

# CODEMEM: AST-Guided Adaptive Memory for Repository-Level Iterative Code Generation

Peiding Wang<sup>1</sup>, Li Zhang<sup>1</sup>, Fang Liu<sup>\*1</sup>, Chongyang Tao<sup>1</sup>, Yinghao Zhu<sup>2</sup>

<sup>1</sup>State Key Laboratory of Complex & Critical Software Environment,  
School of Computer Science and Engineering, Beihang University, China

<sup>2</sup>School of Computing and Data Science, The University of Hong Kong  
{wangpeiding, fangliu}@buaa.edu.cn

## Abstract

Large language models (LLMs) substantially enhance developer productivity in repository-level code generation through interactive collaboration. However, as interactions progress, repository context must be continuously preserved and updated to integrate newly validated information. Meanwhile, the expanding session history increases cognitive burden, often leading to forgetting and the reintroduction of previously resolved errors. Existing memory management approaches show promise but remain limited by natural language-centric representations. To overcome these limitations, we propose CODEMEM, an AST-guided dynamic memory management system tailored for repository-level iterative code generation. Specifically, CODEMEM introduces the *Code Context Memory* component that dynamically maintains and updates repository context through AST-guided LLM operations, along with the *Code Session Memory* that constructs a code-centric representation of interaction history and explicitly detects and mitigates forgetting through AST-based analysis. Experimental results on the instruction-following benchmark CodeIF-Bench and the code generation benchmark CoderEval demonstrate that CODEMEM achieves state-of-the-art performance, improving instruction following by 12.2% for the current turn and 11.5% for the session level, and reducing interaction rounds by 2–3, while maintaining competitive inference latency and token efficiency<sup>1</sup>.

## 1 Introduction

In recent years, Large Language Models (LLMs) (Zhu et al., 2024; Hui et al., 2024) in code generation have substantially improved developers' productivity, especially in repository-level code generation tasks (Wang et al., 2025b; Zhao et al.,

<sup>\*</sup>Corresponding author.

<sup>1</sup>The code and data are available at <https://github.com/zhu-zhu-ding/CodeMem>

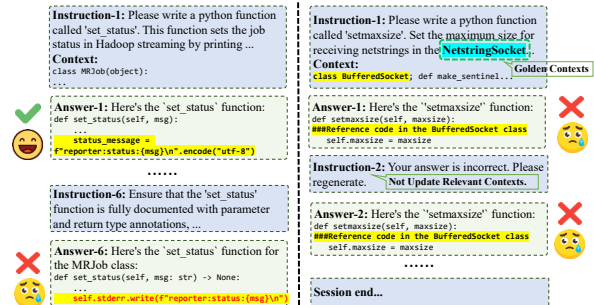


Figure 1: Examples of failure cases in iterative code generation. Left: As interactions progress, the LLM exhibits forgetting and overwrites previously correct modifications. Right: Static context handling propagates erroneous information and prevents the incorporation of relevant contexts, resulting in repeated incorrect code.

2025), where awareness of rich intra-repository context is essential (Wang et al., 2025c; Zhang et al., 2023; Liu et al., 2024). However, real-world development workflows commonly involve multi-turn interactions, such as refining requirements or requesting fixes for erroneous code (Wang et al., 2025a; Zhan et al., 2025). We refer to this as **repository-level iterative code generation**. As illustrated in Figure 1, the code context in such scenarios is inherently dynamic. Satisfying evolving session instructions requires selectively preserving critical historical code context while continuously integrating newly relevant repository information. As interactions span multiple rounds, increasingly long and complex session histories impose growing cognitive burdens on LLMs, leading to forgetting, overwriting, or contradicting previously correct code modifications (Laban et al., 2025; Bogomolov et al., 2024).

Memory management systems (Zhong et al., 2023; Anonymous, 2025; Li et al., 2025; Packer et al., 2024) offer a promising direction for supporting long-term interactions by augmenting LLMs with external memory. However, existing approaches are primarily designed for natural language generation and rely on natural-language-

centric memory representations. When applied to repository-level iterative code generation, such representations often blur code structure and behavior, hindering the dynamic retention of effective historical code context and the integration of newly relevant repository information. Moreover, session memory based on LLM summaries or dialogue retrieval fails to capture the evolving interaction state, leading to rigid behaviors where LLMs may forget prior corrections or repeatedly reintroduce resolved errors (Laban et al., 2025; Bai et al., 2024).

To address these limitations, we propose CODEMEM, a memory management mechanism specifically designed for repository-level iterative code generation. CODEMEM consists of two memory components:

**1) Code Context Memory** dynamically preserves effective historical code context while integrating newly relevant repository information to support evolving session instructions. CODEMEM uses an AST-guided selection mechanism to retain useful history and an LLM to decide when updates are needed. Upon updating, newly retrieved code is incrementally merged with retained context, allowing the memory to remain effective with the session.

**2) Code Session Memory** addresses rigid behaviors such as forgetting prior corrections or reintroducing resolved errors. CODEMEM organizes memory around code-centric units and their iterative modifications, augmented with feedback from later rounds. CODEMEM combines the latest memory with similar past cases to form a compact session-level representation, and detects potential forgetting via AST-based change analysis, mitigating inconsistencies through LLM guidance.

Experiments on instruction following benchmark CodeIF-Bench for iterative code generation and the extended multi-turn repository-level code generation benchmark CoderEval demonstrate that CODEMEM improves current-turn and session-level instruction following by 12.2% and 11.5%, and can reduce interaction rounds about 2–3, significantly outperforming existing memory management and full context baselines. Moreover, CODEMEM has achieved highly competitive inference time and token cost.

The contributions are summarized as:

- We propose CODEMEM, a memory management system for repository-level iterative code generation that dynamically updates code context and mitigates forgetting during interaction.

- We introduce an AST-guided code memory strategy that preserves valid historical context and detects potential forgetting via code-change analysis.
- We conduct extensive evaluations on iterative code generation benchmarks, demonstrating state-of-the-art performance with competitive inference latency and token efficiency.

## 2 Related Work

**Repository-Level Code Generation.** Code generation has evolved from standalone function synthesis (Austin et al., 2021; Chen, 2021; Jain et al., 2025) to repository-level settings (Yu et al., 2024; Li et al., 2024b,a), where models exploit intra-repository context for coherent code generation. Prior work (Zhang et al., 2023; Wang et al., 2025c; Zhang et al., 2024; Wu et al., 2024) has mainly focused on effective repository context retrieval. Although effective, these methods largely target single-round interactions. Recent benchmarks such as CodeIF-Bench (Wang et al., 2025a) and SR-Eval (Zhan et al., 2025) extend this setting to multi-round iterative scenarios, yet systematic approaches for improving LLM performance in such settings remain limited.

**Memory Systems for LLMs.** Memory management has been proposed to support long-term interactions between LLMs and their environment (Xu et al., 2025; Chhikara et al., 2025; Anonymous, 2025; Zhong et al., 2023). Representative systems include MemGPT (Packer et al., 2024), which employs hierarchical memory, and MemoryOS (Li et al., 2025), which introduces system-style multi-level storage and dynamic updates. However, these approaches are primarily designed for natural language dialogue. When applied to code generation, they often treat code as unstructured text, overlooking code structure, evolution, and code-centric interaction states, thereby limiting their ability to maintain consistency and stability in code changes.

## 3 Methodology

### 3.1 Task Definition

In this section, we define the **repository-level iterative code generation** task. Given a code repository, an LLM incrementally generates and refines code through multi-turn interactions with the user. After each interaction round, the generated code is executed against a test suite to evaluate compliance

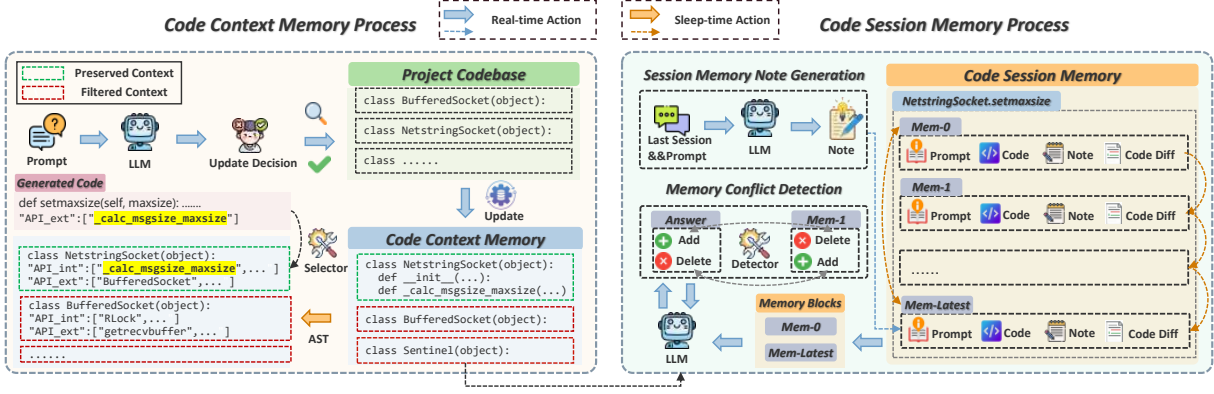


Figure 2: The overall pipeline of CODEMEM.

with the current instruction.

Formally, given code repository  $R$ , user instruction  $I$ , and LLM-generated answer  $A$ , we get the following interaction sequence:  $\{R, I_1, A_1, I_2, A_2, \dots, I_n, A_n\}$ . At round  $i$ , the correctness of  $A_i$  is quantitatively assessed by executing the associated test suite  $T_i$ .

### 3.2 Overview

As shown in Figure 2, CODEMEM consists of two memory components: **Code Context Memory** for managing repository-level code context and **Code Session Memory** for managing session history. At the start of each session, CODEMEM uses the LLM to decide whether the code context requires updating and applies updates when necessary. Upon session completion, the **AST-based Selector** retains effective code contexts. Meanwhile, CODEMEM constructs session memory by combining the latest memory blocks with linked blocks based on prompt similarity. After code generation, the **AST-based Detector** identifies and mitigates potential memory conflicts, such as forgetting.

### 3.3 Code Context Memory

#### 3.3.1 Code Context Memory Block

In this section, we describe the construction of the code context memory block. Using an AST parser<sup>2</sup>, CODEMEM decomposes the repository into code blocks  $m_{code}$  which corresponds to either a function or a class.

Each block is represented as a key–value pair:

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

where  $s$  denote the function or class signatures with comments,  $a_c$  denotes class attributes,  $f_c$  de-

notes class function methods, and  $v$  stores the complete implementation. CODEMEM utilizes keys when subsequently managing code context memory and employs values to generate responses. This lightweight and efficient design minimizes input context length, improving decision quality while reducing inference overhead for downstream LLMs.

#### 3.3.2 AST-based Memory Selector

The *Selector* aims to preserve only effective historical contexts as memory. After round  $t$ , the *Selector* first extracts external APIs from the generated code and collects internal and external APIs from memory entities. Let  $\mathcal{A}(C_t)_{ext}$  denote the set of external APIs for the generated code  $C_t$ , and let  $\mathcal{M}_t = \{m_i\}$  be the code context memory, each associated with its APIs  $\mathcal{A}(m_i)$  including both external and internal APIs. External APIs are methods that call others, while internal APIs can be called by others.

The *Selector* retains the  $\mathcal{M}_t$  whose APIs intersect with the  $C_t$ 's external APIs:

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

This ensures interface-level validity while preserving semantically related implementation patterns, enabling the LLM to effectively reuse relevant code structures in subsequent generation steps.

#### 3.3.3 Memory Processing

At the beginning of round  $t$ , the LLM determines—using the prompt template in Appendix E—whether the current code context memory  $\mathcal{M}_{t-1}$  (initialized as empty) should be updated, based on  $I$  including both historical and current instructions. This decision is made using only the *key* representations  $\{k_i\}$  extracted from the memory:

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

<sup>2</sup><https://docs.python.org/3/library/ast.html>

where ADD indicates that a memory update is required and KEEP indicates that no update is necessary. If  $D = \text{ADD}$ , the relevant code blocks  $\mathcal{M}_{\text{rel}}$  retrieved from the source code contexts according to the instructions are merged with the existing memory; if  $D = \text{KEEP}$ , the memory remains unchanged. The  $\mathcal{M}_t$  can be denoted as:

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

After interaction round  $t$ , the *Selector* is applied to filter and preserve only valid context entities in  $\mathcal{M}_t$  based on code  $\mathcal{C}_t$ , preventing redundant accumulation and improving generation stability and efficiency over long-term interactions.

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

### 3.4 Code Session Memory

#### 3.4.1 Session Memory Block

We construct session memory blocks around code-centric units rather than natural language (Chhikara et al., 2025; Xu et al., 2025). Given a conversation, its memory block is represented as:

$$m_{\text{session}} = \{I, \mathcal{C}, \Delta\mathcal{C}, \mathcal{N}\} \quad (6)$$

where  $I$  is the user prompt,  $\mathcal{C}$  is the generated function code,  $\Delta\mathcal{C}$  is the code modification (diff) relative to the last round by AST-based analysis, and  $\mathcal{N}$  is the note describing the modification generated by the LLM with the subsequent prompt. Feedback in the next round allows the LLM to generate more accurate notes, ensuring that unsatisfactory changes are properly corrected or correctness can be reused, thereby guiding subsequent edits. The block design also highlights the operational impact of prompts on code behavior and enables the LLM to track, reuse, and correct modifications in subsequent rounds.

In addition, iterative edits to the same function naturally link multiple memory blocks into a **memory sequence** ( $m_{\text{session}}$ ):

$$m_{\text{session}} = \{m_0, m_1, \dots, m_{\text{latest}}\} \quad (7)$$

where  $m_0$  corresponds to the initial function and subsequent blocks ordered by modification history and  $m_{\text{latest}}$  corresponds to the latest memory.

#### 3.4.2 AST-based Memory Detector

The *Detector* aims to identify conflicts between newly generated code and historical session memory. The key idea is that inconsistencies between new code and stored memory blocks indicate memory conflicts and forgetting, exemplified by reverted corrections or reintroduced constraints.

First, instruction-level filtering is applied to exclude historical memory blocks corresponding to similar intents. Let  $I_t$  denote the current instruction and  $\{I_i\}_{i < t}$  denote historical instructions. We select the candidate detection session memory  $\mathcal{M}_c$  such that:

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

where  $\text{sim}(\cdot)$  denotes an instruction similarity function and  $\tau$  is the filtering threshold. This filtering avoids spurious conflicts arising from iterative refinements or reversible changes under similar instructions.

Next, the *Detector* extracts code changes  $\Delta^t$  for current generated code  $\mathcal{C}_t$  using AST-level diffs, where  $\Delta^t = (\Delta_{\text{add}}^t, \Delta_{\text{del}}^t)$  denote the sets of added and deleted AST nodes. Let  $\Delta^i = (\Delta_{\text{add}}^i, \Delta_{\text{del}}^i)$  denote the recorded code changes associated with the candidate detection memory  $m_i \in \mathcal{M}_c$ . The *Detector* determines whether the session memory block  $m_i$  is conflicted with the  $\mathcal{C}_t$  by checking whether the current changes contradict its historical changes:

$$\text{Conf}(\Delta^t, \Delta^i) \triangleq (\Delta_{\text{del}}^t \cap \Delta_{\text{add}}^i \cup \Delta_{\text{add}}^t \cap \Delta_{\text{del}}^i) \quad (9)$$

where  $\text{Conf}(\Delta^t, \Delta^i)$  captures AST nodes whose contradictory add–delete operations indicate forgetting or reversion of previously code changes.

Accordingly, the set of potentially conflicting session memory blocks is defined as:

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

#### 3.4.3 Memory Processing

At the end of round  $t$ , the interaction is recorded as a session memory block. Let  $\mathcal{C}_t$  denote the function code generated at round  $t$ , and let  $MS = \{m_{s_i}\}$  denote the set of existing memory sequences. CODEMEM updates the session memory as:

$$MS \leftarrow \begin{cases} MS \cup \{m_t\}, & \mathcal{C}_t \notin MS \\ m_{s_i} \oplus m_t, & \mathcal{C}_t \in m_{s_i} \end{cases} \quad (11)$$

where  $m_t$  stores the AST-level code diff  $\Delta^t$  with respect to the latest version, and  $\oplus$  denotes appending a block to an existing memory sequence.

Within each memory sequence, blocks are linked by instruction similarity:

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

This allows the latest block  $m_t$  and its neighbors  $\text{Link}(m_t)$  to serve as a compact yet information-rich representation.

At the beginning of round  $t$ , CODEMEM generates a modification note from  $m_{t-1}$  conditioned on the current instruction and constructs the Code Session Memory as:

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

which is jointly used to guide code generation. After a candidate solution  $\mathcal{C}_{t+1}$  is produced, the *Detector* identifies conflicting memory blocks:

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

The final output  $\hat{\mathcal{C}}_{t+1}$  is determined as:

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

where  $\text{LLM}(\mathcal{M}_t, \mathcal{M}_c)$  represents that the LLM re-generates code based on  $\mathcal{M}_c$ .

## 4 Experimental Setups

In this section, we conduct experiments to evaluate the effectiveness of CODEMEM. We aim to answer the following research questions:

- **RQ1: Overall Performance.** How does CODEMEM perform overall in iterative repository-level code generation compared with baselines?
- **RQ2: Ablation Study.** How do the proposed memory components impact the performance of CODEMEM?
- **RQ3: Efficiency and Cost Analysis.** What are the time efficiency and computational cost of CODEMEM?
- **RQ4: Further Analysis for AST Parts.** The explainability Analysis of AST Components for CODEMEM.

### 4.1 Benchmarks and Metrics

**Benchmarks.** To evaluate CODEMEM’s effectiveness in repository-level iterative code generation tasks, we selected the iterative code generation instruction-following benchmark CodeIF-Bench (Wang et al., 2025a) and the repository-level

code generation benchmark CoderEval (Yu et al., 2024).

For CodeIF-Bench, we selected the L-2 repository-level programming task, encompassing 40 dialogues with 360 instructions align with real-world development scenarios. Each dialogue centres around a python code generation task, comprising 9 distinct and non-conflicting instructions that can be tested and validated. CodeIF-Bench is to evaluate the LLM’s ability to follow instructions during interactions: 1) The ability to follow the user’s current instruction 2) The ability to follow session-level instructions 3) Errors arising from forgetting previously correct modifications during interaction (the phenomenon of forgetting).

For CoderEval, it comprises 230 python repository-level code generation tasks collected from real-world projects. Following prior study (Wang et al., 2024), we extended CoderEval into multi-round iterative code generation tasks: we provide simple verbal feedback on code that fails testing to facilitate its re-generation. It simulates an iterative human–LLM interaction process in which a user repeatedly requests revisions to erroneous code, aiming to evaluate collaborative efficiency by measuring the number of user tasks completed within a fixed number of interaction rounds. Data examples are presented in the Appendix C.

**Metrics.** We employ the following metrics (He et al., 2024; Wang et al., 2025a) to evaluate both the correctness of LLM generated code as well as the LLMs’ instruction following capability:

- **IA(↑):** The LLM’s ability to follow the current instruction at each turn, measured by round-specific tests.
- **CA(↑):** The LLM’s ability to satisfy all instructions issued so far, evaluated by cumulative tests.
- **IFR(↓):** The proportion of previously satisfied instructions that fail in later turns, indicating forgetting.
- **Pass@k(↑):** The probability that at least one of  $k$  generated programs passes the tests in each iteration.

For further details, please refer to Appendix A.

### 4.2 Baselines and Implementation Details

We selected the following baselines: Full-Context (FC), which uses all conversation history as context, and state-of-the-art memory management methods

Method	Turn-1	Turn-2	Turn-3	Turn-4	Turn-5	Turn-6	Turn-7	Turn-8	Turn-9	Avg.
Instruction Accuracy (IA)										
FC <sub>BM25</sub>	37.5	40.0	52.5	45.0	27.5	<b>60.0</b>	37.5	35.0	35.0	41.1
FC <sub>RL-Coder</sub>	35.0	37.5	50.0	37.5	<b>35.0</b>	60.0	<b>47.5</b>	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	<b>40.0</b>	<b>50.0</b>	<b>57.5</b>	<b>50.0</b>	32.5	40.0	<b>47.5</b>	<b>50.0</b>	<b>47.5</b>	<b>46.1</b>
w/o CtxMem	40.0	47.5	57.5	47.5	27.5	50.0	32.5	37.5	42.5	42.5 <sub>↓3.6</sub>
w/o CtxAST	40.0	42.5	47.5	47.5	30.0	50.0	37.5	40.0	40.0	41.7 <sub>↓4.4</sub>
w/o SessAST	40.0	42.5	62.5	47.5	27.5	42.5	47.5	47.5	40.0	44.2 <sub>↓1.9</sub>
Conversation Accuracy (CA)										
FC <sub>BM25</sub>	37.5	38.8	43.3	43.1	35.5	<b>39.6</b>	37.9	34.4	35.3	38.4
FC <sub>RL-Coder</sub>	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	<b>40.0</b>	<b>47.5</b>	<b>48.3</b>	<b>46.3</b>	<b>37.0</b>	37.9	<b>41.1</b>	<b>43.1</b>	<b>44.4</b>	<b>42.8</b>
w/o CtxMem	40.0	45.0	50.8	46.9	30.5	34.6	37.1	33.8	37.5	39.6 <sub>↓3.2</sub>
w/o CtxAST	40.0	42.5	41.7	39.4	36.5	36.3	37.7	37.2	37.4	38.7 <sub>↓4.1</sub>
w/o SessAST	40.0	43.8	50.8	46.3	33.5	36.2	39.3	42.2	40.8	41.4 <sub>↓1.4</sub>

Table 1: The overall performance of LLMs in CodeIF-Bench. The metrics are IA and CA.

Method	Turn-1	Turn-2	Turn-3	Turn-4	Turn-5
FC <sub>BM25</sub>	40.0	45.2	48.7	50.4	50.9
FC <sub>RL-Coder</sub>	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	<b>42.2</b>	<b>49.6</b>	<b>51.7</b>	<b>54.3</b>	<b>55.7</b>
w/o CtxMem	42.2	47.8 <sub>↓1.8</sub>	49.6 <sub>↓2.1</sub>	50.0 <sub>↓4.3</sub>	51.7 <sub>↓4.0</sub>
w/o CtxAST	42.2	46.5 <sub>↓3.1</sub>	50.0 <sub>↓1.7</sub>	50.8 <sub>↓3.5</sub>	51.3 <sub>↓4.4</sub>
w/o SessAST	-	-	-	-	-

Table 2: The overall performance of LLMs in CoderEval. The metric is Pass@1.

MemGPT (Packer et al., 2024), Mem0 (Chhikara et al., 2025) and A-Mem (Xu et al., 2025). We also compared the single-turn code generation SOTA method RL-Coder, whose multi-turn configuration is identical to FC. We defined it as FC<sub>RL-Coder</sub>. The backbone model for all methods is DeepSeek-V3.2 with greedy decoding. We also provide the results of Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) as the baseline model in Appendix B. All methods except FC<sub>RL-Coder</sub> use BM25 for code context retrieval. For memory retrieval and similarity, all memory-based methods—including CODEMEM—adopt the text-embedding-3-small vector retriever. The total number of documents for all retrieval settings is 5. The similarity threshold for CODEMEM is 0.95.

## 5 Experimental Results

### 5.1 RQ1: Overall Performance

**Performance on CodeIF-Bench.** Table 1 shows that CODEMEM consistently outperforms all base-

lines. For the IA score, it achieves the best performance in all rounds except 5 and 6. While memory management baselines such as MemGPT and A-Mem achieved results comparable to FC, their natural language-oriented memory representation limits their effectiveness, making them inferior to CODEMEM. Our advantage comes from dynamic code context updates and efficient session memory with accurate modification notes, enabling the LLM to better satisfy current instructions. Rounds 5 and 6 focus primarily on non-functional requirements, such as reducing circle complexity, which remain challenging for LLMs. The slight performance drop in these rounds indicates that CODEMEM integrates prior instructions effectively but struggles with complex non-functional constraints. For the CA metric, CODEMEM outperforms all baselines except in round 6. Notably, while nearly all baseline methods, such as A-Mem and MemGPT, exhibit performance degradation as the number of interaction rounds increases, such as round 6, 7 and 8, CODEMEM demonstrates sustained improvement over time. This is due to the Code Session Memory mechanism that mitigates forgetting. Overall, these results highlight the effectiveness of CODEMEM in iterative repository-level code generation. We also provide a case study (Figure 5) to demonstrate the effectiveness of CODEMEM.

**Performance on CoderEval.** Table 2 presents results on the CoderEval benchmark. In the first

round, performance differences largely stem from prompt design (Appendix E). In subsequent rounds, CODEMEM consistently outperforms all baselines due to dynamic code-context updating, which filters irrelevant code and integrates task-relevant context, and user-guided feedback, which generate more accurate reflections notes. These mechanisms reduce the number of interaction rounds needed to resolve bugs: for example, CODEMEM achieves the similar pass@1 score by round 3 that FC requires until round 5. Baselines like MemGPT and A-Mem, which performed similarly to FC on CodeIF-Bench, show weaker performance here, highlighting their limited generalizability. These results further confirm CODEMEM’s effectiveness for iterative, repository-level code generation.

**RQ1 Summary:** CODEMEM consistently outperforms all baselines on both instruction following and iterative generation. Compared to the FC, CODEMEM improves instruction following ability by 12.2% for IA and 11.5% for CA and can also reduce interaction rounds by 2–3.

## 5.2 RQ2: Ablation Study

To evaluate the effectiveness of our AST-based memory components, we conduct an ablation study by systematically removing individual elements. Specifically, **CtxMem** denotes the full Code Context Memory, **CtxAST** the AST-based selector for Code Context Memory, and **SessAST** the AST-based detector for Code Session Memory.

On CodeIF-Bench (Table 1), removing CtxMem—treating code context as static—substantially reduces performance, highlighting the importance of actively managing context during iterative generation. Removing CtxAST causes even larger degradation, as the LLM alone cannot filter irrelevant code and timely updates, leading to noisy context that misguides decision-making. Similarly, omitting SessAST consistently degrades performance, particularly in later rounds (e.g., CA in round 9 drops from 44.4% to 40.8%), due to increased forgetting of previously corrected changes.

For CoderEval (Table 2), each session focuses on a single task and exhibits minimal forgetting; thus, SessAST ablations are unnecessary. Nevertheless, removing CtxMem or CtxAST still significantly impairs performance, whereas CODEMEM maintains consistent gains by dynamically preserving relevant code context. These results underscore the

Method	CodeIF-Bench		CoderEval	
	Avg. #Time	Avg. #Token	Avg. #Time	Avg. #Token
FCBM25	34.3	131.8k	26.5	35.6k
FCRLCoder	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
w/o CtxMem	51.3	159.4k	30.2	40.3k
w/o CtxAST	58.6	315.2k	33.1	80.2k
w/o SessAST	36.4	75.1k	–	–

Table 3: Average cost (in tokens) per data and completion time (in seconds) per round across methods.

critical role of our AST-guided memory management in iterative repository-level code generation.

**RQ2 Summary:** CtxMem greatly improves LLM performance by dynamically updating code context memory. Within CtxMem, CtxAST plays a crucial role by selectively preserving effective code context memory, while SessAST further enhances performance by identifying and mitigating memory forgetting.

## 5.3 RQ3: Efficiency and Cost Analysis

We further analyze the inference-time efficiency and token cost of CODEMEM (Table 3). CODEMEM achieves competitive efficiency while delivering the best code generation quality. Baselines like A-Mem treat code context as natural language, incurring substantial inference overhead and higher token consumption. Mem0 compresses context to reduce tokens but requires many inference iterations, resulting in higher latency. MemGPT attains the lowest cost on CodeIF-Bench, yet its poor performance on CoderEval reflects unstable efficiency. FC with RL-Coder generally has lower time and cost than BM25 due to retrieval of shorter, finer-grained contexts.

CODEMEM leverages key–value–based code context representation, reducing token usage by 30k on CodeIF-Bench compared to multi-turn dialogue methods. While slightly more costly than FC and Mem0 on CoderEval, it remains more efficient than other baselines. Ablation analysis shows that removing CtxMem slightly reduces inference time by eliminating LLM-controlled context management, but the key–value design adds only marginal overhead (2.8s per round on CodeIF-Bench, 4.0s on CoderEval) while substantially lowering token usage despite a partial increase on CoderEval. Omitting CtxAST increases time and cost further, even doubles token consumption on CodeIF-Bench, indicating that unfiltered context negatively affects

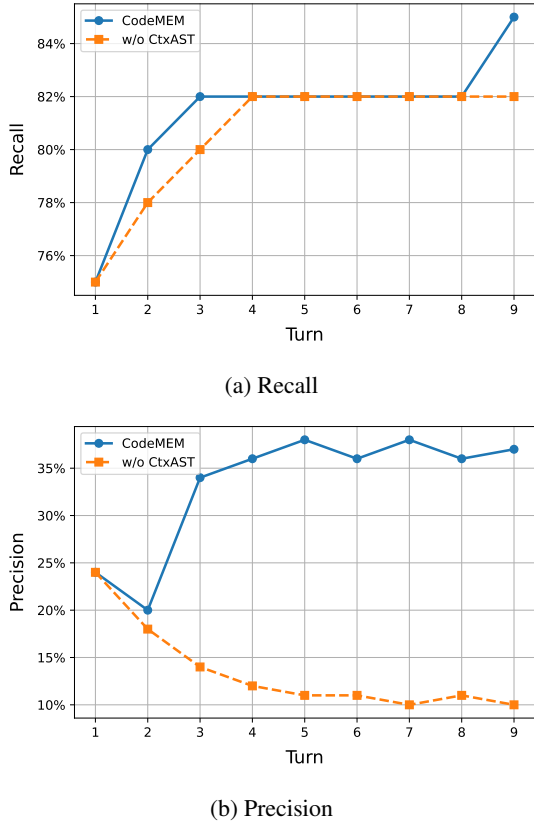


Figure 3: Cumulative recall and precision per round for code contexts on CodeIF-Bench.

LLM decision-making to continually introducing more irrelevant context. Interestingly, removing SessAST substantially reduces cost and time, suggesting that iterative generation often produces conflicting code changes, which require additional validation. Exploring more efficient mechanisms to handle such contradictions is our future work.

**RQ3 Summary:** Compare to baselines, CODEMEM achieves competitive performance in terms of time efficiency and token cost. CtxMem introduces negligible time overhead and achieves net token savings despite occasional minor cost increases, and SessAST introduces additional cost and inference time to handle forgetting.

#### 5.4 RQ4: Further Analysis for AST Parts

**AST-based Memory Selector.** We evaluate the effectiveness of CtxAST by measuring recall and precision of relevant code in memory. On CodeIF-Bench, which provides ground-truth contexts, CODEMEM progressively retrieves more valid contexts while steadily improving precision (Figure 3). This improvement stems from the filtering effect of CtxAST. A temporary dip in precision in the second round is caused by noisy contexts

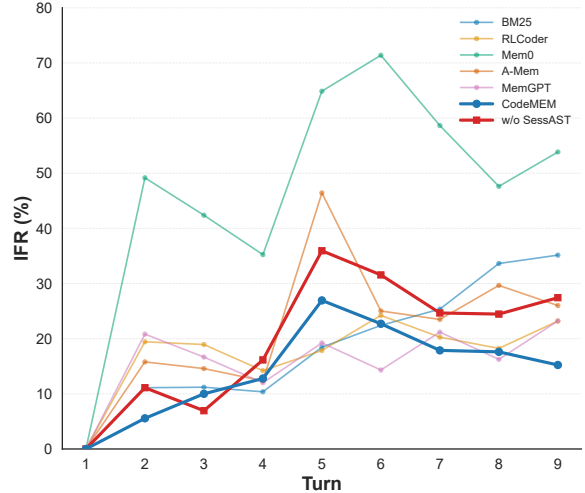


Figure 4: Forgetting rate (IFR) result on CodeIF-Bench.

during re-retrieval, but subsequent rounds show effective pruning. When CtxAST is removed and context is managed solely by the LLM, the LLM continuously retrieves and increases noise contexts, degrading generation quality and increasing both cost and inference time.

**AST-based Memory Detector.** We assess SessAST using the forgetting rate (IFR) metric on CodeIF-Bench (Figure 4). CODEMEM substantially reduces forgetting compared to baselines, with improvements more pronounced in later rounds. Mem0 shows a sharp IFR increase over successive rounds, highlighting the limitations of natural language-based memory for iterative code tasks. Removing SessAST significantly increases forgetting, underscoring its critical role in code session memory. Moreover, CODEMEM demonstrates a sustained decline in IFR in later rounds, indicating its ability to mitigate forgetting in long-horizon, iterative code generation.

**RQ4 Summary:** Further results demonstrate that CtxAST enhances the effective context recall and precision to deliver high-quality code context memory, while SessAST reduces the LLM’s forgetting rate, thereby further improving the performance of CODEMEM.

## 6 Conclusion

In this work, we propose CODEMEM, a memory management system tailored for repository-level iterative code generation. It dynamically preserves and incorporates effective code context through AST-guided Code Context Memory, and constructing code-centric Code Session Memory via AST to mitigate LLM forgetting. Experimental results

on the CodeIF-Bench and CoderEval demonstrate that CODEMEM not only surpasses state-of-the-art baselines in quality but also achieves competitive time efficiency and cost consumption. In future work, we will further optimise CODEMEM to adapt it to more complex scenarios.

## 7 Limitations

**Limited Evaluation Scenarios.** Although our evaluation employs the instruction-following benchmark (CodeIF-Bench), which closely approximates real-world scenarios, and extends the existing code generation benchmark CoderEval in line with prior work, the following limitations remain: the efficacy of our model has not been validated across broader and more complex scenarios. For instance, longer interaction rounds and more intricate conversational scenarios and larger, industrial-scale code repositories containing thousands of classes and function blocks. Constrained by the absence of such high-quality benchmarks, we shall continue to monitor and adapt CODEMEM to these benchmarks in future work.

**Biases Inherent to LLM.** CODEMEM partially relies on the LLM for decision-making and generating contextually relevant memories. This dependency on the LLM’s inherent capabilities and preference limitations may lead to erroneous decisions and hallucinations. Furthermore, Code Context Memory relies on the LLM correctly utilizing context to select relevant information. The LLM may overlook pertinent context, thereby filtering out crucial elements. In future work, we will investigate methods to enhance the LLM’s contextual utilization accuracy to mitigate this adverse effect.

## 8 Ethical Considerations

We strictly comply with the applicable licensing terms of all scientific materials and datasets used in this study, ensuring that they are used solely for non-commercial research purposes. We adhere to all data usage policies and restrictions provided by the original sources to mitigate potential legal and ethical risks. All datasets and code repositories used in our experiments are publicly available or properly licensed, and we do not include any proprietary or sensitive data. AI assistants were used only for minor grammatical corrections and improvements to sentence fluency; no scientific content or ideas were generated by AI.

## 9 Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grants Nos. 62302021 and 62577007) and the State Key Laboratory of Complex & Critical Software Environment (Grant No. CCSE2025ZX-09).

## References

- Anonymous. 2025. [Memagent: Reshaping long-context LLM with multi-conv RL-based memory agent](#). In *Submitted to The Fourteenth International Conference on Learning Representations*. Under review.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Ge Bai, Jie Liu, Xingyuan Bu, Yancheng He, Jiaheng Liu, Zhanhui Zhou, Zhuoran Lin, Wenbo Su, Tiezheng Ge, Bo Zheng, and Wanli Ouyang. 2024. [MT-bench-101: A fine-grained benchmark for evaluating large language models in multi-turn dialogues](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7421–7454, Bangkok, Thailand. Association for Computational Linguistics.
- Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie van Deursen, Maliheh Izadi, and Timofey Bryksin. 2024. [Long code arena: a set of benchmarks for long-context code models](#). *Preprint*, arXiv:2406.11612.
- Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. 2025. [Mem0: Building production-ready ai agents with scalable long-term memory](#). *Preprint*, arXiv:2504.19413.
- Yun He, Di Jin, Chaoqi Wang, Chloe Bi, Karishma Mandyam, Hejia Zhang, Chen Zhu, Ning Li, Tengyu Xu, Hongjiang Lv, and 1 others. 2024. [Multif: Benchmarking llms on multi-turn and multilingual instructions following](#). *arXiv preprint arXiv:2410.15553*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. [Qwen2.5-coder technical report](#). *Preprint*, arXiv:2409.12186.

- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. [Live-codebench: Holistic and contamination free evaluation of large language models for code](#). In *The Thirteenth International Conference on Learning Representations*.
- Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. [Llms get lost in multi-turn conversation](#). *Preprint*, arXiv:2505.06120.
- Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024a. [Evocodebench: An evolving code generation benchmark aligned with real-world code repositories](#). *Preprint*, arXiv:2404.00599.
- 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, and Mengfei Yang. 2024b. [DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3603–3614, Bangkok, Thailand. Association for Computational Linguistics.
- Zhiyu Li, Shichao Song, Hanyu Wang, Simin Niu, Ding Chen, Jiawei Yang, Chenyang Xi, Huayi Lai, Jiahao Zhao, Yezhaohui Wang, Junpeng Ren, Zehao Lin, Jiahao Huo, Tianyi Chen, Kai Chen, Kehang Li, Zhiqiang Yin, Qingchen Yu, Bo Tang, and 3 others. 2025. [Memos: An operating system for memory-augmented generation \(mag\) in large language models](#). *Preprint*, arXiv:2505.22101.
- Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. [Graphcoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph](#). In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24*, page 570–581, New York, NY, USA. Association for Computing Machinery.
- Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. [Memgpt: Towards llms as operating systems](#). *Preprint*, arXiv:2310.08560.
- Peiding Wang, Li Zhang, Fang Liu, Lin Shi, Minxiao Li, Bo Shen, and An Fu. 2025a. [Codeif-bench: Evaluating instruction-following capabilities of large language models in interactive code generation](#). *Preprint*, arXiv:2503.22688.
- Peiding Wang, Li Zhang, Fang Liu, Yinghao Zhu, Wang Xu, Lin Shi, Xiaoli Lian, Minxiao Li, Bo Shen, and An Fu. 2025b. [Efficientedit: Accelerating code editing via edit-oriented speculative decoding](#). *Preprint*, arXiv:2506.02780.
- Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024. [Mint: Evaluating llms in multi-turn interaction with tools and language feedback](#). *Preprint*, arXiv:2309.10691.
- Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2025c. [RLCoder: Reinforcement Learning for Repository-Level Code Completion](#), page 1140–1152. IEEE Press.
- Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. [Repoformer: Selective retrieval for repository-level code completion](#). *Preprint*, arXiv:2403.10059.
- Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. [A-mem: Agentic memory for llm agents](#). *Preprint*, arXiv:2502.12110.
- Hao Yu, Bo Shen, Dezhi Ran, Jiabin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. [Codereval: A benchmark of pragmatic code generation with generative pre-trained models](#). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24*, New York, NY, USA. Association for Computing Machinery.
- Zexun Zhan, Shuzheng Gao, Ruida Hu, and Cuiyun Gao. 2025. [Sr-eval: Evaluating llms on code generation under stepwise requirement refinement](#). *Preprint*, arXiv:2509.18808.
- Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. [RepoCoder: Repository-level code completion through iterative retrieval and generation](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.
- Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. [CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repository-level coding challenges](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, Bangkok, Thailand. Association for Computational Linguistics.
- Qianhui Zhao, Li Zhang, Fang Liu, Xiaoli Lian, Qiaoyuanhe Meng, Ziqian Jiao, Zetong Zhou, Jia Li, and Lin Shi. 2025. [Fastcoder: Accelerating repository-level code generation via efficient retrieval and verification](#). *Preprint*, arXiv:2502.17139.
- Wanjuan Zhong, Lianghong Guo, Qiqi Gao, and Yanlin Wang. 2023. [Memorybank: Enhancing large language models with long-term memory](#). *arXiv preprint arXiv:2305.10250*.
- Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. [Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence](#). *arXiv preprint arXiv:2406.11931*.

Table 4: Results of Qwen2.5-Coder-32B-Instruct and DeepSeek-V3.2 on CoderEval.

CoderEval	Turn-1 (↑)	Turn-2 (↑)	Turn-3 (↑)	Turn-4 (↑)	Turn-5 (↑)
FC <sub>Qwen2.5</sub>	27.6±0.3	31.0±0.3	32.4±0.2	32.8±0.2	34.7±0.3
CodeMEM <sub>Qwen2.5</sub>	<b>30.7±0.3</b>	<b>36.8±0.2</b>	<b>39.4±0.2</b>	<b>39.8±0.2</b>	<b>40.3±0.2</b>
FC <sub>DeepSeek</sub>	40.3±0.3	44.9±0.4	48.5±0.2	50.3±0.2	50.9±0.0
CodeMEM <sub>DeepSeek</sub>	<b>42.7±0.4</b>	<b>49.0±0.3</b>	<b>51.5±0.3</b>	<b>54.2±0.2</b>	<b>55.7±0.0</b>

## A Metrics Details

### Instruction Accuracy (IA) (Wang et al., 2025a):

This metric quantifies the proportion of instructions that the LLM correctly follows in each round of a dialogue. Specifically, in the  $n$ -th round, given the historical dialogue and the current instruction  $I_n$ , the LLM generates  $A_n$ . Compliance with instruction  $I_n$  is determined by whether  $A_n$  passes the corresponding test  $T_n$ :

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

### Conversation Accuracy (CA) (Wang et al., 2025a):

This metric measures the fraction of all instructions, from the first turn up to the current turn, that the LLM has successfully followed. In the  $n$ -th round, given the historical dialogue  $\{I_1, A_1, \dots, I_{n-1}, A_{n-1}\}$  and the current instruction  $I_n$ , the LLM outputs  $A_n$ . The CA score for this turn is computed by evaluating  $A_n$  against the full test sequence  $TS = \{T_1, \dots, T_n\}$ :

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

### Instruction Forgetting Ratio (IFR) (Wang et al., 2025a):

This metric captures the proportion of previously followed instructions that the LLM fails to follow in later turns. An instruction is considered forgotten if it was satisfied in one of the previous rounds (1, 2, ...,  $n - 1$ ) but not in the current round  $n$ . Let  $PTS = \{T'_1, \dots, T'_k\}$  denote the tests passed in previous rounds. Then, given  $A_n$ , IFR is computed as:

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

**Pass@k (Chen, 2021):** This metric evaluates the LLM’s performance by executing multiple generated programs per instruction:

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

where  $k$  is the number of programs generated by the LLM for a given instruction,  $c$  is the number of programs that pass the tests, and  $n$  is the total number of programs generated.

## B Cross-Model Generalization Validation

Table 5: Results of Qwen2.5-Coder-32B-Instruct and DeepSeek-V3.2 on CodeIF-Bench.

CodeIF-Bench	Avg. IA (↑)	Avg. CA (↑)
FC <sub>Qwen2.5</sub>	27.5±0.6	29.2±0.4
CodeMEM <sub>Qwen2.5</sub>	<b>32.8±0.6</b>	<b>30.6±0.4</b>
FC <sub>DeepSeek</sub>	40.9±0.2	38.4±0.2
CodeMEM <sub>DeepSeek</sub>	<b>46.0±0.2</b>	<b>42.7±0.3</b>

To evaluate cross-model generalization, we replicated our experiments three times using the code model Qwen2.5-Coder-32B-Instruct(Hui et al., 2024) under the same experimental settings described in the paper. The Table 4 and 5 show that CodeMEM consistently improves performance on both CoderEval and CodeIF-Bench, with p-values bellowing 0.05, indicating statistically significant gains. These findings confirm that CodeMEM generalizes effectively across different model architectures and scales.

## C Data Example

### Example of CodeIF-Bench

```
{
  namespace: "boltons.socketutils.NetstringSocket.setmaxsize",
  project_path: "Utilities/boltons",
  completion_path: "Utilities/boltons/boltons/socketutils.py",
  prompt: "Set the maximum size for receiving netstrings in the NetstringSocket instance. It updates the maxsize of the instance and calculates the maximum size for a netstring message based on the new maxsize value..."

  requirement: {
    Input-Output Conditions: {
      requirement: "The 'setmaxsize' function should accept an integer 'maxsize' parameter and update the instance's 'maxsize' attribute accordingly...",
      test: "tests/test_socketutils.py::test_setmaxsize_updates_attributes"
    },
    Exception Handling: {
      requirement: "The 'setmaxsize' function should raise a ValueError if the 'maxsize' parameter is not a positive integer or zero.",
      test: "tests/test_socketutils.py::test_setmaxsize_raises_valueerror_on_invalid_maxsize"
    },
    Edge Case Handling: {
      requirement: "The setmaxsize method should correctly handle setting the maximum size to zero and ensure that any non-empty netstring payloads cause a NetstringMessageTooLong exception.",
      test: "tests/test_socketutils.py::test_setmaxsize_zero_behavior"
    },
    Functionality Extension: {
      requirement: "Extend the 'setmaxsize' function to print a message: 'Maxsize set to new_maxsize' indicating the change in 'maxsize' for debugging purposes.",
      testtest: "tests/test_socketutils.py::test_setmaxsize_logs_message"
    },
    Annotation Coverage: {
      requirement: "Ensure that the 'setmaxsize' function includes type annotations for its parameters and return type, including one parameters: 'maxsize': int, and return type: None.",
      test: "tests/test_socketutils.py::test_setmaxsize_annotations"
    },
    Code Complexity: {
      requirement: "The 'setmaxsize' function should maintain a cyclomatic complexity of 1, indicating a simple, linear function.",
      test: "tests/test_socketutils.py::test_setmaxsize_complexity"
    },
    Code Standard: {
      requirement: "The 'setmaxsize' function should adhere to PEP 8 standards, including proper indentation, line length, and spacing.",
      test: "tests/test_socketutils.py::test_check_code_style"
    },
    Context Usage Verification: {
      requirement: "The 'setmaxsize' function should utilize the '_calc_msgsize_maxsize' method to update '_msgsize_maxsize'.",
      test: "tests/test_socketutils.py::test_setmaxsize_uses_calc_msgsize_maxsize"
    },
    Context Usage Correctness Verification: {
      requirement: "Verify that the '_msgsize_maxsize' is correctly updated based on the new 'maxsize' using '_calc_msgsize_maxsize'.",
      test: "tests/test_socketutils.py::test_setmaxsize_updates_attributes"
    }
  }
}
```

### Example of CoderEval in Our Experiment

```
{
  _id: "62e60f43d76274f8a4026e28",
  file_path: "neo4j/_codec/hydration/v1/temporal.py",
  project: "neo4j/neo4j-python-driver",
  prompt: "Please finish the following code: def hydrate_time(nanoseconds, tz=None): \"\"\"for 'Time' and 'LocalTime' values...\"\"\"",
  feedback_prompt: "Your answer is incorrect. Please regenerate."
}
```

## D Memory Example

### Example of Code Context Memory

```
{
  memory_key: {
    class_signature: "class NetstringSocket(object):
Reads and writes using the netstring protocol
(see Wikipedia and protocol specification)...",
    class_attributes: [ "_msgsize_maxsize", "bsock", "maxsize", "timeout" ],
    class_methods: [ "__init__", "fileno", "settimeout", "read_ns", "write_ns" ]
  },
  memory_value: "class NetstringSocket(object):
def __init__(...):
self.bsock = BufferedSocket(...)
...
def read_ns(...):
..."
}
```

### Example of Code Session Memory

```
{
  boltons.socketutils.NetstringSocket.setmaxsize: [
    {
      id: 0,
      instruction: "Please write a python function called 'setmaxsize' base the context...",
      code: "def setmaxsize(self, maxsize): self.maxsize = maxsize self._msgsize_maxsize =
self._calc_msgsize_maxsize(maxsize)"
      note: "Function correctly implemented setmaxsize method.",
      diff_nodes: {
        added: [ ],
        removed: [ ]
      },
      state_links: [ ]
    },
    ...
    {
      id: 2,
      instruction: "The 'setmaxsize' function should raise a 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: "The answer correctly implements the setmaxsize function as specified, raising a
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: [ ]
    },
  ]
}
```

## E Prompts

### The Code Context Memory Judge Prompt.

**You are an expert repository memory manager for repository code generation tasks.**

Your goal is to decide whether the current repository code context memory needs updating based on the user's programming instructions.

#### Decision Objective

Decide if you need to modify the repository memory (**Existing Repository Context**) based on how well it already covers the entities mentioned in the user instructions.

#### Modes (Mutually Exclusive)

- **KEEP** — Use this mode when the existing repository context already contains all relevant classes/functions to understand or execute the instruction.
- **ADD** — Use this mode when Existing Repository Context lacks code context related to user instructions.

#### User Instructions:

```
{instructions}
```

#### Existing Repository Context:

```
{existing_repository_context}
```

#### Output Format (strict JSON)

```
{  
  "mode": "<ADD | KEEP>",  
  "action": "<short, specific description of what to update or not update>",  
  "target_context": "<list of relevant namespaces or []>"  
}
```

### The Generation Prompt with Memory.

**You are an expert repository-level code generator.**

Your goal is to generate the correct function implementation by leveraging the provided repository context and historical memory blocks.

#### Repo Context

```
{repo_context}
```

#### Memory Blocks

```
{memory_blocks}
```

#### Current Instruction

```
{instruction}
```

#### Output Requirement

Please output the correct function implementation.

## F Case Study

```
#####Turn-1 #####
#User-1
Instruction:
Please write a python function called 'setmaxsize'. Set the maximum size for receiving netstrings.
Code Context:
class BufferedSocket...
class NetstringMessageTooLong...
class Sentinel...
class NetstringSocket...
#LLM-1 Answer
def setmaxsize(self, maxsize):
    self.maxsize = maxsize
    self._msgsize_maxsize = self._calc_msgsize_maxsize(maxsize)
#####Turn-2 #####
#User-2
Instruction:
The 'setmaxsize' function should raise a ValueError.
Code Context:
class NetstringSocket... (CtxMem: The context is sufficient; no update is needed.)
#LLM-2 Answer
def setmaxsize(self, maxsize):
    if not isinstance(maxsize, int) or maxsize < 0:
        raise ValueError("maxsize must be a non-negative integer")
    self.maxsize = maxsize
    self._msgsize_maxsize = self._calc_msgsize_maxsize(maxsize)
#####Turn-6#####
#User-6
Instruction:
The 'setmaxsize' function should adhere to PEP 8 standards.
Code Context:
class NetstringSocket...
#LLM-6 Init Answer
def setmaxsize(self, maxsize):
    # Set the maximum size for receiving netstrings (missing ValueError)
    self.maxsize = maxsize
    self._msgsize_maxsize = self._calc_msgsize_maxsize(maxsize)
#LLM-6 Correct Answer
def setmaxsize(self, maxsize):
    # Set the maximum size for receiving netstrings
    if not isinstance(maxsize, int) or maxsize < 0:
        raise ValueError("maxsize must be a non-negative integer")
    self.maxsize = maxsize
    self._msgsize_maxsize = self._calc_msgsize_maxsize(maxsize)
```

Figure 5: The example of success cases where CodeMEM preserves the valid 'NetstringSocket' class through CtxMem, effectively removing noisy context and reducing token cost. Meanwhile, SessMem detects forgetting phenomena during interaction and successfully corrects regressive edits.