MEMGRAD: A MEMORY-GUIDED OPTIMIZATION OF AGENTIC SOFTWARE DEVELOPMENT VIA ABSTRACTED TEXTUAL GRADIENTS

Anish Natekar, Ashutosh Ranjan, Vivek Srivastava & Shirish Karande
TCS Research, India
{anish.natekar,ashutosh.ranjan2}@tcs.com
{srivastava.vivek2,shirish.karande}@tcs.com

ABSTRACT

Agentic systems built on large language models increasingly operate in settings that demand stable reasoning, effective collaboration, and reliable adaptation. Existing optimization methods offer valuable signals through prompting strategies, alignment techniques, decentralized coordination, and experiential retrieval, but they do not translate feedback gathered across multiple trajectories into persistent improvements in agent behavior. We introduce ***MemGrad***, a memory-guided optimization framework that uses textual gradients to transform batches of behavioral feedback into coherent and interpretable improvement directions. These gradients support a retrospective–prospective memory structure: retrospective memory captures recurring patterns and common failure modes, while prospective memory encodes gradient-derived strategies that guide future reasoning and coordination. The framework also updates system prompts so that agents internalize these improvements without model fine-tuning. Applied to AgileCoder, a multi-agent software development framework, our approach improves task success, reasoning stability, and alignment with user intent. These results show that text-based, memory-centered optimization provides a practical and scalable route toward more reliable agentic systems.

1 INTRODUCTION

Building robust *agentic systems* for complex, long-horizon tasks requires more than stronger single-shot reasoning. It requires mechanisms that (i) **extract** rich behavioral signals from execution, (ii) **aggregate and abstract** these signals across trajectories, roles, and time, and (iii) **translate** them into persistent updates to system policy and memory. Although recent methods improve deliberate reasoning (e.g., chain-of-thought) (Wei et al., 2022), enable structured perception–action loops (e.g., ReAct) (Yao et al., 2022), and align behavior through preference optimization (e.g., RLHF and Constitutional AI) (Ouyang et al., 2022; Bai et al., 2022), they do not provide a unified way to convert *batched, trajectory-level feedback* into durable, role-aware improvements. At the same time, textual optimization primitives such as *textual gradients* (Yuksekgonul et al., 2024) offer interpretable, language-native update signals, but their use for multi-agent, long-horizon adaptation is still not well understood. We present a detailed discussion on the related works in Section A in the Appendix.

In realistic settings, *natural feedback* appears as a stream of heterogeneous textual signals (for example, code review comments, test logs, bug reports, and acceptance notes) collected across multiple decision points and trajectories. This feedback is *multi-granular* (ranging from line-level corrections to system-level design issues), *multi-role* (including programmers, reviewers, testers, and product owners), and *multi-instance* (reoccurring with context-specific variations). Reducing such information to scalar rewards or isolated critiques removes the structure needed to generalize corrections across tasks. Moreover, improvements must persist without modifying model weights, instead influencing future behavior through *memory* and *prompt* updates that depend on both role and context (Madaan et al., 2023; Shinn et al., 2023; Packer et al., 2023). The central challenge is therefore

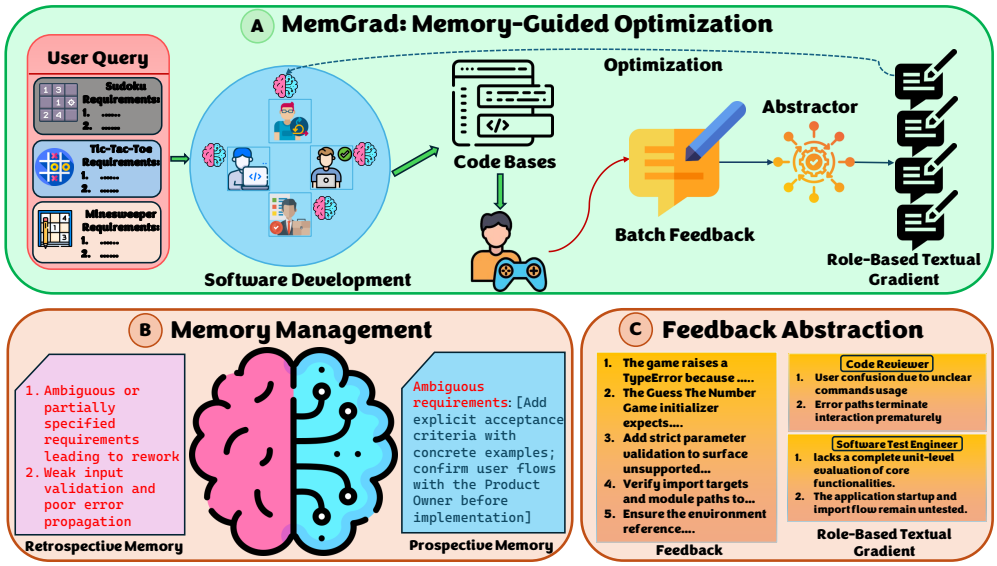


Figure 1: **Overview of the MemGrad framework.** Starting from a user query, agents collaboratively generate artifacts and receive batch feedback from execution traces. This feedback is decomposed into feedback–resolution pairs and routed to role-specific clusters, where it is abstracted into compact textual gradients. These gradients update a dual memory system comprising retrospective memory (capturing recurring failure patterns) and prospective memory (encoding actionable improvement strategies) as shown in (B). The abstracted gradients (as shown in C) further drive prompt optimization, enabling agents to incorporate persistent, role-aware improvements without parameter updates, resulting in more stable and effective long-horizon behavior.

to transform batched textual feedback into compact, role-aware, and actionable representations that update both memory and policy.

A multi-agent software engineering workflow illustrates this challenge. Over a sprint, a batch of tickets generates multiple trajectories: the programmer edits code, the reviewer flags API misuse, the tester reports flaky integration failures, and the product owner notes unmet acceptance criteria. These failures are heterogeneous (for example, cross-module inconsistencies, missing edge-case validation, nondeterministic tests), distributed across files and commits, and often recur with small variations. An effective optimization mechanism must (i) **route** feedback to the correct role, (ii) **abstract** recurring patterns (for example, "improper error handling for I/O operations"), and (iii) **store** both the *cue* ("during file I/O wrapper design") and the *intention* ("validate return codes, apply backoff retries, surface typed errors") so that future actions integrate these corrections automatically.

We introduce ***MemGrad***, a memory-guided optimization framework that uses *textual gradients* (Yuksekgonul et al., 2024) to connect *batched behavioral feedback* with *role-conditioned memory* and *prompt* updates. *MemGrad* operates along three principles. First, it **aggregates feedback** from multiple trajectories and abstracts it into textual gradients that express high-level improvement directions rather than isolated fixes. Second, it **updates agent memory** using a retrospective–prospective structure that stores distilled failure patterns along with improvement heuristics, enabling consistent adaptation across future interactions (Tan et al., 2025). Third, it **optimizes system prompts** so that each agent's policy incorporates both user instructions and gradient-informed memory updates (refer Figure 1).

We evaluate *MemGrad* on *AgileCoder* (Nguyen et al., 2025), a multi-agent software development framework involving Agile-style sprints, dynamic code-graph modifications, and a beta-testing protocol that generates trajectory-level textual critiques. This environment stresses long-horizon coordination, role specialization, and recurring failure patterns, making it an ideal setting to test batched feedback abstraction and memory-conditioned optimization. We report results on end-to-end task success, human evaluation and success rate of unit-test cases, and analyze the contributions of role-based routing, feedback abstraction, dual-store memory, and prompt optimization.

2 PRELIMINARIES

2.1 TEXTGRAD

TextGrad extends automatic differentiation to natural language systems by treating textual artifacts (e.g., prompts, code snippets, intermediate outputs) as variables in a computation graph. The optimization process begins with a forward pass in which a variable x produces an output $y = f(x)$ and a textual loss is computed via `tg.TextLoss` (Eq. (12) in Figure 7). This loss is itself a natural-language critique describing deficiencies of the current output relative to a specification.

The backward pass then converts this critique into a *textual gradient*, as shown in Eq. (13) in Figure 7. TextGrad produces a structured natural-language gradient that specifies how x should be modified to reduce the loss. Intuitively, the textual gradient serves as a improvement signal in language space. The update step (Eq. (14) in Figure 7) applies Textual Gradient Descent (TGD), which rewrites the current x conditioned on this feedback, analogous to how stochastic gradient descent adjusts parameters along the negative gradient direction.

In minibatch settings, multiple losses are aggregated using `tg.sum` (Eq. (15) in Figure 7), and their corresponding textual gradients are composed according to the backward rule shown in Eq. (16) in Figure 7. Operationally, this results in concatenation of gradient signals from each batch element, preserving detailed instance-level feedback. While this design maintains fidelity to individual critiques, the size and verbosity of the resulting gradient object grow with batch size and trajectory length. In complex agentic systems involving long-horizon interactions and heterogeneous feedback sources, such unstructured aggregation can introduce redundancy, interference, and instability, motivating mechanisms that structure and abstract textual gradients prior to propagation (Foerster et al., 2018; Zhou et al., 2025).

2.2 AGILECODER

AgileCoder instantiates professional Agile methodology within a multi-agent LLM framework by assigning agents to specialized roles such as Product Manager, Developer, Reviewer, and Tester. Development proceeds through iterative *sprints*, each comprising requirement analysis, implementation, testing, debugging, and review. Within a sprint, agents exchange artifacts (e.g., specifications, code patches, and test results), and execution feedback generated during testing is used to revise the current repository state. Across successive sprints, refined artifacts are inherited and further improved, while a dynamic code graph maintains repository-level dependencies to support targeted debugging and context-aware retrieval.

AgileCoder's feedback loop focuses on artifacts: tests and critiques trigger small code fixes that propagate through sprint-to-sprint inheritance. This process does not convert feedback into persistent updates to agent behavior or accumulate generalizable tendencies across batches. In contrast, our approach interprets feedback as textual gradients, aggregates them across trajectories, and stores their issue–gradient structure in role-specific memories. These memories capture both recurring failure patterns and their forward-looking corrections, enabling updates to role-conditioned prompts. As a result, improvement occurs not only within the artifacts but also in the agents' underlying policies.

3 OUR APPROACH

3.1 FEEDBACK ABSTRACTION

We consider a multi-agent software development framework inspired by AgileCoder (Section 3.2), where a team of role-specialized agents (programmer, code reviewer, software test engineer, product owner) develops software in response to a batch of queries $Q = \{q^{(1)}, \ldots, q^{(B)}\}$.

For each query $q \in Q$ (we drop the superscript for ease of reading), the team produces a software artifact $S(q)$. Executing the artifact under a beta-testing specification $\mathcal{E}$ produces a trajectory $\tau(q)$, which is evaluated to produce a textual loss

$$L(q) = \texttt{tg.TextLoss}(\tau(q); \mathcal{E}) \tag{1}$$

The prompt for Beta-Tester is presented in the Appendix (see Figure 20). We compute a loss for every trajectory in the batch, yielding $\mathcal{L}_{\text{batch}} = \{\, L(q) \mid q \in Q \,\}$.

The prompt for TextLoss is provided in the Appendix(see Figure 21). Each trajectory-level loss $L(q)$ may describe multiple independent failure modes. We therefore extract a set of feedback statements F from the loss $L(q)$, where each feedback F captures a deficiency in the generated artifact $S(q)$. For every extracted feedback, we compute a textual gradient $\frac{\partial L(q)}{\partial F}$ which we call as *resolution*. A resolution provides a structured natural-language instruction describing how the issue identified by F should be corrected.

This operation is implemented through a `TextGradDecomposer` class which is presented in the Appendix (see Figure 22):

$$\texttt{TextGradDecomposer}\big(L(q)\big) \longrightarrow \left\{ \left(F, \frac{\partial L(q)}{\partial F} \right) \right\} \tag{2}$$

A single loss $L(q)$ can therefore yield multiple feedback–resolution pairs, since a trajectory often reveals several distinct problems. Aggregating these across the batch gives the collection of feedback statements and their associated resolutions $\{F,\ \frac{\partial L(q)}{\partial F}\}^{\text{batch}}$, which serves as the complete set of improvement signals for that training step.

Role-Based Gradient Routing. To distribute improvement signals by agent role, each feedback statement is assigned to the role whose description is most semantically aligned with it. Let $\phi(\cdot)$ be a text encoder and $\mathcal{R}$ the set of roles. We define the role assignment function

$$\text{assignRole}(F) = \arg\max_{r \in \mathcal{R}} \cos\big(\phi(F),\ \phi(r)\big)$$

which selects the role whose embedding is closest to that of the feedback.

This assignment induces role-specific clusters by collecting all feedback–resolution pairs associated with a given role:

$$\text{Cluster}(r) = \left\{ (F,\ \frac{\partial L(q)}{\partial F} \ \middle|\ \text{assignRole}(F) = r \right\} \tag{3}$$

Each cluster therefore contains the feedback and its corresponding resolution for which role r is responsible.

Role-Level Gradient Abstraction. Multiple trajectory-level losses may generate overlapping or semantically redundant feedbacks for the same role, the raw cluster $Cluster(r)$ can become verbose and repetitive. We therefore compress each role-conditioned cluster using a `RoleBasedAbstractor` class (see Figure 23):

$$\texttt{RoleBasedAbstractor}\big(Cluster(r)\big) \longrightarrow \left(\bar{F}_r,\ \frac{\partial \mathcal{L}_{\text{role}}}{\partial \bar{F}_r} \right) \tag{4}$$

This class is presented in the Appendix (see Figure 21). Here, $\bar{F}_r$ denotes a generalized, non-redundant feedback that summarizes recurring failure patterns associated with role r, while $\frac{\partial \mathcal{L}_{\text{role}}}{\partial \bar{r}_r}$ represents the corresponding abstracted resolution capturing the corrective direction. Here, $\mathcal{L}_{\text{role}}$ denotes the feedback-accumulated loss for role r computed over the entire batch (refer Figures 1(C) and 2).

3.2 Memory Updation

To make improvements persist across long-horizon agent runs, we maintain an explicit memory state that separates (i) *what went wrong* from (ii) *what to do next*. This split is motivated by the classical distinction where prospective memory concerns reinstating an intention when the appropriate context or cue is encountered, whereas retrospective memory concerns recalling past content and events (Uttl et al., 2018). Prior work further decomposes prospective memory into subdomains (e.g., episodic event-cued vs. vigilance/monitoring), highlighting that future-oriented behavior benefits from compact, cue-like representations rather than continuously maintained verbose plans (Uttl et al., 2018; Einstein & McDaniel, 2005; Graf & Uttl, 2001).

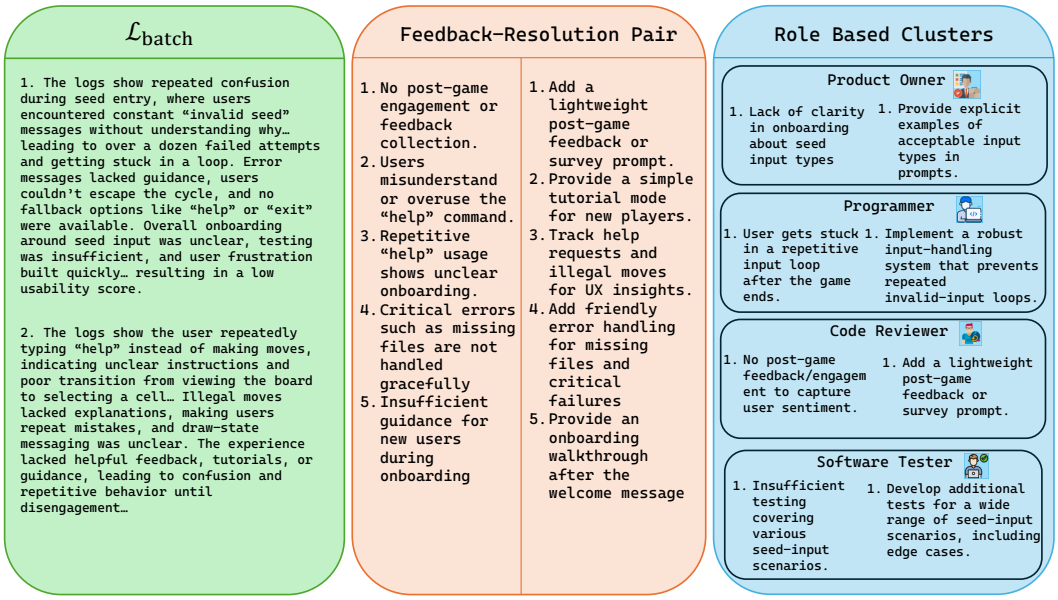


Figure 2: **Feedback Abstraction**: Example of raw losses in $\mathcal{L}_{\text{batch}}$ (left), followed by the generation of feedback–resolution pairs (center), and the role-based distribution of the feedback–resolution pairs (right).

In our setting, each trajectory-level critique $L(q)$ is decomposed into role-conditioned feedback–resolution (as discussed in Section 3.1). We store this non-abstracted pairs directly in memory, allowing us to maintain detailed instance-level structure and richer contextual variation. (see Figures 1(A) and (B)).

Let $M_{\text{ret},r}$ and $M_{\text{pro},r}$ denote the retrospective and prospective memory stores associated with role r, respectively, both initialized as empty. Retrospective memory records concrete failure patterns observed across trajectories, while prospective memory records future-oriented corrective intentions. Memory is updated via the following write operations:

$$M_{\text{ret},r} \leftarrow \texttt{Write}(M_{\text{ret},r}\{\, F \mid \text{assignRole}(F) = r \,\}) \tag{5}$$

$$M_{\text{pro},r} \leftarrow \texttt{Write}\left(M_{\text{pro},r}\left\{ \left(F,\ \frac{\partial L(q)}{\partial F}\right) \,\middle|\, \text{assignRole}(F) = r \right\}\right) \tag{6}$$

Storing both the feedback and its associated resolution in prospective memory allows the feedback to serve as a triggering cue, while the resolution specifies the corrective action linked to that cue. Because a broad feedback can appear in different forms across tasks, it may accumulate multiple resolution over time. Formally, for an feedback F, we maintain the set

$$M_{\text{pro},r} = \left\{F : \left[\ \tfrac{\partial L(q_1)}{\partial F},\ \tfrac{\partial L(q_2)}{\partial F},\ \ldots\ \right]\right\} \tag{7}$$

which contains all resolution associated with F across trajectories. This captures how the same failure may call for different responses depending on context. We present example of how the memory looks in Figure 3 (see Figure 14, 15 and 16 in the Appendix for additional examples).

3.3 Optimization and Inference

Following Equation 4, we obtain abstracted role-conditioned feedback–resolution pair $(\left(\bar{F}_r,\ \frac{\partial \mathcal{L}_{\text{role}}}{\partial \bar{F}_r}\right))$. We use the abstracted feedback–resolution pair as input to a TextGrad-style backward operator to obtain a prompt-level gradient for role r. The prompt for backward operator is presented in the Figure 17. Concretely, we compute

$$\frac{\partial L_{\text{role}}}{\partial p_r} \;=\; \nabla_{\text{LLM}}\left(p_r,\ \left(\bar{F}_r,\ \frac{\partial \mathcal{L}_{\text{role}}}{\partial \bar{F}_r}\right)\right) \tag{8}$$

Retrospective Memory

FEEDBACK

1. Ambiguous or partially specified requirements leading to rework.
2. Weak input validation and poor error propagation.
3. Missing or insufficient unit/integration tests; low meaningful coverage.
4. Tight coupling and unclear module boundaries that hinder change.
5. Performance bottlenecks due to inefficient algorithms or data structures.
6. Inadequate logging and observability, making defects hard to diagnose.

Prospective Memory

FEEDBACK-RESOLUTION PAIRS

1. Ambiguous requirements: [Add explicit acceptance criteria with concrete examples; confirm user flows with the Product Owner before implementation]
2. Weak input validation and error propagation: [Introduce a centralized validation module/schemas; validate at boundaries (API/CLI) and in the domain; add negative tests for invalid, empty, and large inputs]
3. Missing/insufficient tests: [add unit tests for core paths and edge cases]
4. Tight coupling and unclear boundaries: [Refactor into cohesive modules; introduce interfaces/-ports]
5. Performance bottlenecks: [batch I/O where possible]
6. Inadequate logging/observability: [define log levels; log key inputs/outcomes]

Figure 3: Illustration of **retrospective and prospective memory** in the *Programmer Agent*.

where ∇_{LLM} applies the TextGrad backward operation to $\left(\bar{F}_r,\ \frac{\partial \mathcal{L}_{\text{role}}}{\partial \bar{F}_r}\right)$ to produce a textual gradient indicating how p_r should be updated. The role-specific system prompt is then updated using Textual Gradient Descent:

$$p_r^{(t+1)} \leftarrow \text{TGD.step}\left(p_r^{(t)},\ \frac{\partial L_{\text{role}}}{\partial p_r}\right) \tag{9}$$

The optimized system prompt by our our method and TextGrad is presented in the Figure 4 (refer to the Figure 10, 11, and 12 in the Appendix for optimized system prompt of other agents). Simultaneously, the feedback–directive pairs are written into memory stores as shown in Equation 5 and 6.

After T optimization steps, the learned agent configuration is defined by the optimized prompts $\{p_r^{(T)}\}$ and the consolidated memory stores $\{M_{\text{ret},r}^{(T)}, M_{\text{pro},r}^{(T)}\}$.

Inference with Memory-Conditioned Prompts. During test-time execution, the system is frozen.When an agent with role r is about to execute an action a, it retrieves relevant retrospective cues by embedding the current query and matching it against entries stored in retrospective memory, selecting the top-k most similar elements based on a similarity metric.

$$F_r(a) \ = \ \text{Retrieve}(M_{\text{ret},r},\ a) \tag{10}$$

Given the retrieved feedback, the agent then performs a second retrieval step over prospective memory to obtain corresponding resolutions. Specifically, the agent retrieves resolution-level guidance that addresses the identified issues:

$$Re_r(a) = \text{Retrieve}(M_{\text{pro},r},\ F_r(a)) \tag{11}$$

The agent's system prompt (p_r) is augmented with both the retrospective feedback cues and the retrieved prospective resolutions, yielding a memory-conditioned prompt $\tilde{p}_r$. The prompt for the

Base Prompt

You are Programmer. we are both working at AgileCoder. We share a common interest in collaborating to successfully complete a task assigned by a new customer. You can write/create computer software or applications by providing a specific programming language to the computer. You have extensive computing and coding experience in many varieties of programming languages and platforms, such as Python, Java, C, C++, HTML, CSS, JavaScript, XML, SQL, PHP, etc, Here is a new customer's task: task.To complete the task, you must write a response that appropriately solves the requested instruction based on your expertise and customer's needs.

TextGrad

You are Programmer. In alignment with the goal of creating an engaging and functional software experience, the following software design principles should be prioritized. Development should focus on building a deterministic, reliable system that responds predictably to user actions while maintaining a clear and consistent internal state. The system must represent its core state accurately and enforce well-defined rules for state transitions. Only valid inputs should be accepted, and invalid inputs should be rejected without altering the system state. Clear feedback must be provided to users for both valid and invalid actions, ensuring transparency and ease of use. Core logic should be implemented in a centralized and well-encapsulated component to ensure correctness and maintainability. The system should also provide user feedback for invalid inputs, ensure the system state remains unchanged after such inputs, and log invalid interaction attempts for diagnostic purposes. Robust error handling should be implemented to gracefully manage unexpected conditions, with detailed error messages that describe the nature of the issue and potential corrective actions. All significant events, state changes, and errors should be logged to facilitate debugging and post-execution analysis. The system should handle invalid inputs gracefully without crashing, provide suggestions for valid inputs, and ensure stability across repeated interactions. The system should log events, state changes, and errors comprehensively after each user interaction to support traceability and debugging.....

MemGrad

You are an advanced programming assistant tasked with building a turn-based, CLI-based game with an engaging, user-friendly experience grounded in robust logic and defensive error handling. Adopt a structured, user-centered development approach that emphasizes clarity, modularity, and continuous feedback throughout the lifecycle of a software system. Clearly define core objectives, rules, and success or failure conditions while designing interactions that encourage engagement and meaningful user responses. Incorporate strong input validation with informative, user-friendly feedback to prevent errors and guide correct usage. Maintain thorough documentation and clear in-code commentary to explain functionality, design decisions, and interaction flows for future maintainability. Apply proactive error handling that anticipates common user mistakes and communicates issues without frustration. Design the system using modular components with well-defined responsibilities to support scalability, testing, and collaboration. Establish focused testing protocols and iterative feedback mechanisms that integrate real user insights into each development cycle, and ensure that visual or interface elements consistently reflect system state changes in a clear and intuitive manner to reinforce usability and engagement.

Figure 4: **Optimization of system prompt for the *Programmer* agent**: (A) the original system prompt, (B) the prompt optimized using TextGrad, where **colored segments indicate redundant or overlapping instructions introduced during optimization** (shortened for readability; full version in Figure 9), and (C) the prompt optimized using MemGrad.

retrieval from prospective memory is provided in the Appendix (see Figure 19). Each agent executes its action conditioned on the augmented prompt $\tilde{p}_r$, thereby proactively integrating previously learned remedies. All roles perform this retrieval–augmentation step before producing their outputs, and the coordinated execution of memory-conditioned agents yields the final artifact.

4 Experimental Setup

Dataset and Task Setup. We curate a set of 30 command-line games as tasks to be implemented by *AgileCoder*. Inspired by the AgileCoder framework, we adopt several games from its ProjectDev dataset that support command-line interaction, and additionally generate new games to further aug-

Lights Out

Toggle a 5×5 grid where a move flips the selected cell and its orthogonal neighbors; turn all lights off within the move cap.

- Use a 5×5 binary grid with compact text rendering.
- Seed or fix the initial grid deterministically.
- Accept zero-based `(row,col)` with $0 \leq row, col < 5$.
- Reject out-of-bounds inputs without using a move.
- Flip the selected cell and orthogonal neighbors only.
- Render the full grid after each move.
- Track and display move count and remaining moves.
- End immediately when all cells are off.
- End when the move cap is reached if not solved.
- Refuse inputs after termination.

Sudoku (4×4)

Fill a 4×4 Sudoku with digits 1–4 respecting row, column, and 2×2 constraints; errors counted to a cap.

- Provide a deterministic starting 4×4 puzzle (fixed/seeded).
- Accept moves `(row,col,value)` with 0-based indices and value in {1,2,3,4}.
- Reject placements on non-empty cells.
- Validate row, column, and 2×2 subgrid constraints.
- On invalid moves, report error and increment violation count.
- On valid moves, update grid and render compactly.
- Track filled cells and remaining empties.
- Terminate when all cells validly filled.
- Terminate when violation cap is reached (failure).
- Ignore inputs after termination.

Figure 5: **Dataset example**: Representative games from the evaluation test set. For each game, we present the specific requirements for game creation.

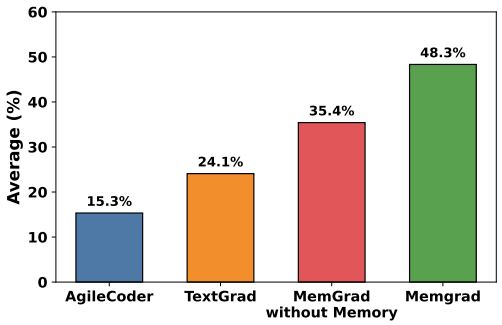
ment and diversify our dataset. From this dataset, 10 samples are used for training the system, and the remaining 20 are reserved for testing. An example illustrating the structure of the samples in the dataset is shown in Figure 5 and full details are presented in the Appendix (refer Section B).

System and Training. Our system follows the AgileCoder paradigm, employing four collaborative agents that operate across successive development phases to synthesize, refine, and validate software artifacts. These agents include: product owner, programmer, code reviewer, and software test engineer. To enable artifact-level loss computation and provide executable feedback signals, we introduce an additional Beta-Tester agent that runs each generated game and logs all salient actions and outcomes (refer Figures 20 and 21 in the Appendix). These execution logs are analyzed to compute the loss and derive feedback–resolution pairs, which are stored in memory to guide subsequent iterations. Training is conducted for two epochs with a batch size of five, after which the optimized system prompt and the populated memory are used to generate games for the held-out test set. We use GPT-4o mini for all our experiments. In all components requiring embedding generation, textual inputs are encoded using the text-embedding-3-small model

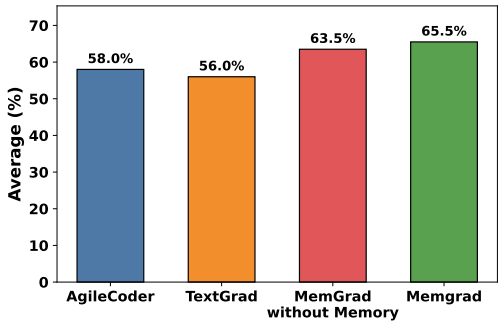
Evaluation Protocol. At test time, the system is evaluated under two complementary setups, and we report the final score as the average of both evaluations. In the first setup, the software test engineer agent generates unit-test cases based on the requirements provided for the target game, and performance is measured as the percentage of test cases successfully passed (example is presented in the Figure 8 in the Appendix). In the second setup, a human software engineer interacts with the generated game and verifies the extent to which the specified requirements (provided as a query) are satisfied; the percentage of fulfilled requirements is recorded. For the human-evaluation setup, each game is played twice by a human evaluator, and the better-performing run is selected as the representative score for that game. We report the unit-test case performance and the human requirement-fulfillment performance. We compare MemGrad against Agile Coder with and without TextGrad as the baseline.

5 Results and Analysis

1. **Memory improves long horizon performance**: Figures 6, 26, and 27 show that MemGrad outperforms TextGrad across most evaluated games, with the performance gap widening as game complexity increases. While TextGrad remains competitive on tasks such as 8-Puzzle, Chess and



(a) Performance score from the proportion of test cases successfully passed.



(b) Performance score from requirements satisfied during human gameplay.

Figure 6: **Quantitative Results.** We present the average score achieved when evaluating on a test set of 20 games. We compare MemGrad against AgileCoder with and without TextGrad, as well as AgileCoder augmented with MemGrad but without memory.

Nim, its performance deteriorates on other games that require sustained state tracking. TextGrad-generated games also tend to be less user-friendly. For example, in Figures 24 and 25 , the Mastermind game reveals the secret code at the start, fails to prompt the user for a seed, and does not communicate the number of steps in advance. MemGrad, in contrast, satisfies these usability criteria consistently. These results suggest that memory-driven optimization enables more reliable and durable improvements in both gameplay correctness and user experience.

2. **Abstraction stablizes prompts and gradients**: Directly accumulating raw losses can produce large gradients, as shown in Figure 13 in the Appendix. This leads to unstable updates and excessively long prompts (see Figures 4 and 9). Moreover, the system prompt starts carrying redundant instruction, for example, repetitive instruction proper invalid input handling. MemGrad mitigates this issue by abstracting feedback before using it as a gradient signal, preventing gradient explosion during optimization. The abstracted gradients compress repeated feedback and resolutions into concise signals, enabling more stable updates and producing prompts that are more effective for implementing the user requirements.
3. **Role based clustering improves optimization balance**: Figure 4, 10, 11, and 12 suggests that aggregating feedback without role-based clustering dilutes the learning signal for agents that participate less frequently. For example, in Table 1 of the Appendix it can be seen that Software Test Engineer and Code Reviewer participate less frequently than the other two. Thus the impact on their system prompt by TextGrad is minimal. On the other hand, by distributing the losses and gradients at the role level, MemGrad ensures that each agent receives a focused and relevant update signal. Thus, resulting in more balanced learning and improved overall performance.
4. **Efficient performance gains with minimal cost overhead** :
   Overall token usage and monetary cost remain minimal for both methods, with total expenditure staying well below one dollar even after three full training epochs. While MemGrad incurs a minimal increase in token usage compared to TextGrad (2.92M vs. 2.29M tokens), the resulting cost difference is small approximately 0.08 dollor under the GPT-4o-mini. Despite this minimal increase in computational overhead, MemGrad consistently achieves higher evaluation scores than TextGrad, demonstrating that the performance improvements are obtained efficiently without a meaningful rise in cost.

6 Conclusion

This work demonstrates that structured textual feedback combined with explicit memory enables multi-agent systems to adapt more reliably over long time horizons. By organizing feedback, abstracting recurring patterns, and retaining actionable guidance in persistent memory, agents can learn from prior experience and reduce repeated errors. Our findings indicate that MemGrad offers a foundation for building adaptive, interpretable, and robust agentic systems across diverse tasks, with clear implications for real-world applications.

REFERENCES

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Gilles O Einstein and Mark A McDaniel. Prospective memory: Multiple retrieval processes. *Current Directions in Psychological Science*, 14(6):286–290, 2005.

Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Peter Graf and Bob Uttl. Prospective memory: A new focus for research. *Consciousness and cognition*, 10(4):437–450, 2001.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Yilong Li, Chen Qian, Yu Xia, Ruijie Shi, Yufan Dang, Zihao Xie, Ziming You, Weize Chen, Cheng Yang, Weichuan Liu, et al. Cross-task experiential learning on llm-based multi-agent collaboration. *arXiv preprint arXiv:2505.23187*, 2025.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

Minh Huynh Nguyen, Thang Phan Chau, Phong X Nguyen, and Nghi DQ Bui. Agilecoder: Dynamic collaborative agents for software development based on agile methodology. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pp. 156–167. IEEE, 2025.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35: 27730–27744, 2022.

Charles Packer, Vivian Fang, Shishir G Patil, Kevin Lin, Sarah Wooders, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *CoRR*, 2023.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, 2024.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Zhen Tan, Jun Yan, I-Hung Hsu, Rujun Han, Zifeng Wang, Long Le, Yiwen Song, Yanfei Chen, Hamid Palangi, George Lee, et al. In prospect and retrospect: Reflective memory management for long-term personalized dialogue agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8416–8439, 2025.

Bob Uttl, Carmela A White, Kelsey Cnudde, and Laura M Grant. Prospective memory, retrospective memory, and individual differences in cognitive abilities, personality, and psychopathology. *PLoS One*, 13(3):e0193806, 2018.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

Yingxuan Yang, Huacan Chai, Shuai Shao, Yuanyi Song, Siyuan Qi, Renting Rui, and Weinan Zhang. Agentnet: Decentralized evolutionary coordination for llm-based multi-agent systems. *arXiv preprint arXiv:2504.00587*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Peiyan Zhang, Haibo Jin, Leyang Hu, Xinnuo Li, Liying Kang, Man Luo, Yangqiu Song, and Haohan Wang. Revolve: Optimizing ai systems by tracking response evolution in textual optimization. *arXiv preprint arXiv:2412.03092*, 2024.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents. *arXiv preprint arXiv:2506.15841*, 2025.

A Related Work

Optimization in language space. TextGrad formalizes natural-language feedback as textual gradients and offers a PyTorch-like API to optimize prompts, solutions, code, and other variables (Yuksekgonul et al., 2024). REVOLVE stabilizes iterative textual optimization by tracking response evolution across steps (Zhang et al., 2024). DSPy compiles declarative language-model programs into self-improving pipelines via a compiler that learns demonstrations and tunes multi-stage graphs (Khattab et al., 2023). OPRO treats large language models as derivative-free optimizers that iteratively propose improved instructions or solutions (Yang et al., 2023). DPO directly optimizes policies from preference data, simplifying alignment relative to RLHF (Rafailov et al., 2023). MemGrad differs by aggregating *batch-level* behavioral feedback into textual gradients that *co-update* retrospective and prospective memory and system prompts, providing a persistent, interpretable target that can be scheduled by controllers like REVOLVE or serve as an endpoint for compiler-style tuning and derivative-free search (Yuksekgonul et al., 2024; Zhang et al., 2024).

Self-improvement without weight updates. Self-Refine iteratively critiques and revises outputs without supervised training or RL (Madaan et al., 2023), and Reflexion introduces verbal reinforcement with episodic reflective memory to guide future trials (Shinn et al., 2023). MemGrad inherits the interpretability and parameter-free benefits of these approaches but *abstracts across batches of trajectories* into structured gradients and writes them into a retrospective–prospective memory, rather than refining single outputs or storing ad hoc reflections.

Acting, tools, and persistent state. ReAct interleaves chain-of-thought with environment actions for grounded decision making (Yao et al., 2022); Toolformer self-learns API usage to calculators, search, and other tools from few demonstrations (Schick et al., 2023). MemGPT implements OS-style virtual context management with multi-tier memory and interrupts for long-horizon interaction (Packer et al., 2023), while Voyager demonstrates open-ended lifelong learning with an automatic curriculum and a transferable skill library (Wang et al., 2023). MemGrad is orthogonal to planning and tools and instead *makes improvements persistent* by updating memory and prompts using batch-abstracted textual gradients.

Multi-agent software engineering and orchestration. AgileCoder structures collaborative software development into Agile-style sprints with specialized roles and a dynamic code-dependency graph (Nguyen et al., 2025). MetaGPT encodes standardized operating procedures to guide role-based collaboration (Hong et al., 2023), while ChatDev coordinates agents through staged processes of design, coding, testing, and documentation (Qian et al., 2024). Beyond these workflow-oriented systems, AgentNet explores decentralized coordination over a dynamically evolving DAG with retrieval-based agent memories (Yang et al., 2025), and MAEL focuses on cross-task transfer by maintaining per-agent experience pools containing high-reward exemplars (Li et al., 2025). Our work is complementary to these approaches: MemGrad aims to distill patterns present in batches of agent trajectories into textual gradients that update memory and prompts. In contexts such as AgileCoder, this enables role-specific retrospective and prospective cues in the agent's memory. More broadly, MemGrad offers a distinct mechanism for shaping agent behavior through aggregate feedback, without modifying the internal retrieval or coordination strategies of systems like MAEL or AgentNet.

B Evaluation Setup

The evaluation process is conducted using both manual inspection and automated unit test cases to ensure the correctness and reliability of the evaluation results. For each task, we run a method two times using the same prompt, producing a program for each run. For each generated program, we attempt to execute this program. If the program is executable, we evaluate it against all expected requirements. The final score is determined by the percentage of requirements the program meets. We present the games along with their corresponding requirements for both the training and test sets.

B.1 Train Set

- **Guess the Number**

■ **Forward** ■ **Backward** ■ **Update**

Forward: Textual Loss Given an optimizable variable x (e.g., a prompt or code snippet) and output $y = f(x)$, TextGrad defines:

$$L = \texttt{tg.TextLoss}(y, \text{spec}) \tag{12}$$

Example loss output (schematic):

> *"The solution fails on edge cases (empty input, overflow). The algorithm is $O(n^2)$ and may time out. Clarify invariants and handle the case when k is not found."*

Backward: Textual Gradient TextGrad converts the critique into a textual gradient for x:

$$\frac{\partial L}{\partial x} = \nabla_{\text{LLM}}(x, y, L) \tag{13}$$

where ∇_{LLM} denotes an LLM prompted to produce directional feedback. A minimal gradient-prompt template is:

> *"Given variable x, output $y = f(x)$, and critique L, describe precisely how x should be modified to reduce L."*

Example gradient (schematic):

> *"Modify x to: (i) explicitly handle empty input and missing k; (ii) replace nested loops with a prefix-balance method to achieve $O(n)$; (iii) document the invariant for balance."*

Update: Textual Gradient Descent TextGrad applies a rewrite update conditioned on the textual gradient:

$$x^{(t+1)} \leftarrow \texttt{TGD.step}\left(x^{(t)}, \frac{\partial L}{\partial x}\right) \tag{14}$$

Minibatch aggregation via `tg.sum`. For losses $\{L_b\}_{b=1}^{B}$:

$$L = \texttt{tg.sum}(L_1, \ldots, L_B) \tag{15}$$

$$\frac{\partial L}{\partial x} = \texttt{tg.sum}\left(\frac{\partial L_1}{\partial x}, \ldots, \frac{\partial L_B}{\partial x}\right) \approx \left\{\frac{\partial L_1}{\partial x}\right\} \| \cdots \| \left\{\frac{\partial L_B}{\partial x}\right\} \tag{16}$$

Figure 7: Illustration of TextGrad's forward textual loss (Eq. 12), backward textual gradient construction (Eq. 13), and update via TGD (Eq. 14). Minibatch composition uses `tg.sum` (Eq. 15–16).

  - Fixed integer range (e.g., 1–100)
  - Deterministic secret number via seed
  - Accept integer guesses within range only
  - Reject invalid inputs without consuming attempts
  - Feedback: higher / lower / correct
  - Attempts consumed only on valid guesses
  - Track remaining attempts
  - Terminate on correct guess or attempt cap
  - Reject inputs after termination
- **Tic Tac Toe**
  - 3x3 board rendered each turn
  - Accept moves as index or (row, col)
  - Reject invalid or occupied cells without consuming turn
  - Alternate turns with deterministic opponent
  - Detect win (rows, columns, diagonals)
  - Detect draw after 9 moves
  - Terminate on win or draw

  - Reject inputs after termination
- **Bulls and Cows (3-digit)**
  - Deterministic 3-digit secret with distinct digits
  - Accept only valid 3-digit guesses with distinct digits
  - Reject invalid guesses without consuming attempts
  - Compute bulls (correct digit and position)
  - Compute cows (correct digit, wrong position)
  - Track attempts and remaining guesses
  - Terminate on 3 bulls or attempt cap
  - Reject inputs after termination
- **2048 (Move-Limited)**
  - Standard 4x4 grid
  - Deterministic tile movement and merging
  - Accept directions: up/down/left/right
  - Reject invalid commands without consuming moves
  - Seeded deterministic tile spawning
  - Render grid after each move
  - Terminate on win (2048), loss, or move cap
  - Reject inputs after termination
- **Minesweeper (Micro)**
  - Small grid (e.g., 6x6) with seeded mine placement
  - Actions: reveal or flag
  - Reject invalid actions without consuming moves
  - Auto-expand zero cells
  - Render board with hidden/flagged states
  - Terminate on mine reveal or victory
  - Enforce move cap
  - Reject inputs after termination
- **Hangman (Short Words)**
  - Seeded word list (4–6 letters)
  - Deterministic word selection
  - Accept single alphabetic letter guesses
  - Reject invalid or repeated guesses
  - Reveal all correct letter positions
  - Track remaining lives
  - Terminate on success or life exhaustion
  - Reject inputs after termination
- **Snake (Turn-Based Tiny)**
  - Small grid (e.g., 6x6)
  - Deterministic snake and food placement
  - Accept U/D/L/R commands
  - Move snake each turn
  - Detect wall and self-collision
  - Grow snake and update score on food
  - Terminate on collision or move cap
  - Reject inputs after termination
- **Chain Reaction (Tiny Grid)**
  - Small grid (e.g., 4x4)
  - Track cell ownership and counters

  - Deterministic overflow propagation
  - Alternate turns with deterministic opponent
  - Reject illegal placements without consuming turn
  - Detect win/loss by cell ownership
  - Terminate on win, loss, or move cap
  - Reject inputs after termination
- **Othello (Small Board)**
  - 6x6 board with standard initial layout
  - Validate moves in all 8 directions
  - Flip captured discs deterministically
  - Enforce forced pass if no legal moves
  - Alternate turns with deterministic opponent
  - Detect termination when no moves remain
  - Compute final disc counts
  - Reject inputs after termination
- **Tetris (Turn-Based Micro)**
  - Small grid (e.g., 6x10)
  - Deterministic tetromino sequence
  - Accept movement and rotation commands
  - Reject illegal moves or rotations
  - Apply gravity each turn
  - Lock pieces and clear full lines
  - Detect top-out condition
  - Terminate on top-out or move cap
  - Reject inputs after termination

B.2 Test Set

- **Rock Paper Scissors (Best-of-5)**
  - Accept only `rock`, `paper`, or `scissors` (case-insensitive)
  - Use seeded deterministic opponent moves
  - Compute outcomes using standard rules
  - Display per-round opponent move and result
  - Maintain and display running scores
  - Count ties as rounds without score change
  - End early on reaching 3 wins
  - End after exactly 5 rounds otherwise
  - Print final result and scores
  - Reject input after termination
- **Coin Toss Predictor (Streak)**
  - Accept only `H` or `T` (case-insensitive)
  - Generate flips deterministically from seed
  - Increment streak on correct prediction
  - Reset streak on incorrect prediction
  - Track longest streak
  - Show flip and correctness each step
  - Terminate on target streak or move cap
  - Print final summary
  - Reject input after termination
- **Dice Sum Guess**

  - Accept integer guesses in range [2,12]
  - Reject invalid inputs without using attempts
  - Roll two dice deterministically
  - Compare guess to true sum
  - Return `low`, `high`, or `correct`
  - Consume attempts only for valid guesses
  - Terminate on correct guess or attempt cap
  - Print end summary
  - Ignore input after termination
- **Word Scramble (Tiny)**
  - Use fixed list of 3–5 letter words
  - Deterministically select and scramble word
  - Display scrambled word
  - Accept alphabetic full-word guesses only
  - Reject wrong-length guesses without penalty
  - Case-insensitive exact match
  - Consume attempts only on valid guesses
  - Reveal original word at end
  - Reject input after termination
- **Quiz Game (MCQ)**
  - Load fixed MCQ bank with one correct option
  - Present questions in deterministic order
  - Accept only A/B/C/D inputs
  - Reject invalid input without advancing
  - Provide immediate correctness feedback
  - Increment score only on correct answers
  - Advance exactly one question per answer
  - Terminate after final question
  - Print final score summary
- **Flashcard Trainer**
  - Use predefined small deck
  - Deterministic card order
  - Accept string answers only
  - Case-insensitive trimmed exact match
  - Provide per-card feedback
  - Advance after each valid answer
  - Track correct/incorrect totals
  - Terminate after last card
  - Reject input after termination
- **Nim (Take-Away)**
  - Initialize single pile (default 15)
  - Accept removals in $\{1,2,3\}$
  - Reject illegal removals without consuming turn
  - Update pile size correctly
  - Support solo or deterministic opponent
  - Alternate turns in versus mode
  - Declare winner correctly
  - Terminate on zero pile or move cap
  - Ignore input after termination

- **Lights Out**
  - Use 5x5 binary grid
  - Deterministic initial configuration
  - Accept valid (row,col) coordinates
  - Reject out-of-bounds moves
  - Flip cell and orthogonal neighbors
  - Render grid after each move
  - Track remaining moves
  - Terminate on all lights off or move cap
  - Reject input after termination
- **Sudoku (4x4)**
  - Deterministic starting puzzle
  - Accept (row,col,value) moves
  - Reject moves on filled cells
  - Enforce row, column, and subgrid constraints
  - Count violations on invalid moves
  - Update and render grid on valid moves
  - Track remaining empty cells
  - Terminate on completion or violation cap
  - Ignore input after termination
- **Connect Four (Turn-Capped)**
  - Use small grid (e.g., 6x7)
  - Accept valid column indices
  - Reject full-column moves
  - Apply gravity correctly
  - Alternate turns with deterministic opponent
  - Detect wins in all directions
  - Terminate immediately on win
  - Terminate as draw on turn cap
  - Reject input after termination
- **Mastermind**
  - Fixed alphabet and code length 4
  - Deterministic secret generation
  - Accept valid-length guesses only
  - Reject invalid guesses without attempt use
  - Compute exact and partial matches
  - Handle duplicates correctly
  - Track remaining guesses
  - Terminate on solve or guess cap
  - Reveal code at end
- **Battleship (Micro)**
  - Deterministic ship placement (sizes 2 and 3)
  - Accept valid grid coordinates
  - Reject repeated or invalid shots
  - Resolve miss, hit, and sunk
  - Track ship health
  - Render fog-of-war grid
  - Terminate when all ships sunk
  - End on move cap if unsolved

  - Reveal layout after termination
- **8-Puzzle (Sliding)**
  - Deterministic solvable start state
  - Accept U/D/L/R moves
  - Reject illegal moves
  - Swap blank with adjacent tile
  - Render board after each move
  - Track remaining moves
  - Terminate on goal state
  - End on move cap
  - Ignore input after termination
- **Checkers (Minimal Ruleset)**
  - Initialize 6x6 board
  - Accept diagonal moves only
  - Validate men and king movement
  - Support single captures
  - Promote to king on last rank
  - Alternate turns with deterministic opponent
  - Reject illegal moves
  - Terminate on no moves or turn cap
  - Ignore input after termination
- **Caro (Gomoku Small Board)**
  - Use 9x9 board
  - Accept valid placements
  - Reject occupied or invalid cells
  - Alternate turns with deterministic opponent
  - Detect exact 5-in-a-row
  - Render board after each move
  - Terminate immediately on win
  - End as draw on turn cap
  - Ignore input after termination
- **Text-Based Maze (Tiny)**
  - Deterministic maze layout
  - Accept N/S/E/W commands
  - Reject blocked or invalid moves
  - Update position correctly
  - Render position or map
  - Track remaining steps
  - Detect goal cell
  - Terminate on success or step cap
  - Ignore input after termination
- **Sokoban (Micro Warehouse)**
  - Deterministic tiny level
  - Accept N/S/E/W commands
  - Allow pushing one box only
  - Reject illegal pushes
  - Update player and box positions
  - Track boxes on targets
  - Render grid after moves

  - Terminate on solve or move cap
  - Reject input after termination

- **Hex (Small Board)**
  - Use 5x5 hex grid
  - Accept valid placements
  - Reject occupied or invalid cells
  - Alternate turns with deterministic opponent
  - Check connectivity after each move
  - Detect correct player win condition
  - Terminate immediately on win
  - End as draw on turn cap
  - Ignore input after termination

- **Chess Endgame Trainer (K+R vs K)**
  - Initialize K+R vs K positions
  - Validate rook and king moves
  - Prevent illegal or self-check moves
  - Deterministic black replies
  - Detect check, mate, and stalemate
  - Optional 50-move rule
  - Render board after each move
  - Terminate on mate, stalemate, or cap
  - Ignore input after termination

- **Nonogram (5x5)**
  - Fixed row and column clues
  - Start with unknown grid
  - Accept valid marking commands
  - Reject invalid coordinates
  - Prevent basic contradictions
  - Render grid after each move
  - Track remaining moves
  - Detect exact clue satisfaction
  - Terminate on completion or move cap

Mastermind: Additional Test Cases

```python
import pytest
from mastermind_game import MastermindGame

def test_generate_code():
    game = MastermindGame(seed=1)
    assert game.secret_code == "BDAF"

def test_validate_guess():
    game = MastermindGame()
    assert game.validate_guess("ABCD") is True
    assert game.validate_guess("AB") is False
    assert game.validate_guess("ABXYZ") is False

def test_give_feedback():
    game = MastermindGame(seed=1)
    game.secret_code = "ABCD"
    assert game.give_feedback("ABCD") == (4, 0)
    assert game.give_feedback("ADEF") == (1, 0)
    assert game.give_feedback("EFGH") == (0, 0)

def test_fixed_code_length():
    game = MastermindGame()
    assert len(game.secret_code) == 4

def test_secret_uses_fixed_alphabet():
    game = MastermindGame(seed=1)
    allowed = set("ABCDEF")
    assert all(ch in allowed for ch in game.secret_code)

def test_deterministic_secret_generation_same_seed():
    game1 = MastermindGame(seed=7)
    game2 = MastermindGame(seed=7)
    assert game1.secret_code == game2.secret_code

def test_deterministic_secret_generation_different_seed():
    game1 = MastermindGame(seed=1)
    game2 = MastermindGame(seed=2)
    assert game1.secret_code != game2.secret_code

def test_reject_invalid_guess_without_attempt_use():
    game = MastermindGame(seed=1)
    before = game.remaining_guesses
    assert game.validate_guess("AB") is False
    assert game.remaining_guesses == before

def test_reject_invalid_alphabet_guess():
    game = MastermindGame()
    assert game.validate_guess("WXYZ") is False
```

Figure 8: **Example unit test cases used for evaluation:** We present representative unit tests for the Mastermind game, illustrating how deterministic behavior, input validation, and feedback correctness are verified during evaluation.

Phase Name	Participating Agent
1. ProductBacklogCreating	Product Owner
2. ProductBacklogReview	Product Owner
3. ProductBacklogModification	Product Owner
4. SprintBacklogReview	Product Owner
5. SprintBacklogModification	Product Owner
6. SprintBacklogCreating	Product Owner
7. NextSprintBacklogCreating	Product Owner
8. SprintReview	Product Owner
9. Coding	Programmer
10. CodeFormatting	Programmer
11. CodeReviewModification	Programmer
12. CodeReviewComment	Code Reviewer
13. TestingPlan	Software Test Engineer

Table 1: Different phases in the AgileCoder framework and agents that function during that phase.

Base Prompt

You are Programmer. we are both working at AgileCoder. We share a common interest in collaborating to successfully complete a task assigned by a new customer. You can write/create computer software or applications by providing a specific programming language to the computer. You have extensive computing and coding experience in many varieties of programming languages and platforms, such as Python, Java, C, C++, HTML, CSS, JavaScript, XML, SQL, PHP, etc, Here is a new customer's task: task.To complete the task, you must write a response that appropriately solves the requested instruction based on your expertise and customer's needs.

TextGrad

Your task is to develop a robust, fully functional, and user-friendly implementation of the game Othello (Reversi). The code should focus on providing an engaging experience for users while ensuring proactive error handling and comprehensive documentation. Throughout development, you should actively seek user feedback through structured methods such as surveys, interviews, and iterative playtesting, using these insights to refine gameplay mechanics, particularly help functionalities and user interactions. A comprehensive help system must be implemented to clearly explain available commands and address common user frustrations and input expectations, adapting dynamically over time based on user behavior and newly introduced features. Help content should remain contextual, inclusive, and accessible to users with diverse needs. Input handling must be robust, with a modular validation system independent of core gameplay logic, ensuring only valid inputs are accepted while providing clear and constructive error messages; repeated invalid actions should be logged to support continuous refinement of the user experience. Error handling should provide standardized, user-friendly guidance that helps players understand and correct mistakes, while backend logging identifies recurring issues. The codebase should include detailed comments explaining design choices, class responsibilities, and method usage, complemented by a high-level architectural document outlining coding standards and best practices to ensure long-term maintainability. A modular design philosophy should guide the structure of the system, breaking functionality into focused, reusable components that encourage scalability and follow established design patterns. Development should follow a test-driven approach, incorporating unit, integration, and user acceptance tests to ensure robustness across edge cases and real-world scenarios. User engagement can be enhanced by incorporating interactive elements within the help system that evolve in response to user queries and collected feedback. Accessibility considerations must be incorporated by adhering to WCAG guidelines, using appropriate color contrasts and simple, clear language. The system should include mechanisms for players to provide direct in-game feedback, enabling informed decision-making driven by actual user behavior. To support longevity, versioning should be maintained for help content and code practices, with an emphasis on anticipating future user requirements and emerging technologies. Clear, measurable success criteria—such as test performance, user satisfaction ratings, error rates, and feedback metrics—should be defined to objectively evaluate implementation quality. Collaborative development practices, including peer code reviews and pair programming, should be encouraged to promote shared ownership and early detection of issues. Finally, performance optimization should be integrated throughout development to ensure that the codebase remains efficient and responsive as the user base expands. Implementing these guidelines will result in an Othello game that is not only functional and maintainable but also deeply engaging and supportive for users, with a clean architecture, thorough testing, and a strong help system aligned with best practices in software development and user interaction.

MemGrad

You are an advanced programming assistant tasked with building a turn-based, CLI-based game with an engaging, user-friendly experience grounded in robust logic and defensive error handling. Begin by clarifying goals and specs so inputs are handled cleanly and players receive immediate, contextual feedback after each action. Structure the code with clear classes/modules and focused functions for setup, commands, core rules, and any event or collision logic, and manage game state explicitly with deterministic initialization, updates, resets, and tracking of scores/progress and entity positions. Design algorithms for efficiency to avoid unnecessary work, especially in core loops and validation paths. Integrate testing from day one—unit (plus integration or acceptance where useful) that covers normal flows and edge cases, particularly around input handling. Maintain thorough docstrings and inline comments explaining each class/function's purpose, inputs, outputs, side effects, and core logic, alongside a concise architecture overview and coding standards. Iterate regularly based on test outcomes and user feedback to refine mechanics and prompts, and present game state clearly in text with readable symbols, minimal flicker, and accessibility-friendly choices (simple language, adequate contrast if using ANSI colors).

Figure 9: **Optimization of system prompt for the *Programmer* agent**: We present (A) the original system prompt of the agent, (B) the prompt optimized using TextGrad, and (C) the prompt optimized using MemGrad.

Base Prompt

You are the Code Reviewer. We both work at AgileCoder and share a common goal of collaborating to successfully complete a task assigned by a new customer. You can help programmers assess source code for troubleshooting, fix bugs to improve code quality and robustness, and propose enhancements. Here is the new customer's task: task. To complete the task, write a response that appropriately addresses the requested instruction based on your expertise and the customer's needs.

TextGrad

You are the Code Reviewer. Furnish clear, actionable, and constructive feedback that fosters growth, guarantees accuracy, and upholds integrity in your critiques of the submitted code.

MemGrad

You are the Code Reviewer. During the code review process, provide constructive and insightful feedback that strengthens the codebase. Your comments should be clear, specific, and actionable, focusing on helping the programmer understand what to improve and why. Deliver feedback in a supportive and collaborative tone to encourage open communication, shared learning, and continuous improvement within the team.

Figure 10: **Optimization of system prompt for the *Code Reviewer* agent**: We present (A) the original system prompt of the agent, (B) the prompt optimized using TextGrad, and (C) the prompt optimized using MemGrad.

Base Prompt

You are a Software Test Engineer. We are both working at AgileCoder and share a common interest in collaborating to successfully complete a task assigned by a new customer. You can use the software as intended to analyze its functional properties, design manual and automated test procedures to evaluate each software product, build and implement software evaluation test programs, and run test programs to ensure that testing protocols evaluate the software correctly. Here is the new customer's task: task. To complete the task, you must write a response that appropriately solves the requested instruction based on your expertise and the customer's needs.

TextGrad

You are a Software Test Engineer. We are both working at AgileCoder. You are currently in the Testing-Plan phase. Your primary objective is to develop a comprehensive testing strategy that includes rigorous evaluation procedures. Ensure that you address all necessary requirements and criteria to deliver a thorough assessment of the project's functionality and performance

MemGrad

As a Software Test Engineer at AgileCoder, I will create a structured testing plan that defines clear objectives, including verifying that the software behaves correctly, handles errors gracefully, and provides a smooth user experience. The plan will outline measurable success criteria, such as meeting all functional requirements, having no critical bugs, and ensuring that core features work reliably across different scenarios. I will also describe how test results will be documented, including logs of what was tested, any issues found, and the final status of each test. This documentation will help the team make informed decisions at every stage of development and ensure that all important aspects of testing—functional checks, edge cases, usability, and stability—are fully covered.

Figure 11: **Optimization of system prompt for the *Software Test Engineer* agent**: We present (A) the original system prompt of the agent, (B) the prompt optimized using TextGrad, and (C) the prompt optimized using MemGrad.

Base Prompt

You are the Product Owner. We are both working at AgileCoder and share a common interest in collaborating to successfully complete a task assigned by a new customer. AgileCoder uses Agile Scrum for software development. You are responsible for maximizing the value delivered by the team by ensuring that the team backlog is aligned with customer and stakeholder needs. Here is the new customer's task: task. To complete the task, you must write a response that appropriately solves the requested instruction based on your expertise and the customer's needs.

TextGrad

You are the Product Owner. To create a comprehensive and actionable product for the given user task in the phase, you should explore the user's motivations, emotions, and anticipated experiences for each user story, clearly articulating intent, personas, and pain points. Each feature must have measurable goals and acceptance criteria reflecting quantifiable outcomes such as engagement, error reduction, and satisfaction. Break down each story into manageable subtasks with roles, responsibilities, edge cases, and error-handling guidance. Write each backlog item in the user story format "As a [type of user], I want [feature] so that [benefit]." Prioritize all items using a structured method and label them as High, Medium, or Low based on expected user impact. Identify dependencies between tasks to improve planning and workflow efficiency. Describe testing strategies for each task and outline how iterative feedback will be integrated throughout development. Ensure accessibility considerations are included in each story and design features that accommodate diverse abilities. Emphasize iterative development, using feedback to continuously refine user stories and mechanics. Use visual story-mapping techniques to represent user journeys and highlight important interactions. Maintain clear documentation of decisions and processes to support transparency and knowledge sharing. Include tasks for post-sprint reviews to capture lessons learned and promote continuous improvement. By following these guidelines, the resulting product backlog will be user-centered, structured, actionable, and aligned with project objectives, enabling iterative enhancements and improved user satisfaction.

MemGrad

You are the Product Owner. Create a comprehensive product for the task at hand, focusing on developing a project that effectively meets user expectations and fosters an engaging game experience. Each backlog item should include detailed user stories that outline features while elaborating on user motivations, behaviors, and expectations. The user stories should emphasize interactive elements that enhance engagement, such as dynamic visual cues or exciting gameplay mechanics at program launch, to build user investment in the game. Incorporate measurable acceptance criteria for all items, defining success metrics such as user satisfaction and engagement rates to show how well features meet user needs. Outline mechanisms for collecting user feedback throughout development to ensure the implementation remains adaptable and aligned with real user experiences. Prioritize the backlog based on user value, categorizing items as Must Have, Should Have, and Could Have, and recommend a structured method for ranking urgency and importance based on potential user impact and gameplay flow. Consider diverse user pathways that reflect different skill levels to ensure inclusivity and accommodate a wide range of players. This approach will enhance engagement and satisfaction while ensuring the overall development process remains user-centered and inclusive, ultimately supporting a richer final game experience.

Figure 12: **Optimization of system prompt for the *Product Owner* agent**: We present (A) the original system prompt of the agent, (B) the prompt optimized using TextGrad, and (C) the prompt optimized using MemGrad.

TextGrad

Your task is to develop a robust, fully functional, and user-friendly implementation of the game. The code should focus on providing an engaging experience for users while ensuring proactive error handling and comprehensive documentation. Key Requirements: Actively seek user feedback through structured methods like surveys, interviews, and iterative playtesting throughout development, and use this feedback to refine gameplay mechanics, especially for help functionalities and user interactions. Implement a comprehensive help system that clearly explains available commands and addresses potential user frustrations and input expectations, designing the help system to dynamically adapt based on user interactions and learning from user queries over time. Create help content that resonates with user context, allowing for seamless updates as new features are introduced, and ensure this content is inclusive and accessible, accommodating diverse user needs. Develop a robust input validation system that is modular and independent from gameplay logic, ensuring that only valid inputs are accepted while providing clear feedback for any errors, and implement logging for repeated invalid actions to continually refine user experience. Design user-friendly error messages that guide players in understanding mistakes, provide standardized formats across the application for consistency, and incorporate error logging to help identify common user frustrations. Include detailed comments in the code explaining class purposes, method usages, and design decisions, and create a high-level document outlining architecture decisions, coding standards, and best practices alongside traditional documentation to improve maintainability. Structure the code in a modular way, breaking functionality into focused classes and functions, and encourage the development of reusable components with clear responsibilities, utilizing design patterns that facilitate future scalability and enhancements. Adopt a test-driven development (TDD) approach, where comprehensive unit tests are written alongside or before implementation, including integration and user acceptance tests to ensure robustness against edge cases and real-world usage scenarios. Consider incorporating interactive elements within the help system that evolve based on user queries and feedback, implementing feedback loops to enhance user interaction capabilities. Ensure that your help content and gameplay are compliant with WCAG (Web Content Accessibility Guidelines) to facilitate use by individuals with disabilities, implementing color contrasts and simple language usage for better accessibility. Build mechanisms for users to provide insights directly within the gameplay context, enabling developers to make informed decisions based on actual player experiences and behaviors. Maintain versioning information for all aspects of the help content and coding practices, and design with growth in mind by anticipating future user requirements and emerging technologies to ensure lasting relevance. Define clear, measurable success criteria for functionality and user experience, such as user ratings during playtests, acceptable error rates, and feedback metrics, allowing for an objective evaluation of implementation quality. Foster a culture of collaboration through regular peer code reviews and pair programming sessions, encouraging knowledge sharing and early issue detection. Integrate performance considerations throughout the development process to ensure that the codebase remains efficient and responsive as the user base grows. Implementing these guidelines will ensure your game is not only functional and maintainable but also delivers a truly engaging and supportive experience for users, with a well-organized codebase, thorough testing, and a help system that reflects best practices in software development and user interaction.

MemGrad

The primary objective is to deliver an engaging and user-friendly experience while ensuring robust game logic and well-engineered error handling. The implementation should accurately reflect the game requirements, effectively handle user inputs, and provide immediate and informative feedback that supports strategic gameplay. Error handling must be integrated throughout the system to manage unexpected inputs or invalid states, with clear communication to the user whenever corrective action is needed. The code should follow a modular structure with clearly defined classes and functions that handle game setup, user commands, core game logic, and collision detection. Game state management must be explicit and well organized, including proper initialization, resetting, and tracking of scores, snake position, and all relevant variables. Performance considerations should guide design decisions, particularly in food spawning and collision detection logic, to ensure smooth and efficient execution. A testing framework should be incorporated from the outset, with unit tests covering all major functionalities and edge cases, especially those involving user input. Thorough inline documentation must accompany the code, explaining the purpose, inputs, outputs, and internal logic of each component to support clarity and maintainability. Development should follow an iterative approach informed by user feedback and testing results, enabling continuous refinement of gameplay mechanics. Although the game operates in a command-line environment, attention should be given to clear and expressive visual representation of the game state to enhance user engagement. By adhering to these principles, the resulting implementation will provide a high-quality, functional, and user-centered turn-based experience.

Figure 13: Textual Gradient computed for Programmer agent using Backward Engine.

Retrospective Memory

FEEDBACK

1. Vague error messages reduce traceability.
2. Weak input validation across entry points.
3. Sparse logs hinder defect diagnosis.
4. Help content is hard to access quickly.
5. Inconsistent guidance after invalid inputs.
6. Minimal feedback on near-correct commands.

Prospective Memory

FEEDBACK-RESOLUTION PAIRS

1. Vague error messages: [Standardize error format; add clear remediation hints]
2. Weak input validation: [Validate at boundaries; add negative tests]
3. Sparse logs: [Add structured logging for key actions and failures]
4. Hard-to-access help: [Streamline help display; link from prompts]
5. Inconsistent guidance: [Prompt next steps after invalid input]
6. Minimal feedback on near-correct commands: [Acknowledge partial validity; suggest exact format]

Figure 14: Illustration of **retrospective and prospective memory** in the *Code Reviewer* agent.

Retrospective Memory

FEEDBACK

1. Command recognition issues causing failed interactions.
2. Unclear feedback after illegal moves.
3. Limited user interaction beyond exceptions.
4. Insufficient debugging information during playtesting.
5. Users confused by unclear command usage.
6. No feedback on partially valid commands.

Prospective Memory

FEEDBACK-RESOLUTION PAIRS

1. Command recognition issues: [Improve recognition and list allowed commands clearly]
2. Game flow and feedback: [Enhance messaging for illegal moves to guide users]
3. Limited interaction beyond exceptions: [Implement user-friendly error messages/screens]
4. Insufficient debugging information: [Embed logging for detailed errors during tests]
5. Unclear command usage: [Provide a concise help message with examples at start]
6. No feedback on partially valid commands: [Confirm help command and prompt the next action]

Figure 15: Illustration of **retrospective and prospective memory** in the *Software Test Engineer* agent.

Retrospective Memory

FEEDBACK

1. Onboarding lacks clear, concise instructions.
2. Early feedback loops are limited.
3. Errors do not reflect user context well.
4. Help content lacks quick-start examples.
5. In-game prompts are inconsistent across flows.
6. Accessibility cues are minimal.

Prospective Memory

FEEDBACK-RESOLUTION PAIRS

1. Onboarding clarity: [Provide a short quick-start guide at launch]
2. Early feedback loops: [Add lightweight prompts for user feedback]
3. Context-aware errors: [Tailor messages to user actions and next steps]
4. Help examples: [Include command examples and common scenarios]
5. Prompt consistency: [Unify tone and structure across prompts]
6. Accessibility cues: [Use simple language and clear visual/text cues]

Figure 16: Illustration of **retrospective and prospective memory** in the *Product Owner* agent.

Gradient Calculation

BACKWARD ENGINE — GLOSSARY AND SYSTEM PROMPT

GLOSSARY OF TAGS

1. <LM_SYSTEM_PROMPT>: The system prompt for the language model.
2. <LM_INPUT>: The input to the language model.
3. <LM_OUTPUT>: The output of the language model.
4. <OBJECTIVE_FUNCTION>: The objective of the optimization task.
5. <VARIABLE>: Specifies the span of the variable.
6. <ROLE>: The role description of the variable.

BACKWARD ENGINE SYSTEM PROMPT

You are part of an optimization system that improves a given text (i.e., the variable). You are the gradient (feedback) engine. Your only responsibility is to give intelligent and creative feedback and constructive criticism to variables, given an objective specified in <OBJECTIVE_FUNCTION> </OBJECTIVE_FUNCTION> tags. The variables may be solutions to problems, prompts to language models, code, or any other text-based variable.

Pay attention to the role description of the variable and the context in which it is used. You should assume that the variable will be used in a similar context in the future. Only provide strategies, explanations, and methods to change in the variable. Do not propose a new version of the variable; that will be the job of the optimizer. Your only job is to send feedback and criticism (compute "gradients").

For instance, feedback can be in the form of "Since language models have the X failure mode...", "Adding X can fix this error because...", "Removing X can improve the objective function because...", "Changing X to Y would fix the mistake...", aimed at the downstream objective. If a variable is already working well (e.g., the objective function is perfect, an evaluation shows the response is accurate), you should not give feedback.

Figure 17: Backward engine system prompt for gradient calculation.

Retrieving feedback from retrospective memory

According to the new user's task and our software designs:

1. Task: task
2. Modality: modality
3. Programming Language: language

As the assistant_role, extract from the retrospective memory the most relevant feedback that can help in performing this task.

Figure 18: Prompt for retrieving feedback from retrospective memory.

Retrieving resolutions from prospective memory

According to the new user's task and our software designs:

1. Task: task
2. Modality: modality
3. Programming Language: language

As the assistant_role, based on the feedback extracted, retrieve the corresponding resolutions from the prospective memory.

Figure 19: Prompt for retrieving resolutions from prospective memory.

Beta Tester

You are a CLI Beta-Tester.
You will be given:

- Game metadata including name, description, and requirements
- A brief summary of session memory
- The most recent terminal output since the last input
- The current raw prompt passed to the input function

Your task is to decide the next single line a user would type.
Output format:

- You may include one optional note line wrapped in note tags with a maximum of about twenty words
- You must include exactly one action tag containing the final single line input

Rules:

- The action tag must contain only the next input with no additional commentary
- Follow the expected CLI input formats such as y or n, menu numbers, or row column pairs
- Do not repeat inputs that have already failed
- If unsure, use safe navigation commands such as menu, back, help, or exit when appropriate

Figure 20: Prompt used to instruct the Beta Tester on providing realistic, single-line inputs that simulate natural player behavior during gameplay.

Loss Computation

You have data collected from a Beta Tester running a program

1. Task: task
2. Modality: modality
3. Programming Language: language

As a Play Testing Analyst, evaluate the program by reviewing the code, the done undone tasks, the test case summary, and the beta testing logs. Determine whether the requirements are satisfied. The requirements checklist in the logs may be incomplete, so cross-check the full trajectory and use reasonable judgment to assess completeness and feasibility. Identify errors, abnormal behavior, moments where the Beta Tester bot appears stuck, usability issues, or logical inconsistencies. Reference specific examples from the logs, code, tasks, or testing summary.Your output must:- Use declarative statements to describe issues. Use imperative statements to give improvement instructions. Include no code. Overall execution quality rubric:

1. Severely broken or barely functional.
2. Major gaps or frequent failures.
3. Mixed; multiple missing or unstable parts.
4. Mostly correct; several issues.
5. Fully or nearly fully correct; minor issues.

Figure 21: Prompt for loss computation from a trajectory generated by the Beta Tester after playing the game.

`TextGradDecomposer`

Extract feedback to resolution pairs from the paragraph and return JSON strictly as:

OUTPUT FORMAT

- pairs
  - feedback ...
  - resolution ...
  - ...

RULES

1. Each feedback must have a concrete and actionable resolution that addresses it.
2. Exactly one resolution per feedback; choose the most actionable and specific if multiple exist.
3. Use imperative voice for resolution; keep both feedback and resolution concise but specific.
4. No markdown and no commentary outside the JSON payload.

Figure 22: **`TextGradDecomposer`** class designed to decompose loss into structured feedback–resolution pairs for optimization.

RoleBasedAbstractor

You are abstracting multiple related issue to todo pairs into a single generalized pair. Return structured output strictly as:
Name: Short theme title Abstract issue: A generalized and precise problem statement. If measurable, include a key performance indicator. Abstract todo: One recommended plan that may include steps and may reference an owner persona if useful. Tags: A list of optional taxonomy labels.
Rules:

- Preserve the core intent across all examples and avoid vendor or tool specific constraints unless necessary.
- Keep the output testable, with acceptance conditions either implied or explicitly stated.
- Be concise, using one to three sentences each for the issue and the todo.

Figure 23: **RoleBasedAbstractor** class that abstracts role-based clusters of feedback–resolution pairs into a single representative issue and resolution.

TextGrad

1. Secret code (for testing purposes): FAAF
2. Enter your guess (or 'help' for instructions, 'quit' to exit): ABCD
   Exact matches: 0, Partial matches: 1
3. Enter your guess (or 'help' for instructions, 'quit' to exit): DEFA
   Exact matches: 0, Partial matches: 2
4. Enter your guess (or 'help' for instructions, 'quit' to exit): aaaa
   Exact matches: 2, Partial matches: 0
5. Enter your guess (or 'help' for instructions, 'quit' to exit): ABCDE
   Invalid input. Please enter exactly 4 characters from A to F.
6. Enter your guess (or 'help' for instructions, 'quit' to exit): 12
   Invalid input. Please enter exactly 4 characters from A to F.
7. Enter your guess (or 'help' for instructions, 'quit' to exit): FAAF
   Exact matches: 4, Partial matches: 0
8. Congratulations! You've guessed the correct code.

MemGrad

1. Enter a seed for code generation (must be a non-empty string): 12345 Welcome to Mastermind! You have 10 guesses to find the code.
2. Enter your guess (4 characters from A-F): ABCD
   Exact matches: 0, Partial matches: 1
3. Enter your guess (4 characters from A-F): GHIJ
   Invalid guess: Input contains invalid characters. Only A, B, C, D, E, F are allowed.
4. Enter your guess (4 characters from A-F): aaaa
   Exact matches: 0, Partial matches: 0
5. Enter your guess (4 characters from A-F): ABCDE
   Invalid guess: Input must be exactly 4 characters long.
6. Enter your guess (4 characters from A-F): EFEF
   Exact matches: 1, Partial matches: 2
7. Enter your guess (4 characters from A-F): EEEE
   Exact matches: 1, Partial matches: 0
8. Enter your guess (4 characters from A-F): FEFE
   Exact matches: 2, Partial matches: 1
9. Enter your guess (4 characters from A-F): EEEE
   Exact matches: 1, Partial matches: 0
10. Enter your guess (4 characters from A-F): FFFF
    Exact matches: 2, Partial matches: 0
11. Enter your guess (4 characters from A-F): FFEE
    Exact matches: 2, Partial matches: 1
12. Enter your guess (4 characters from A-F): FFFE
    Exact matches: 3, Partial matches: 0
13. Game over! The secret code was: DFFE.

Figure 24: **Mastermind Game**: We present the games played by the human evaluator for both the TextGrad and MemGrad versions. The TextGrad version revealed the secret code at the beginning of the game, even though it was meant to be inferred through gameplay.

TextGrad

1. Welcome to the 4x4 Sudoku Game!
   1 . . 2
   . . 1 .
   . 1 . .
   2 . . 1

2. Enter your move in the format (row, col, value) or 'help' for instructions: help
   Invalid input format. Use (row,col,value).
   1 . . 2
   . . 1 .
   . 1 . .
   2 . . 1

3. Enter your move in the format (row, col, value) or 'help' for instructions: 1,1,2
   Invalid input format. Use (row,col,value).
   1 . . 2
   . . 1 .
   . 1 . .
   2 . . 1

4. Enter your move in the format (row, col, value) or 'help' for instructions: (1,1,2)
   Move accepted: (1, 1) → 2.
   1 . . 2
   . 2 1 .
   . 1 . .
   2 . . 1

5. Enter your move in the format (row, col, value) or 'help' for instructions: (5,5,1)
   IndexError: list index out of range

MemGrad

1. Welcome to 4x4 Sudoku! Type 'help' for usage instructions or 'exit' to quit. Current Sudoku grid:
   1 . . 3
   . . 1 .
   3 . . 1
   . 2 . .

2. Enter a move (row,col,value) or 'help': help
   Usage: Enter your move in the format (row,col,value) where row and col are 0-indexed.
   Example: (0,1,2) to place 2 at row 0, column 1. Type 'exit' or 'quit' to stop playing.

3. Enter a move (row,col,value) or 'help': (1,1,3)
   Placed 3 at (1,1)
   Current Sudoku grid:
   1 . . 3
   . 3 1 .
   3 . . 1
   . 2 . .

4. Enter a move (row,col,value) or 'help': (6,6,1) Error: Indices must be within 0-3 and value must be between 1-4.

5. Enter a move (row,col,value) or 'help': (0,1,4)
   Placed 4 at (0,1)
   Current Sudoku grid:
   1 4 . 3
   . 3 1 .
   3 . . 1
   . 2 . .

6. Enter a move (row,col,value) or 'help': (1,0,2)
   Placed 2 at (1,0)
   Current Sudoku grid:
   1 4 . 3
   2 3 1 .
   3 . . 1
   . 2 . .

7. Enter a move (row,col,value) or 'help': (3,0,4)
   Placed 4 at (3,0)
   Current Sudoku grid:
   1 4 . 3
   2 3 1 .
   3 . . 1
   4 2 . .

8. Enter a move (row,col,value) or 'help': exit
   Thank you for playing!

Figure 25: **Sudoku Game**: We present the games played by the human evaluator for both the TextGrad and MemGrad versions. The TextGrad version failed to handle the edge cases efficiently.

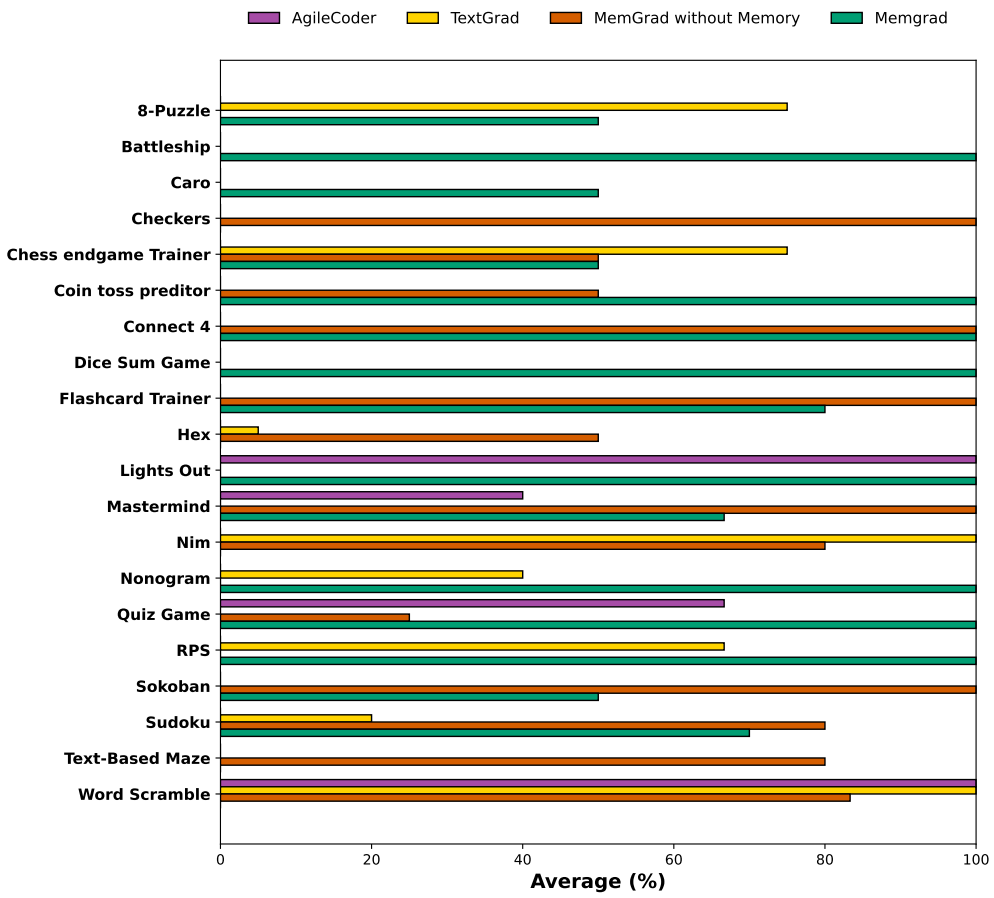


Figure 26: **Quantitative Result:** We present the performance score computed from the proportion of test cases successfully passed.

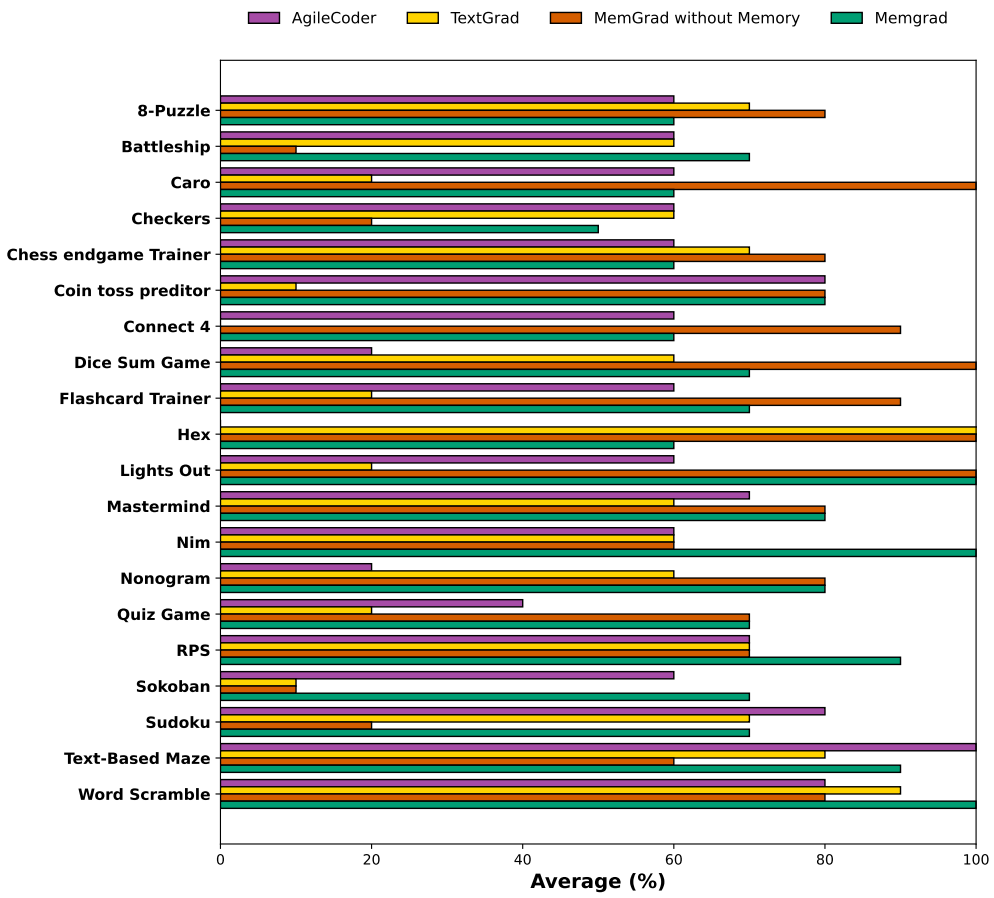


Figure 27: **Quantitative Result:** We present the performance score computed from the number of requirements satisfied during human gameplay.