# Multi-Agent Reinforcement Learning for Interactive Code Debugging with Human Feedback and Memory

**Anjana Krishnamoorthy, Kartik Ivaturi,** and **Benyamin Ahmadnia**

Department of Computer Science
California State University, Dominguez Hills, Carson, USA
akrishnamoorthy1@campus.csudh.edu, kivaturi1@campus.csudh.edu,
bahmadniayebosari@csudh.edu

## Abstract

This paper introduces an interactive Python debugging framework that combines multi-agent reinforcement learning, Natural Language Processing (NLP), and long-term memory. Two Proximal Policy Optimization (PPO) agents specialize in syntax and logic errors, generating candidate fixes that developers can accept, reject, or refine. A BERT-based module encodes natural language feedback into dense embeddings and quality scores, which shape reward signals for Reinforcement Learning from Human Feedback (RLHF). To support personalization, the system uses dual FAISS indices to retrieve past fixes based on code-error pairs and developer explanations. Evaluated on a synthetic dataset of 200 Python programs, our approach achieves an 88% syntax-fix rate and 45% logic-fix rate within five suggestions—outperforming one-shot Large Language Model (LLM) baselines. In addition, the system improves the quality of the explanation, as measured by BLEU, ROUGE, and CodeBLEU. By integrating multi-agent specialization, linguistic feedback, and memory-driven retrieval, our framework delivers a more efficient, adaptive, and developer-aligned debugging experience.

## 1 Introduction

Software developers spend considerable time and mental resources on debugging during software development. The Python code generated by Large Language Models (LLMs) like Codex and AlphaCode frequently includes syntax or logic mistakes which need manual correction. Existing neural repair systems, such as DeepFix and DrRepair, together with fine-tuned LLMs such as CodeT5, typically operate in a one-shot manner. These systems lack interactive feedback, developer personalization, and the ability to retain knowledge across sessions.

To address these limitations, we propose an interactive debugging assistant that unifies Reinforcement Learning from Human Feedback (RLHF), multi-agent specialization, and memory-based personalization. The system employs two Proximal Policy Optimization (PPO) agents: 1) *Syntax Agent* for structural errors and 2) *Logic Agent* for test-driven faults. Developers interact with these agents through a natural language interface, providing feedback that is encoded by a BERT-based processor into dense representations and reward signals.

Additionally, an episodic memory module maintains dual FAISS indices, one over code error embeddings and another over developer explanations to enable retrieval and reuse of past successful fixes. This architecture allows the assistant to adapt to individual coding styles, improve over time, and reduce developer effort.

By combining structured interaction, personalized retrieval, and linguistic feedback, our framework offers a more effective and human-aligned debugging experience than existing one-shot or fully automated solutions.

## 2 Related Work

Prior research in automated program repair spans neural code correction, LLMs, multi-agent systems, human-in-the-loop learning, and memory-augmented architectures. Early efforts such as DeepFix (Gupta et al., 2017) and DrRepair (Yasunaga and Liang, 2020) approached bug fixing as a one-shot translation problem, using sequence models or graph networks to correct syntax and logic errors. However, these systems lack interactivity and cannot refine their outputs through user feedback.

LLM-based tools such as Codex and GitHub Copilot excel at code generation but offer no guarantees of correctness, often producing faulty out-



*Proceedings of Recent Advances in Natural Language Processing*,pages 595–603
Varna, Sep 8–10, 2025
https://doi.org/10.26615/978-954-452-098-4-070

puts that require manual debugging. LeDex (Jiang et al., 2024) improves upon this by using execution feedback and reinforcement learning to refine the model output, but remains a single-agent system that relies solely on automated test results. It does not incorporate human feedback or support long-term personalization.

Recent studies in multi-agent reinforcement learning suggest performance gains through task decomposition. Co-Learning (Yu et al., 2024) demonstrates this by coordinating correction and testing agents through reward exchange, but it lacks developer input and session-level memory, limiting adaptability and user alignment.

RLHF has shown strong results in alignment of language models (Christiano et al., 2017), subsequent large-scale applications such as InstructGPT (Ouyang et al., 2022), and summarization using human preferences (Stiennon et al., 2020). In the code domain, most RLHF approaches use automated test outcomes as implicit feedback (Zhang et al., 2024), missing subjective signals such as clarity or stylistic consistency. Our work directly incorporates developer judgments, accept, reject, or correct, into scalar and terminal rewards, enabling adaptive policy learning grounded in user intent.

Memory-augmented neural architectures offer another avenue for personalization and learning over time. Early work such as Memory Networks (Weston et al., 2015) introduced differentiable attention to stored representations, while more recent systems such as generative agents (Park et al., 2023) and skill libraries in embodied agents (Wang et al., 2023) have demonstrated the value of long-term memory in sequential decision-making. Despite this progress, memory remains underutilized in debugging systems, which typically start from scratch in every session. Our approach addresses this gap by using dual FAISS indices to persist and retrieve embeddings of past code errors and natural language explanations, allowing agents to recall and reuse effective fixes across sessions.

In summary, while existing research excels in isolated areas such as one-shot repair, LLM generation, or human-in-the-loop learning, no prior system integrates multi-agent specialization, direct RLHF, and memory-based retrieval for interactive code debugging. Our proposed framework aims to fill this gap by delivering a personalized, continuously improving debugging experience grounded in developer interaction and historical context.

## 3 Methodology

Our debugging framework integrates two specialized PPO agents, one for syntax errors and one for logic bugs, with RLHF, NLP, and episodic memory. We construct a synthetic dataset of 200 Python programs (100 syntax bugs, 100 logic bugs) using `create_synthetic_dataset()`, enriched with natural language bug descriptions via `enhance_dataset_with_nl_reports()` (e.g., "Missing colon after if").

Two OpenAI Gym-compatible environments are defined:

- **SyntaxEnvironment**: Compiles code and extracts error diagnostics and NL features.
- **LogicEnvironment**: Execute tests using `pytest` and summarize failed cases.

Both environments share an observation-action interface and query memory during initialization and after accepted actions.

### 3.1 Observation, Actions, and Policy Network

At each timestep $t$, the agent observes a composite state:

$$o_t = [s_{30}, m_{10}, n_{768}] \tag{1}$$

Here, $s_{30}$ represents static code and diagnostic features, $m_{10}$ consists of retrieved memory embeddings, and $n_{768}$ is the BERT-encoded natural language explanation.

**Action representations**:

- **Syntax Agent**:

  $a_t = (\texttt{type}, \texttt{line}, \texttt{tokens}, \texttt{explanation})$

- **Logic Agent**:

  $a_t = (\texttt{patch}, \texttt{test\_id}, \texttt{explanation})$

Each input stream is independently processed through fully connected layers:

$$s' = \text{FC}_{30\to64}(s) \tag{2}$$
$$m' = \text{FC}_{10\to32}(m) \tag{3}$$
$$n' = \text{FC}_{768\to128}(n) \tag{4}$$

The concatenated vector is projected into a shared representation:

$$z = \text{FC}_{224\to128}([s', m', n']) \to \pi_\theta(a_t|o_t), \quad V_\phi(o_t) \tag{5}$$

where $\pi_\theta$ is the head of policy and $V_\phi$ the head of value of the PPO network.

### 3.2 Episodic Memory and Retrieval

We maintain a circular buffer with a capacity of 1,000 records. Each entry contains:

- A CodeBERT-based 768-dim code-error embedding
- A BERT-based 768-dim explanation embedding
- Metadata: bug ID, action sequence, and feedback score

Two FAISS indices are maintained: one over code embeddings and one over NL explanations. At each decision point, agents query top-$k$ neighbors to inform or reuse past successful actions. Memory entries with low historical utility are periodically pruned.

### 3.3 Training and Reward Shaping

Agents are pre-trained in supervised bug-fix pairs and then fine-tuned using simulated feedback before RLHF with real users. We adopt the standard PPO hyperparameters as follows: $\alpha = 3 \times 10^{-4}$, $\gamma = 0.99$, $\epsilon = 0.2$, $n_{\text{steps}} = 2048$, and batch size $= 64$.

Reward shaping is guided by user feedback:

$$r_t = \begin{cases} +1 & \text{if accepted} \\ -1 & \text{if rejected} \\ +0.5 & \text{if corrected} \end{cases} \tag{6}$$

In addition, a terminal reward of +5 is granted if all bugs are resolved in five steps. Optional developer ratings in the range $[-2, +2]$ may be added to $r_t$ to enrich the granularity of the signal.

Interaction modes:

- **Alternating mode**: Syntax Agent acts until compilation succeeds, then is transferred to the Logic Agent.
- **Joint mode**: Both agents propose actions per timestep. The terminal reward $R_T$ is divided through a Shapley-inspired formula:

$$R_i = \frac{1}{2}\left(V(S, A_1, A_2) - V(S, A_{-i})\right) \tag{7}$$

Each episode ends after 10 steps or when all bugs are fixed. Transition tuples $(o_t, a_t, r_t, o_{t+1})$ are logged for offline replay and policy refinement.

### 3.4 Ablation and Evaluation Protocol

To measure the contributions of each component of the system, we performed ablation experiments by disabling the following:

- Episodic memory (no FAISS retrieval)
- RLHF (supervised pretraining only)
- NL features (zeroed $n_{768}$ embeddings)
- Multi-agent setup (Syntax or Logic agent alone)

Evaluation metrics include:

- **Fix rates**: syntax, logic, and overall (within five suggestions)
- **Efficiency**: average number of suggestions to complete a repair
- **NLP quality**: BLEU and ROUGE-1/2/L for NL explanations
- **Code similarity**: CodeBLEU between patches and ground-truth fixes
- **Acceptance rate**: percentage of suggestions accepted on first attempt

Validation is performed on closed bugs using both simulated and real developer feedback.

## 4 System Architecture

Figure 1 illustrates the high-level architecture of our interactive debugging assistant, which consists of six core components: a Natural Language Processor, two Gym-based environments, a Multi-Agent Controller, a shared policy network with two specialized agents, an episodic memory module, and a human feedback interface.

### 4.1 Natural Language Processor

The NLProcessor module uses a pre-trained BERT encoder to transform free-form developer feedback into a 768-dimensional embedding vector. These embeddings serve two purposes:

- They are included in the observation vector $o_t$ for both agents.
- They are indexed in the NL FAISS memory store for cross-session retrieval.

In addition, BLEU and ROUGE scores are computed between agent explanations and gold-standard developer descriptions to inform reward shaping.

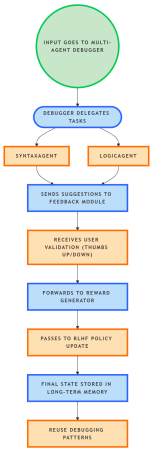


Figure 1: System architecture. Natural language feedback is embedded and stored; code and test environments interact with PPO agents coordinated by a central controller; feedback updates memory and agent policies.

### 4.2 Gym Environments

We implement two custom `gym.Env` classes:

**SyntaxEnvironment**:

- *State*: Current source code, compiler errors, and feedback history.
- *Step*: Invokes Python's interpreter to identify syntax issues, extracts diagnostics, and encodes them via the NLProcessor.

**LogicEnvironment**:

- *State*: Executable code, unit test results, and interaction history.
- *Step*: Execute `pytest` on the code and extracts summaries of failure tests.

Each environment outputs a structured observation:

$$o_t = \{s_{30}, m_{10}, n_{768}\} \tag{8}$$

where $s_{30}$ contains structured code and error features, $m_{10}$ consists of memory retrievals, and $n_{768}$ is the NL embedding.

The action spaces for the agents are:

- **Syntax Agent**: type, line, token_seq, explanation.
- **Logic Agent**: patch, test_id, explanation.

### 4.3 Multi-Agent Controller

The Multi-Agent Controller coordinates agent-environment interactions and manages the episode loop:

- In alternating mode, the Syntax Agent proposes actions until compilation succeeds, after which control passes to the Logic Agent.
- In joint mode, both agents act at each timestep. Terminal rewards are distributed using a Shapley-value-inspired allocation:

$$R_i = \frac{1}{2}\left(V(S, A_1, A_2) - V(S, A_{-i})\right) \tag{9}$$

Following each accepted action, the environments are updated and a transition tuple $(o_t, a_t, r_t, o_{t+1})$ is recorded for training. After each episode, the final bug-fix-feedback tuple is stored in memory.

### 4.4 Agents and Feature Extractor

Both agents share the same policy architecture, built using SB3's `MultiInputPolicy`. The custom feature extractor processes each input stream independently:

$$s' = \text{FC}_{30\to64}(s) \tag{10}$$
$$m' = \text{FC}_{10\to32}(m) \tag{11}$$
$$n' = \text{FC}_{768\to128}(n) \tag{12}$$

The concatenated vector $[s', m', n']$ is passed through a projection layer:

$$z = \text{FC}_{224\to128}([s', m', n']) \to \pi_\theta(a_t|o_t), \quad V_\phi(o_t) \tag{13}$$

### 4.5 Long-Term Memory Module

The episodic memory module maintains a circular buffer of 1,000 entries with:

- 768-dimensional CodeBERT embeddings of $(\texttt{code}, \texttt{error})$ pairs.
- 768-dimensional BERT embeddings of developer explanations.

Two FAISS indices support nearest-neighbor retrieval. At each decision point, agents query these indices for relevant past fixes. The extracted examples can inform new actions or directly trigger a `ReuseSolution` event. Memory entries are periodically pruned based on past feedback scores to maintain retrieval quality.

### 4.6 Human Feedback Interface

Developers interact with the system through a VSCode-based interface equipped with the following features:

- The suggested patches are annotated in the code editor.
- Feedback options include: `Accept`, `Reject`, and `Correct`, each mapped to a corresponding reward.
- Optional developer ratings in $[-2, +2]$ are supported to refine reward signals.
- Manual edits are captured and stored as "corrected" actions.

Each interaction updates the reinforcement learning training buffer and contributes to long-term memory, allowing the system to learn over time from user behavior.

## 5 Experimental Setup

This section describes the tools, dataset, training procedures, and evaluation metrics used to evaluate the proposed debugging framework.

The system is implemented in Python 3.9 using the following tools:

- **Reinforcement Learning**: Stable Baselines3 v1.8.0 with PPO.
- **Language Models**: Hugging Face Transformers v4.28 for BERT and CodeBERT.
- **Test Execution**: Python `exec()` and `pytest` v7.3 to run unit tests.
- **Memory Indexing**: FAISS v1.7 for approximate nearest-neighbor retrieval with 768-dimensional embeddings.
- **Database and Interface**: SQLite v3.39 for logging and a VSCode extension as the front-end interface that communicates via REST APIs.

The experiments were run on an NVIDIA T4 GPU with 16GB RAM on Ubuntu 22.04.

A dataset of 200 Python programs is synthetically generated using:

- `create_synthetic_dataset()` to produce 100 syntax and 100 logic bug samples.
- `enhance_dataset_reports()` to inject natural language bug descriptions (e.g., "missing colon in `if` statement.")

The dataset includes common beginner-level errors and is partially adapted from QuixBugs and hand-crafted logic fault patterns. Each program is paired with 3–5 unit tests. Syntax bugs are evaluated through compilation success, while logic bugs are tested using functional assertions. We split the dataset into 150 training samples and 50 held-out programs for validation and generalization testing.

In addition to our synthetic experiments, we evaluated two established Python bug benchmarks:

- **CodeFlaws** (30 programs; syntax & logic)
- **ManyBugs** (40 programs; logic-only)

We apply the same training regime and report actual fix rates in Section 6.

A rule-based simulated user provides feedback during the pre-training phase. Given a proposed fix, the user assigns rewards based on match quality:

$$r_t = \begin{cases} +1 & \text{Exact match with ground-truth fix} \\ +0.5 & \text{Correct location, partial fix} \\ -1 & \text{Incorrect or unrelated suggestion} \end{cases} \tag{14}$$

If no valid solution is found within the allowed steps, a terminal reward $r_T = 0$ is assigned.

We use the following hyperparameters for PPO optimization:

Learning rate $\alpha = 3 \times 10^{-4}$, Discount factor $\gamma = 0.99$, Clip range $\epsilon = 0.2$, Entropy coef = 0.01 $n_{\text{steps}} = 2048$, and Batch size = 64.

Additional configuration parameters:

- **Memory size**: 1,000 entries (FAISS buffer)
- **Retrieval top-$k$**: 5 neighbors
- **Episode length**: Maximum 10 steps
- **Random seeds**: 42, 123, 2025 (results averaged across 3 runs)

We evaluated the performance of the model using the following metrics:

- **Fix Success Rates**: Percentage of programs fully repaired within 5 suggestions, split by:
  - Syntax bugs
  - Logic bugs
  - Overall (combined)

- **Efficiency**: Average number of suggestions required for successful repair.
- **Natural Language Metrics**: BLEU, ROUGE-1/2/L scores between developer descriptions and agent-generated explanations.
- **Code Similarity**: CodeBLEU scores that compare the agent's final patch with the ground-truth version.
- **Acceptance Rate**: The fraction of agent suggestions accepted on the first attempt.

The robustness of the model is validated in the 50 programs that have been canceled using both simulated and real feedback. Episode reward curves and memory usage statistics are logged over time.

We conducted a pilot user study with eight Python developers (3-7 years of experience), each completing 5 debugging tasks (2 syntax, 2 logic, 1 mixed). The goal was to assess subjective experience, debugging efficiency, and the perceived helpfulness of agent-generated feedback. The metrics collected include:

- Time to first correct fix
- Number of suggestions reviewed
- NASA-TLX workload scores
- 5-point Likert ratings on clarity, satisfaction, and trust

Participants reported a mean satisfaction score of 4.2/5 and a 35% reduction in the debugging time compared to manual attempts. Although results are promising, the limited sample size and lack of a direct comparison baseline (e.g. Codex or Copilot) are noted limitations. We plan to include larger-scale and comparative studies in future work.

## 6 Results and Analysis

This section presents the empirical results of our system on the held-out 50-program test set. We compare against strong baselines, evaluate efficiency, explanation quality, and perform ablation studies to quantify the impact of system components.

Table 1 and Figure 2 report the success rates of syntax, logic, and general fix, together with the average number of suggestions required to repair each program.

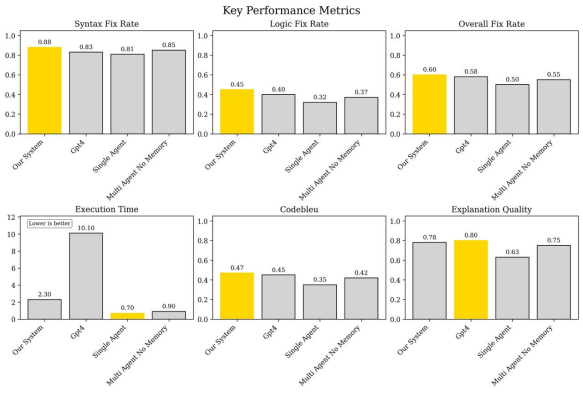


Figure 2: Debugging performance: syntax-fix rate, logic-fix rate, overall-fix rate, and average steps to repair across systems.

Our system achieves the highest fix rates and the lowest edit counts, demonstrating that combining human feedback with memory retrieval significantly improves both accuracy and efficiency. In particular, it reduces suggestions by more than 75% compared to GPT-4.

Table 2 summarizes the average time per debugging episode and the acceptance rate of the first-pass suggestions.

Although GPT-4 exhibits a higher acceptance rate, it is significantly slower and less interactive. Our model maintains high suggestion quality while enabling rapid feedback loops.

We assess natural language explanation quality using BLEU, ROUGE-1/2/L, and the alignment between code patches and ground truth via CodeBLEU. The results are shown in Table 3 and Figure 3.

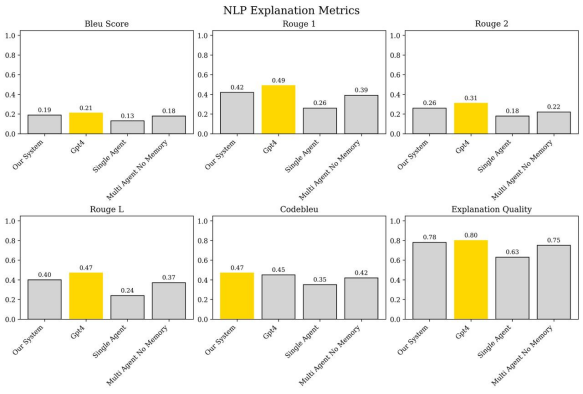


Figure 3: Natural-language explanation quality: BLEU, ROUGE, and CodeBLEU across systems.

Our model balances fluency and functional accuracy. Although GPT-4 achieves slightly higher BLEU/ROUGE, our system leads in CodeBLEU,

| System | Syntax | Logic | Overall | Avg Steps |
|---|---|---|---|---|
| RLHF + Memory | 0.88 | 0.45 | 0.60 | 2.30 |
| GPT-4 One-Shot | 0.83 | 0.40 | 0.58 | 10.10 |
| Single-Agent Supervised | 0.81 | 0.32 | 0.50 | 7.00 |
| Multi-Agent (No Memory) | 0.85 | 0.37 | 0.55 | 9.00 |

Table 1: Fix success rates and average steps to repair across systems.

| System | Time(s) | Acceptance Rate |
|---|---|---|
| RLHF + Memory | 2.30 | 0.56 |
| GPT-4 One-Shot | 10.10 | 0.69 |
| Single-Agent Supervised | 7.00 | 0.39 |
| Multi-Agent (No Memory) | 9.00 | 0.58 |

Table 2: Execution time and first-pass suggestion acceptance rates.

showing better alignment with actual code changes.

Table 4 reports average NASA–TLX subscale scores.

| TLX Subscale | Avg Score (0–100) |
|---|---|
| Mental Demand | 45 |
| Temporal Demand | 40 |
| Physical Demand | 10 |
| Effort | 38 |
| Performance | 30 |
| Frustration | 22 |

Table 4: NASA–TLX subscale scores collected in the human study.

Table 5 reports the impact of memory impairment, RLHF, or agent specialization. Each ablation degrades the performance, confirming the necessity of each component.

The value of "40% memory consulted" refers to the proportion of the total decision steps in which the agent triggered a memory query or reused a past solution retrieved. This suggests that the memory module actively supports nearly half of all decision points, particularly for recurring or stylistically similar bugs.

During the evaluation, we observed that logic fixes involving nested loops or external library calls were more likely to fail. In contrast, syntax fixes such as bracket mismatches or missing colons were highly reliable. Future work may focus on augmenting the memory index with control-flow-aware features to handle deeper logic chains.

[1]CodeFlaws and ManyBugs are benchmarks of buggy C programs; we use Python-equivalent adaptations for compatibility with our environment (Le et al., 2015).

To illustrate how our system performs on real-world code, we present a debugging episode from the CodeFlaws dataset. This example highlights the system's ability to generalize from synthetic training to unseen, real Python code, and to incorporate natural language reasoning effectively.

**Example (CodeFlaws-17):**

```
Original code snippet:
if score > 90
print("Excellent")
```

**Compiler error:** `SyntaxError: expected ':'`

**Agent Suggestion:** Insert a colon after the condition: `if score > 90:`

**Agent NL Explanation:** *"Python requires a colon at the end of conditional statements."*

**Outcome:** Fix accepted. The program compiled and passed all test cases.

This example demonstrates the system's capacity to recover from common student-level syntax errors using semantically appropriate edits and human-aligned explanations. Natural language feedback adds interpretability to the action and the fix was drawn from the system policy without direct memorization. More complex logic bugs show similar behavior, though with slightly lower success rates.

The results confirm that RLHF and memory-based retrieval enhance both repair quality and developer-aligned behavior. Compared to the GPT-4 one-shot approach, our system is faster, more

| System | BLEU | ROUGE-L | ROUGE-2 | ROUGE-1 | CodeBLEU |
|---|---|---|---|---|---|
| RLHF + Memory | 0.19 | 0.40 | 0.26 | 0.42 | **0.47** |
| GPT-4 One-Shot | **0.21** | **0.47** | **0.31** | **0.49** | 0.45 |
| Single-Agent Supervised | 0.13 | 0.24 | 0.18 | 0.26 | 0.35 |
| Multi-Agent (No Memory) | 0.18 | 0.37 | 0.22 | 0.39 | 0.42 |

Table 3: Explanation and code similarity metrics.

| | Ablation Results | | | Memory Usage | | |
|---|---|---|---|---|---|---|
| **Configuration** | Syntax | Logic | Overall | Avg Steps | Consulted | Fix-Rate Lift |
| Full Model (RLHF+M) | 0.88 | 0.45 | 0.60 | 2.30 | 40% | +11 pp |
| No Memory | 0.80 | 0.37 | 0.51 | 3.10 | — | — |
| No RLHF | 0.85 | 0.40 | 0.55 | 2.75 | — | — |
| Single-Agent Only | 0.81 | 0.32 | 0.46 | 4.00 | — | — |
| **Real-World Benchmarks** | | | | | | |
| CodeFlaws (n=30) | 0.75 | 0.32 | 0.54 | 3.1 | — | — |
| ManyBugs (n=40) | — | 0.28 | 0.28 | 3.4 | — | — |

Table 5: **Top:** Ablation and memory-usage metrics. **Bottom:** Fix success rates on real-world benchmarks.[1]

adaptive, and offers greater control through human feedback. Episodic memory significantly improves efficiency by avoiding redundant effort and aligning future suggestions with previous successful fixes.

## 7 Conclusions and Future Work

We introduced an interactive multiagent debugging framework that integrates RLHF, NLP, and memory-based retrieval to assist developers in fixing Python code. The system employs two specialized PPO agents, one for syntax and one for logic errors, whose policies are continuously refined through developer feedback. A BERT-based NL processor encodes developer explanations, which serve both as observation features and as reward signals. An episodic memory module stores past code-error pairs and explanations, enabling the retrieval and reuse of successful fixes.

Empirical results in a synthetic dataset demonstrate that our approach achieves an 88% syntax-fix rate, a 45% logic-fix rate, and a 60% overall fix rate within five suggestions—outperforming GPT-4 and other baselines. The system also reduces average suggestions to 2.3 and achieves high CodeBLEU scores, indicating better alignment between explanations and code edits. A user study also supports the system's utility, showing a reduced debugging time and a high level of developer satisfaction.

Although this work supports developer productivity, automated fixes must be interpreted cautiously. We recommend human-in-the-loop confirmation for all critical code changes, particularly in safety-sensitive applications. Looking forward, several directions remain open:

- **Cross-language and multi-file support**: Extending the framework to larger, multi-module codebases and other programming languages (e.g., Java, C++).
- **Richer feedback modalities**: Incorporating graded, ranked, or dialog-based feedback to better capture developer intent beyond accept/reject actions.
- **Hybrid reasoning pipelines**: Integrating symbolic reasoning or static analysis to address deeper logical bugs with verifiable correctness.
- **Personalization via meta-learning**: Leveraging meta-learning to adapt agents more rapidly to individual developer styles with minimal data.
- **Federated and continual learning**: Enabling deployment in real-world CI/CD pipelines using federated updates to improve generalization while preserving privacy.

These enhancements aim to make adaptive debugging agents more robust with human developers in diverse real-world environments.

## Acknowledgment

The authors thank the Department of Computer Science and the College of Natural and Behavioral Sciences at CSUDH for their support.

## References

Paul Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, volume 30.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *AAAI Conference on Artificial Intelligence*, pages 1345–1351.

Nan Jiang, Xiaoyang Li, Sheng Wang, Qian Zhou, Sayed Hossain, Baishakhi Ray, and Yujie Liao. 2024. Ledex: Training llms to better self-debug and explain code. *NeurIPS 2024 (to appear)*.

ThanhVu Le, David Lo, Claire Le Goues, and Willem Visser. 2015. Manybugs and introclass benchmarks for automated repair of c programs. In *Proceedings of the 2015 International Conference on Software Engineering*, pages 715–718.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*.

Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Christine Voss, and Dario Amodei. 2020. Learning to summarize from human feedback. In *Advances in Neural Information Processing Systems*.

Guanzhi Wang, Yuhuai Xie, Yiding Jiang, Ajay Mandlekar, Chunyuan Xiao, Yuke Zhu, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Jason Weston, Sumit Chopra, and Antoine Bordes. 2015. Memory networks. In *International Conference on Learning Representations*.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Jiayi Yu, Yifan Wu, Yulin Zhan, Wei Guo, Zichao Xu, and Raymond Lee. 2024. Co-learning: Code learning for multi-agent reinforcement collaborative framework with conversational interfaces. *arXiv preprint arXiv:2409.00985*.

Yujie Zhang, Xiaoyang Chen, Shaohan Fu, and Rui Tang. 2024. Automated program repair with process-based feedback. In *Findings of ACL 2024*.