相互矛盾的抽象语法树节点，这表明先前的代码更改已被遗忘或撤销。

因此，潜在冲突的会话记忆块集合定义如下：

$$\hat{\mathcal{M}}_c = \{\, m_i \in \mathcal{M}_c \mid Conf(\Delta^t, \Delta^i) \neq \varnothing \,\} \tag{10}$$

### 3.4.3 记忆处理

在第 $t$ 轮结束时，该次交互被记录为一个会话记忆块。令 $\mathcal{C}_t$ 表示第 $t$ 轮生成的函数代码，令 $MS = \{ms_i\}$ 表示现有记忆序列的集合。CODEMEM 按如下方式更新会话记忆：

$$MS \leftarrow \begin{cases} MS \cup \{\, m_t \,\}, & \mathcal{C}_t \notin MS \\ ms_i \oplus m_t, & \mathcal{C}_t \in ms_i \end{cases} \tag{11}$$

其中，$m_t$ 存储相对于最新版本的抽象语法树级代码差异 $\Delta^t$，$\oplus$ 表示将一个代码块追加到现有记忆序列中。

在每个记忆序列中，代码块通过指令相似度相互关联：

$$Link(m_t) = \{\, m_i \mid \text{sim}(I_t, I_i) \geq \tau \,\} \quad (12)$$

这使最新代码块 $m_t$ 及其邻居$Link$（$m_t$）能够作为一种紧凑但信息丰富的表示。

在第 $t$ 轮开始时，CODEMEM 根据当前指令从 $m_{t-1}$ 生成修改说明，并构建代码会话记忆如下：

$$\mathcal{M}_t = \{m_{t-1}\} \cup Link(m_{t-1}) \quad (13)$$

该记忆将共同用于指导代码生成。生成候选解 $\mathcal{C}_{t+1}$ 后，$Detector$ 会识别存在冲突的记忆代码块：

$$\mathcal{M}_{\text{c}} = Detector(I_{t+1}, \mathcal{C}_{t+1}) \quad (14)$$

ˆ 最终输出 $\mathcal{C}_{t+1}$ 确定如下：

$$\hat{\mathcal{C}}_{t+1} = \begin{cases} \text{LLM}(\mathcal{M}_t, \mathcal{M}_{\text{c}}), & \text{if } \mathcal{M}_{\text{c}} \neq \varnothing \\ \mathcal{C}_{t+1}, & \text{if } \mathcal{M}_{\text{c}} = \varnothing \end{cases} \quad (15)$$

其中，LLM($\mathcal{M}_t, \mathcal{M}_{\text{c}}$) 表示大语言模型根据 $\mathcal{M}_{\text{c}}$ 重新生成代码。

## 4 实验设置

在本节中，我们通过实验评估 CODEMEM 的有效性。我们旨在回答以下研究问题：

• **RQ1：整体性能**。与基线方法相比，CODEMEM 在迭代式仓库级代码生成中的整体表现如何?

• **研究问题二：消融研究**。所提出的记忆组件对 CODEMEM 的性能有何影响?

• **研究问题3：效率与成本分析**。CODEMEM 的时间效率和计算成本如何?

• **研究问题4：对抽象语法树各部分的进一步分析**。对 CODEMEM 的抽象语法树组件进行可解释性分析。

### 4.1 基准测试与指标

**基准测试**。为评估 CODEMEM 在仓库级迭代代码生成任务中的有效性，我们选用了迭代代码生成指令遵循基准测试 CodeIF-Bench（Wang 等人，2025a）以及仓库级代码生成基准 CoderEval（Yu 等人，2024年）。

对于 CodeIF-Bench，我们选择了 L-2 仓库级编程任务，其中包含 40 段对话和 360 条指令，符合真实世界的开发场景。每段对话都围绕一项 Python 代码生成任务展开，包含 9 条各不相同且互不冲突、可供测试和验证的指令。CodeIF-Bench 用于评估大语言模型在交互过程中遵循指令的能力：1）遵循用户当前指令的能力；2）遵循会话级指令的能力；3）交互过程中因遗忘先前正确修改而产生错误的情况（即遗忘现象）。

CoderEval 包含从真实项目中收集的 230 项 Python 仓库级代码生成任务。按照先前的研究（Wang 等人，2024年），我们将 CoderEval 扩展为多轮迭代代码生成任务：对于未能通过测试的代码，我们提供简单的文字反馈，以便重新生成代码。这一设置模拟了人类与大语言模型之间的迭代交互过程：用户反复要求修改有误的代码，并通过衡量在固定交互轮次内完成的用户任务数量，评估协作效率。数据示例见附录 B。

**指标**。我们采用以下指标（He 等人，2024年；Wang 等人，2025a）来评估大语言模型生成代码的正确性以及大语言模型的指令遵循能力：

• **IA(↑)**：大语言模型在每轮次遵循当前指令的能力，通过针对特定轮次的测试进行衡量。

• **CA(↑)**：大语言模型满足迄今为止所有已发布指令的能力，通过累积测试进行评估。

• **IFR(↓)**：此前已满足的指令在后续轮次中失效的比例，表明模型出现遗忘。

• **Pass@k(↑)**：在每次迭代中，至少有一个$k$ 个生成的程序通过测试的概率。

更多详情请参阅附录A。

### 4.2 基线方法与实现细节

我们选择了以下基线方法：Full-Context（FC），它使用全部对话历史作为上下文；以及最先进的记忆管理方法

| 方法 | 第1轮 | 第2轮 | 第3轮 | 第4轮 | 第5轮 | 第6轮 | 第7轮 | 第8轮 | 第9轮 | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|
| 指令准确率（IA） | | | | | | | | | | |
| $FC_{BM25}$ | 37.5 | 40.0 | 52.5 | 45.0 | 27.5 | **60.0** | 37.5 | 35.0 | 35.0 | 41.1 |
| $FC_{RL\text{-}Coder}$ | 35.0 | 37.5 | 50.0 | 37.5 | **35.0** | 60.0 | **47.5** | 35.0 | 27.5 | 40.6 |
| MemGPT | 35.0 | 42.5 | 42.5 | 42.5 | 27.5 | 50.0 | 32.5 | 40.0 | 40.0 | 39.1 |
| Mem0 | 37.5 | 27.5 | 37.5 | 32.5 | 20.0 | 40.0 | 42.5 | 25.0 | 10.0 | 32.5 |
| A-Mem | 37.5 | 42.5 | 45.0 | 47.5 | 27.5 | 50.0 | 30.0 | 47.5 | 42.5 | 41.1 |
| CODEMEM | **40.0** | **50.0** | **57.5** | **50.0** | 32.5 | 40.0 | **47.5** | **50.0** | **47.5** | **46.1** |
| 不含 CtxMem | 40.0 | 47.5 | 57.5 | 47.5 | 27.5 | 50.0 | 32.5 | 37.5 | 42.5 | 42.5↓ |
| 无 CtxAST | 40.0 | 42.5 | 47.5 | 47.5 | 30.0 | 50.0 | 37.5 | 40.0 | 40.0 | 41.7↓ |
| 无 SessAST | 40.0 | 42.5 | 62.5 | 47.5 | 27.5 | 42.5 | 47.5 | 47.5 | 40.0 | 44.2↓ |
| 对话准确率（CA） | | | | | | | | | | |
| $FC_{BM25}$ | 37.5 | 38.8 | 43.3 | 43.1 | 35.5 | **39.6** | 37.9 | 34.4 | 35.3 | 38.4 |
| $FCt_{RL\text{-}Coder}$ | 35.0 | 33.8 | 36.7 | 36.9 | 34.5 | 36.2 | 37.5 | 38.8 | 36.7 | 36.2 |
| MemGPT | 35.0 | 36.3 | 35.0 | 40.0 | 33.5 | 37.9 | 34.3 | 38.1 | 36.4 | 38.2 |
| Mem0 | 37.5 | 27.5 | 28.3 | 28.7 | 17.0 | 16.7 | 21.8 | 25.3 | 20.3 | 24.8 |
| A-Mem | 37.5 | 41.3 | 40.8 | 43.8 | 26.0 | 38.8 | 37.1 | 36.6 | 38.9 | 37.8 |
| CODEMEM | **40.0** | **47.5** | **48.3** | **46.3** | **37.0** | 37.9 | **41.1** | **43.1** | **44.4** | **42.8** |
| 不使用 CtxMem | 40.0 | 45.0 | 50.8 | 46.9 | 30.5 | 34.6 | 37.1 | 33.8 | 37.5 | 39.6↓ |
| 不使用 CtxAST | 40.0 | 42.5 | 41.7 | 39.4 | 36.5 | 36.3 | 37.7 | 37.2 | 37.4 | 38.7↓ |
| 不使用 SessAST | 40.0 | 43.8 | 50.8 | 46.3 | 33.5 | 36.2 | 39.3 | 42.2 | 40.8 | 41.4↓ |

表 1：大语言模型在 CodeIF-Bench 上的总体性能。指标为 IA 和 CA。

| 方法 | 第1轮 | 第2轮 | 第3轮 | 第4轮 | 第5轮 |
|---|---|---|---|---|---|
| $FC_{BM25}$ | 40.0 | 45.2 | 48.7 | 50.4 | 50.9 |
| $FCt_{RL\text{-}Coder}$ | 37.7 | 40.1 | 40.1 | 41.4 | 41.4 |
| A-Mem | 40.4 | 44.3 | 45.7 | 46.5 | 46.5 |
| Mem0 | 40.0 | 41.7 | 42.6 | 43.0 | 43.0 |
| MemGPT | 33.0 | 42.2 | 46.5 | 48.7 | 49.1 |
| CODEMEM | **42.2** | **49.6** | **51.7** | **54.3** | **55.7** |
| 不使用 CtxMem | 42.2 | 47.8↓ | 49.6↓ | 50.0↓ | 51.7↓ |
| 不含 CtxAST | 42.2 | 46.5↓ | 50.0↓ | 50.8↓ | 51.3↓ |
| 不含 SessAST | - | - | - | - | - |

表 2：大语言模型在 CoderEval 上的总体性能。指标为 Pass@1。

MemGPT（Packer et al.，2024年）、Mem0（Chhikaraet al.，2025年）和 A-Mem（Xu et al.，2025年）。我们还对比了单轮代码生成最先进方法 RL-Coder，其多轮配置与 FC 相同。我们将其定义为 $FCt_{RL\text{-}Coder}$。所有方法的骨干模型均为 DeepSeek-V3.2，并采用贪心解码。除 $FCt_{RL\text{-}Coder}$ 外，所有方法均使用 BM25 进行代码上下文检索。在记忆检索和相似度计算方面，所有基于记忆的方法（包括 CODEMEM）均采用 text-embedding-3-small 向量检索器。所有检索设置的文档总数均为 5。CODEMEM 的相似度阈值为 0.95。

## 5 实验结果

### 5.1 RQ1：总体性能

**在 CodeIF-Bench 上的表现**。表1 显示，CODEMEM 的表现始终优于所有基线方法。在 IA 分数方面，除第 5 轮和第 6 轮外，它在所有轮次中都取得了最佳表现。尽管 MemGPT 和 A-Mem 等记忆管理基线方法取得了与 FC 相当的结果，但它们面向自然语言的记忆表示限制了其效用，因此表现不如 CODEMEM。我们的优势源于动态代码上下文更新，以及包含准确修改说明的高效会话记忆，使大语言模型能够更好地满足当前指令。第 5 轮和第 6 轮主要关注非功能性需求，例如降低圈复杂度，而这对大语言模型来说仍然颇具挑战。这两轮中的表现略有下降，表明 CODEMEM 能够有效整合先前指令，但在处理复杂的非功能性约束时仍有困难。在 CA 指标方面，除第 6 轮外，CODEMEM 的表现均优于所有基线方法。值得注意的是，随着交互轮次增加（例如第 6、7 和 8 轮），几乎所有基线方法（如 A-Mem 和 MemGPT）的表现都会下降，而 CODEMEM 则持续提升。这得益于代码会话记忆机制缓解了遗忘问题。总体而言，这些结果凸显了 CODEMEM 在迭代式仓库级代码生成中的有效性。

**在 CoderEval 上的表现**。表2展示了 CoderEval 基准测试的结果。在第一轮中，性能差异主要源于提示设计（附录D）。在后续轮次中，CODEMEM 凭借动态代码上下文更新，持续优于所有基线方法。这种更新会过滤无关代码并整合与任务相关的上下文；此外，用户引导的反馈也能生成更准确的反思备注。这些机制减少了解决错误所需的交互轮次：例如，CODEMEM 在第 3 轮达到的 Pass@1 分数，与 FC 到第 5 轮才达到的分数相近。

在 CodeIF-Bench 上表现与 FC 相近的 MemGPT 和 A-Mem 等基线方法，在这里的表现较弱，凸显了它们有限的泛化能力。这些结果进一步证实了 CODEMEM 在迭代式仓库级代码生成中的有效性。

**RQ1 总结：** CODEMEM 在指令遵循和迭代生成方面始终优于所有基线方法。与 FC 相比，CODEMEM 将 IA 的指令遵循能力提升了 12.2%，将 CA 提升了 11.5%，还能减少 2–3 个交互轮次。

### 5.2 研究问题二：消融研究

为评估基于 AST 的记忆组件的有效性，我们通过系统地移除各个元素进行消融研究。具体而言，**CtxMem** 表示完整的代码上下文记忆，**CtxAST**表示用于代码上下文记忆的基于抽象语法树的选择器，**SessAST**表示用于代码会话记忆的基于抽象语法树的检测器。

在 CodeIF-Bench（表1）上，移除 CtxMem（将代码上下文视为静态内容）会显著降低性能，凸显了在迭代生成过程中主动管理上下文的重要性。移除 CtxAST 会造成更大的性能下降，因为仅靠大语言模型无法筛除无关代码，也无法及时更新，导致上下文充斥噪声并误导决策。同样，省略 SessAST 也会持续降低性能，尤其是在后续轮次中（例如，第 9 轮的 CA 从 44.4% 降至 40.8% ），这是因为先前正确修改内容的遗忘加剧。

对于 CoderEval（表 2），每个会话都专注于单个任务，且几乎不会发生遗忘，因此无需进行 SessAST 消融实验。不过，移除 CtxMem 或 CtxAST 仍会显著降低性能，而 CODEMEM 则通过动态保留相关代码上下文，持续带来稳定的提升。这些结果凸显了我们基于抽象语法树的记忆管理在迭代式仓库级代码生成中的关键作用。

| 方法 | CodeIF-Bench | | CoderEval | |
|---|---|---|---|---|
| | 平均值 #时间 | 平均值 #令牌 | 平均值 #时间 | 平均值 #Token |
| $FC_{BM25}$ | 34.3 | 131.8k | 26.5 | 35.6k |
| $FC_{RLCoder}$ | 36.1 | 72.5k | 19.9 | 19.5k |
| A-Mem | 47.4 | 358.5k | 42.3 | 119.8k |
| Mem0 | 67.0 | 70.0k | 55.2 | 8.0k |
| MemGPT | 27.9 | 31.0k | 38.5 | 61.3k |
| CODEMEM | 54.1 | 107.8k | 34.2 | 52.6k |
| 不含 CtxMem | 51.3 | 159.4k | 30.2 | 40.3k |
| 不含 CtxAST | 58.6 | 315.2k | 33.1 | 80.2k |
| 不含 SessAST | 36.4 | 75.1k | – | – |

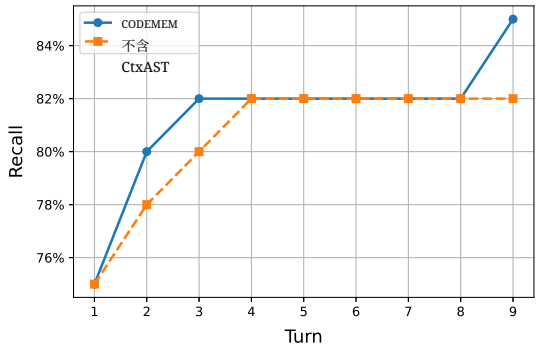
表3：各方法每条数据的平均成本（以令牌数计）和每轮完成时间（以秒计）。

**研究问题二总结：** CtxMem 通过动态更新代码上下文记忆，大幅提升了大语言模型的性能。在 CtxMem 中，CtxAST 通过选择性保留有效的代码上下文记忆发挥着关键作用，而 SessAST 则通过识别并缓解记忆遗忘进一步提升性能。

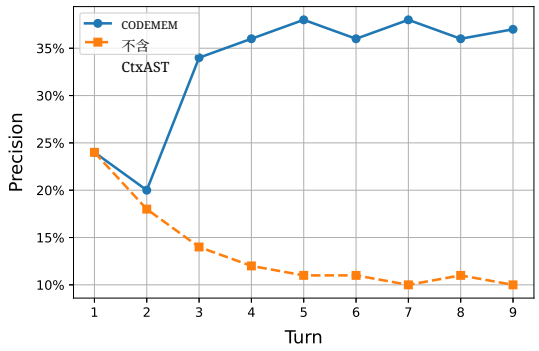
### 5.3 研究问题3：效率与成本分析

我们进一步分析 CODEMEM 的推理时效率和令牌成本（表3）。CODEMEM 在保持效率竞争力的同时，实现了最佳的代码生成质量。A-Mem 等基线方法将代码上下文视为自然语言，导致推理开销显著增加、令牌消耗也更高。Mem0 通过压缩上下文来减少令牌用量，但需要进行多轮推理，因此延迟较高。MemGPT 在 CodeIF-Bench 上的成本最低，但其在 CoderEval 上表现不佳，反映出效率不稳定。由于检索到的上下文更短、更细粒度，采用 RL-Coder 的 FC 通常比 BM25 耗时更短、成本更低。

与多轮对话方法相比，CODEMEM 利用基于键值的代码上下文表示，在 CodeIF-Bench 上减少了 3 万个 token 的使用量。虽然在 CoderEval 上的成本略高于 FC 和 Mem0，但其效率仍高于其他基线方法。消融分析表明，移除 CtxMem 后，由于无需大语言模型控制上下文管理，推理时间略有减少；而键值设计仅带来少量开销（在 CodeIF-Bench 上每轮增加 2.8 秒，在 CoderEval 上增加 4.0 秒），同时显著降低了 token 使用量，尽管在 CoderEval 上有所增加。移除 CtxAST 会进一步增加时间和成本，甚至使 CodeIF-Bench 上的 token 消耗翻倍，这表明未经筛选的上下文会妨碍大语言模型的决策，使其不断引入更多无关上下文。有趣的是，移除 SessAST 后，成本和时间都大幅降低，这表明，




(a) 召回




(b) 精确率

图3：CodeIF-Bench 上每轮代码上下文的累计召回和精确率。

迭代生成往往会产生相互冲突的代码更改，因此需要额外验证。探索更高效的机制来处理此类矛盾，是我们未来的工作方向。

**研究问题3总结：**与基线方法相比，CODEMEM 在时间效率和令牌成本方面表现出具有竞争力的性能。CtxMem 带来的时间开销微乎其微，尽管偶尔会略微增加成本，但总体上节省了令牌；而 SessAST 则通过增加额外成本和推理时间来处理遗忘问题。

### 5.4 研究问题4：对 ASTParts 的进一步分析

**基于AST的记忆选择器。** 我们通过衡量记忆中相关代码的召回和精确率来评估 CtxAST 的有效性。在提供真实标注上下文的 CodeIF-Bench 上，CodeMEM 逐步检索出更多有效上下文，同时持续提高精确率（图 3）。这一改进源于 CtxAST 的过滤作用。第二轮中精确率暂时下降，是由于重新检索过程中引入了噪声上下文，但后续轮次显示出了有效的剪枝效果。当移除 CtxAST、仅由大语言模型管理上下文时，大语言模型

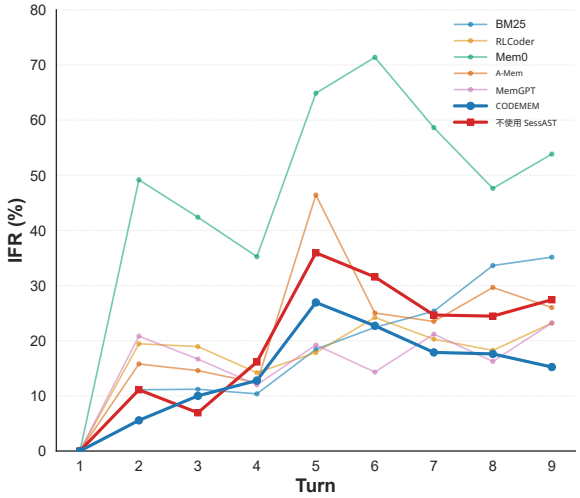



图4：CodeIF-Bench 上的遗忘率（IFR）结果。

会持续检索并增加噪声上下文，导致生成质量下降，同时增加成本和推理时间。

**基于AST的记忆检测器。**我们在 CodeIF-Bench 上使用遗忘率（IFR）指标评估 SessAST（图4）。与基线方法相比，CODEMEM 显著减少了遗忘，且这种改善在后续轮次中更为明显。Mem0 的 IFR 随着轮次推进而急剧上升，凸显了基于自然语言的记忆在迭代代码任务中的局限性。移除 SessAST 会显著增加遗忘，这凸显了它在代码会话记忆中的关键作用。此外，CODEMEM 在后续轮次中持续降低 IFR，表明它能够缓解长时间跨度的迭代式代码生成中的遗忘。

**研究问题4总结：** 进一步的结果表明，CtxAST 提升了有效上下文召回率和精确率，从而提供高质量的代码上下文记忆；而 SessAST 降低了大语言模型的遗忘率，进一步提升了 CODEMEM 的性能。

## 6 结论

在本研究中，我们提出了 CODEMEM，这是一种专为仓库级迭代代码生成设计的记忆管理系统。它通过 AST 引导的代码上下文记忆，动态保留并整合有效的代码上下文；同时借助 AST 构建以代码为中心的代码会话记忆，以缓解大语言模型的遗忘问题。在 CodeIF-Bench 和 CoderEval 上的实验结果表明，CODEMEM 不仅在质量上优于最先进的基线方法，还实现了出色的

时间效率和成本表现。未来，我们将进一步优化 CODEMEM，使其适用于更复杂的场景。

## 7 局限性

**评估场景有限。** 尽管我们的评估采用了指令遵循基准测试（CodeIF-Bench），该基准能够高度近似真实场景，并按照先前工作的做法扩展了现有代码生成基准 CoderEval，但仍存在以下局限：我们的模型尚未在更广泛、更复杂的场景中得到验证，例如更长的交互轮次和更复杂的对话场景。受限于缺乏此类高质量基准测试，我们将在未来的工作中继续关注这些基准测试，并据此调整 CODEMEM。

**大语言模型固有的偏差。** CODEMEM 在一定程度上依赖大语言模型进行决策并生成与上下文相关的记忆。这种对大语言模型固有能力及其偏好局限性的依赖，可能导致错误决策和幻觉。此外，代码上下文记忆依赖大语言模型正确利用上下文来筛选相关信息。大语言模型可能会忽略相关上下文，从而过滤掉关键要素。未来，我们将研究提升大语言模型上下文利用准确性的方法，以减轻这种负面影响。

## 参考文献


匿名作者。2025年。Memagent：通过基于多轮次强化学习的记忆智能体重塑长上下文大语言模型。载于提交至第十四届国际学习表征大会。正在审稿。

Jacob Austin、Augustus Odena、Maxwell Nye、Maarten Bosma、Henryk Michalewski、David Dohan、Ellen Jiang、Carrie Cai、Michael Terry、Quoc Le及另外1位作者。2021年。使用大语言模型进行程序合成。*arXiv*预印本 *arXiv:2108.07732*。

Ge Bai、Jie Liu、Xingyuan Bu、Yancheng He、Jia-heng Liu、Zhanhui Zhou、Zhuoran Lin、Wenbo Su、Tiezheng Ge、Bo Zheng和Wanli Ouyang。2024年。MT-bench-101：用于评估大语言模型多轮对话能力的细粒度基准。载于计算语言学协会第*62*届年会论文集（第一卷：长篇论文），第7421–7454页，泰国曼谷。计算语言学协会。

Egor Bogomolov、Aleksandra Eliseeva、Timur Galimzyanov、Evgeniy Glukhov、AntonShapkin、Maria Tigina、Yaroslav Golubev、Alexander Kovrigin、Arie van Deursen、Maliheh Izadi 和 Timofey Bryksin。2024年 Long code arena：一组面向长上下文代码模型的基准测试。预印本，arXiv:2406.11612。

Mark Chen。2021年。评估基于代码训练的大语言模型。*arXiv* 预印本 *arXiv:2107.03374*。

Prateek Chhikara、Dev Khant、Saket Aryan、Taranjeet Singh 和 Deshraj Yadav。2025年。Mem0：利用可扩展的长期记忆构建适用于生产环境的人工智能智能体。预印本，arXiv:2504.19413。

Yun He、Di Jin、Chaoqi Wang、Chloe Bi、Karishma Mandyam、Hejia Zhang、Chen Zhu、Ning Li、Tengyu Xu、Hongjiang Lv 和其他 1 人。2024年。Multi-if：对大语言模型遵循多轮和多语言指令的能力进行基准测试。*arXiv* 预印本 *arXiv:2410.15553*。

Binyuan Hui、Jian Yang、Zeyu Cui、Jiaxi Yang、Day-iheng Liu、Lei Zhang、Tianyu Liu、Jiajun Zhang、Bowen Yu、Keming Lu、Kai Dang、Yang Fan、Yichang Zhang、An Yang、Rui Men、Fei Huang、BoZheng、Yibo Miao、ShanghaoranQuan 和其他 5 人。2024年。Qwen2.5-Coder 技术报告。预印本，arXiv:2409.12186。

Philippe Laban、Hiroaki Hayashi、Yingbo Zhou 和 Jennifer Neville。2025年。大语言模型在多轮对话中迷失。预印本，arXiv:2505.06120。

Jia Li、Ge Li、Yunfei Zhao、Yongmin Li、Huanyu Liu、Hao Zhu、Lecheng Wang、Kaibo Liu、Zheng Fang、Lanshen Wang、Jiazheng Ding、Xuanming Zhang、Yuqi Zhu、Yihong Dong、Zhi Jin、Binhua Li、Fei Huang、Yongbin Li、Bin Gu 和 Mengfei Yang。2024年。DevEval：与真实代码仓库相契合的人工标注代码生成基准。载于计算语言学协会：*ACL 2024*年年会论文集，第 3603–3614 页，泰国曼谷。计算语言学协会。

Jia Li、Hao Zhu、Huanyu Liu、Xianjie Shi、He Zong、Yihong Dong、Kechi Zhang、Siyuan Jiang、Zhi Jin 和 Ge Li。2025a。aixcoder-7b-v2：训练大语言模型充分利用仓库级代码补全中的长上下文。预印本，arXiv:2503.15301。

Zhiyu Li、Shichao Song、Hanyu Wang、SiminNiu、Ding Chen、Jiawei Yang、Chenyang Xi、Huayi Lai、Ji- hao Zhao、Yezhaohui Wang、Junpeng Ren、Zehao Lin、Jiahao Huo、Tianyi Chen、Kai Chen、Kehang Li、Zhiqiang Yin、Qingchen Yu、Bo Tang 等 3 人。2025年b。Memos：大型语言模型中用于记忆增强生成（MAG）的操作系统。预印本，arXiv:2505.22101。

CharlesPacker, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica 和 Joseph E. Gonzalez。2024年。Memgpt：迈向作为操作系统的大语言模型。预印本，arXiv:2310.08560。

Peiding Wang, Li Zhang, Fang Liu, Lin Shi, Minx- iao Li, Bo Shen 和 An Fu。2025a。Codeif- bench：评估大语言模型在交互式代码生成中的指令遵循能力。预印本，arXiv:2503.22688。

Xingyao Wang、Zihan Wang、Jiateng Liu、Yangyi Chen、Lifan Yuan、Hao Peng 和 Heng Ji。2024年。Mint：使用工具和语言反馈评估大语言模型的多轮交互。预印本，arXiv:2309.10691。

Yanlin Wang、Yanli Wang、Daya Guo、Jiachi Chen、Ruikai Zhang、Yuchi Ma 和 Zibin Zheng。2025年b。*RLCoder*：用于代码库级代码补全的强化学习，第 1140–1152 页。IEEE Press。

Wujiang Xu、Zujie Liang、Kai Mei、Hang Gao、Juntao Tan 和 Yongfeng Zhang。2025年。A-Mem：面向大语言模型智能体的智能体记忆。预印本，arXiv:2502.12110。

Hao Yu、Bo Shen、Dezhi Ran、Jiaxin Zhang、Qi Zhang、Yuchi Ma、Guangtai Liang、Ying Li、Qianxiang Wang 和 Tao Xie。2024年。CoderEval：使用生成式预训练模型进行实用代码生成的基准。载于*IEEE/ACM* 第 *46* 届软件工程国际会议论文集，ICSE '24，美国纽约州纽约。计算机协会。

Zexun Zhan、Shuzheng Gao、RuidaHu 和Cuiyun Gao。2025年。SR-Eval：评估大语言模型在逐步需求细化下的代码生成能力。预印本，arXiv:2509.18808。

Fengji Zhang、Bei Chen、Yue Zhang、Jacky Keung、Jin Liu、Daoguang Zan、Yi Mao、Jian-Guang Lou 和 Weizhu Chen。2023年。RepoCoder：通过迭代检索与生成实现仓库级代码补全。发表于*2023*年自然语言处理实证方法会议论文集，第2471–2484页，新加坡。计算语言学协会。

Kechi Zhang、Jia Li、Ge Li、Xianjie Shi 和 Zhi Jin。2024年。CodeAgent：利用工具集成智能体系统增强代码生成，以应对真实世界的仓库级编码挑战。发表于第*62*届计算语言学协会年会论文集（第一卷：长篇论文），第13643–13658页，曼谷，泰国。计算语言学协会。

Wanjun Zhong、Lianghong Guo、Qiqi Gao 和 Yan- lin Wang。2023年。Memorybank：通过长期记忆增强大语言模型。*arXiv* 预印本 *arXiv:2305.10250*。

Qihao Zhu、Daya Guo、Zhihong Shao、Dejian Yang、PeiyiWang、RunxinXu、Y Wu、YukunLi、Huazuo Gao、Shirong Ma 以及另外 1 位作者。2024年。Deepseek-coder-v2：打破代码智能领域闭源模型的壁垒。*arXiv* 预印本 *arXiv:2406.11931*。

## A 指标详情

**•指令准确率（IA）**（Wang 等人，2025a）：该指标量化了大语言模型在每一轮对话中正确遵循指令的比例。具体而言，在 $n$轮中，给定历史对话和当前指令 $I_n$，大语言模型生成 $A_n$。是否符合指令 $I_n$ 取决于 $A_n$ 是否通过相应的测试 $T_n$:

$$IA = \begin{cases} 1, & \text{if } A_n \text{ passes } T_n, \\ 0, & \text{otherwise.} \end{cases} \tag{16}$$

**• 对话准确率（CA）**(Wang 等人，2025a）：该指标衡量从第一轮到当前轮次的所有指令中，大语言模型成功遵循的指令所占比例。在第 $n$轮中，给定历史对话 $\{I_1, A_1, \ldots, I_{n-1}, A_{n-1}\}$ 和当前指令 $I_n$，大语言模型输出$A_n$。通过根据完整测试序列 $A_n$ 评估$TS = \{T_1, \ldots, T_n\}$，即可计算该轮次的 CA 分数:

$$CA = \frac{\text{Number of tests in } TS \text{ passed by } A_n}{\text{Total number of tests in } TS}. \tag{17}$$

**• 指令遗忘率（IFR）**(Wang 等人，2025a）：该指标反映大语言模型在后续轮次中未能遵循此前已遵循指令的比例。如果某条指令在此前某一轮次（1、2、.....、 $n-1$）中得到满足，但在当前轮次$n$中未得到满足，则视为该指令已被遗忘。令 $PTS = \{T'_1, \ldots, T'_k\}$ 表示此前轮次中通过的测试。给定 $A_n$，IFR 的计算方式如下:

$$IFR = \frac{\text{Number of tests in } PTS \text{ failed by } A_n}{\text{Total number of tests in } PTS}. \tag{18}$$

**• Pass@k**(Chen，2021年）：该指标通过针对每条指令执行多个生成的程序来评估大语言模型的性能:

$$\text{Pass@}k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}},$$

其中， $k$ 是大语言模型针对给定指令生成的程序数量， $c$ 是通过测试的程序数量， $n$是生成的程序总数。

## B 数据示例

### CodeIF-Bench 示例

{

命名空间: "boltons.socketutils.NetstringSocket.setmaxsize",
项目_路径: "Utilities/boltons",
补全_路径: "Utilities/boltons/boltons/socketutils.py",
提示词: "设置 NetstringSocket 实例接收 NetString 字符串的最大大小。 It
更新实例的最大大小，并根据新的最大大小值计算 NetString 消息的最大大小
基于新的最大大小值.....

要求: {
输入-输出条件: {
要求: "setmaxsize"函数应接受一个整数"最大大小"参数并更新
该实例的 "最大大小" 相应的属性.....",
测试: "tests/test_socketutils.py::test_setmaxsize_updates_属性"
},
异常处理: { "The 'setmaxsize' 函数在以下情况下应引发 ValueError: '最大大小' 参数
不是正整数或零。",
测试: "tests/test_socketutils.py::test_setmaxsize_raises_valueerror_on_invalid_最大大小"
},
边界情况处理: {
要求: "设置最大值方法应正确处理将最大大小设置为零
并确保任何非空 Netstring 负载都会引发 NetstringMessageTooLong 异常。",
测试: "tests/test_socketutils.py::test_setmaxsize_zero_行为"
},
功能扩展: {
需求: "扩展 'setmaxsize' 函数，使其打印一条消息：'最大大小已设置为新值_最大大小'
用于调试，以指示 '最大大小' 的变化。",
测试测试: "tests/test_socketutils.py::test_setmaxsize_logs_message"
},
注解覆盖率: {
要求: "确保 'setmaxsize' 函数为其参数添加类型注解
和返回类型， 包括一个参数: 'maxsize': 整数，以及返回类型: 无。",
测试: "tests/test_socketutils.py::test_setmaxsize_annotations"
},
代码复杂度:{
要求: "setmaxsize"函数应保持圈复杂度为 1，表示
一个简单的，线性函数。" ,
测试: "tests/test_socketutils.py::test_setmaxsize_复杂度"
},
代码规范:{
要求: "The 'setmaxsize' 函数应遵循 PEP 8 规范， 包括恰当的
缩进， 行长度， 以及间距。",
测试: "tests/test_socketutils.py::test_check_代码_style"
},
上下文使用验证: {
要求: "The 'setmaxsize'函数应使用 '_calc_消息大小_最大大小' 方法来
更新 '_msgsize_maxsize'。",
测试: "tests/test_socketutils.py::test_setmaxsize_uses_calc_msgsize_最大大小"
},
上下文使用正确性验证: {
要求: "验证 '_msgsize_maxsize' 是否根据新的 '最大大小'
使用 '_calc_msgsize_maxsize'。",
测试: "tests/test_socketutils.py::test_setmaxsize_更新_属性"
}
}
}

### 我们实验中的 CoderEval 示例

{_id: "62e60f43d76274f8a4026e28",file_path: "neo4j/_codec/hydration/v1/temporal.py",project: "neo4j/neo4j-python-driver",prompt: "请完成以下代码: def hydrate_time(nanoseconds, tz=None): """用于 'Time' 和 'LocalTime' 值....."feedback_prompt: "你的答案不正确。请重新生成。" }

## C 记忆示例

代码上下文记忆示例 { memory_key: { class_signature: "class NetstringSocket(object): 使用 netstring 协议（参见维基百科和协议规范）进行读取和写入……", class_attributes: ["_msgsize_最大大小", "bsock", "最大大小", "timeout" ], class_methods: ["__init__", "fileno", "settimeout", "read_ns", "write_ns" ]}, memory_value: "class NetstringSocket(object): def __init__(...): self.bsock = BufferedSocket(...) ... def read_ns(...): "... }

代码会话记忆示例 { boltons.socketutils.NetstringSocket.setmaxsize: [{ id: 0, instruction: "请根据上下文编写一个名为 'setmaxsize' 的 Python 函数……", code: "def setmaxsize(self, maxsize): self.maxsize = maxsize self._msgsize_maxsize =self._calc_msgsize_maxsize(maxsize)", note: "函数已正确实现 setmaxsize 方法。", diff_nodes: { added: [ ],removed: [ ]}, state_links: [ ]}, ... { id: 2, instruction: "' setmaxsize' 函数应引发 ValueError……", code: "def setmaxsize(self, maxsize): if not isinstance(maxsize, int) or maxsize < 0: raise ValueError self.maxsize = maxsize self._msgsize_maxsize =self._calc_msgsize_maxsize(maxsize)", note: "答案按照要求正确实现了 setmaxsize 函数，并在相应情况下引发 ValueError……", diff_nodes: { added: ['type': 'If+Raise', 'block': 'if not isinstance(maxsize, int) or maxsize < 0: raise ValueError("maxsize must be a non-negative integer")'], removed: [ ]}, state_links: [ ]},]}

## D 提示词

代码上下文记忆判定提示

**你是代码仓库代码生成任务方面的资深代码仓库记忆管理器。**
你的目标是根据用户的编程指令，判断当前代码仓库的代码上下文记忆是否需要更新。

**决策目标**
根据现有代码仓库上下文对用户指令中提及的实体的涵盖程度，判断是否需要修改代码仓库记忆（**现有代码仓库上下文**）。

**模式（互斥）**

- **保留** — 当现有仓库上下文已包含理解或执行该指令所需的所有相关类/函数时，使用此模式。
- **添加** — 当现有代码仓库上下文缺少与用户指令相关的代码上下文时，使用此模式。

**用户指令：**
{instructions}
**现有代码仓库上下文：**
{现有_代码库_上下文}
**输出格式（严格 JSON）**

```
{
    "mode": "<添加 | 保留>", "action": "<简短、具体地说明要更新或不更新的内
容>", "target_context": "<相关命名空间或 []>的列表" }
```

带记忆的生成提示。

**你是一位专业的仓库级代码生成器。**
你的目 标是利用提供的代码仓库生成正确的函数实现
上下文 和历史记忆块。
**仓库上下文**

```
{repo_context}
```

**记忆块**

```
{memory_blocks}
```

**当前指令**

```
{instruction}
```

**输出要求**
请输出正确的函数实现。