# NLPG: NATURAL-LANGUAGE POLICY GRADIENTS FOR SELF-EVOLVING LANGUAGE AGENTS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Large language model agents increasingly rely on compound programs for retrieval, tool use, reasoning, and verification, yet their failures often arise from local procedural decisions. Existing reinforcement-learning and prompt-optimization approaches typically rely on scalar rewards or repeatedly modify entire prompts, making it difficult to capture and reuse procedural improvements while preserving a frozen agent. To address this problem, We propose **Natural-Language Policy Gradients (NLPG)**, an external policy-memory method for improving a fixed agent without changing its model parameters or program structure. NLPG diagnoses execution traces, propagates downstream feedback backward through the module graph, and converts recurring failures into route-local natural-language corrections that are aggregated into bounded policy updates for subsequent executions. Across six benchmarks covering memory, reasoning, instruction following, and evidence verification, NLPG also outperforms the strongest listed baseline for each benchmark by 8.71 percentage points on average. These results provide evidence that evaluated procedural experience can be transformed into local and interpretable policy updates, enabling continual improvement of frozen agents.

## 1 INTRODUCTION

Large language models (LLMs) increasingly solve tasks as compound agents: a fixed program coordinates multiple model calls, retrieval operations, tools, and intermediate representations before producing an answer Yao et al. (2023); Schick et al. (2023); Wu et al. (2024); Khattab et al. (2024). This decomposition improves capability, but it also creates a credit-assignment problem. A failed final answer may result from an earlier query, an omitted entity, an incorrect memory write, or the final synthesis step. Updating the whole prompt after every failure therefore provides only coarse feedback and may modify parts of the agent that were not responsible for the error.

Parameter-based reinforcement learning optimizes scalar objectives but typically requires many rollouts and offers limited interpretability Sutton & Barto (2018); Shao et al. (2024). Non-parametric prompt optimizers generate, refine, or evolve natural-language instructions based on task performance Zhou et al. (2023); Pryzant et al. (2023); Yang et al. (2024); Opsahl-Ong et al. (2024); Fernando et al. (2024); Agrawal et al. (2026). Related adaptation methods exploit user corrections, self-feedback, verbal reflection, or cross-task experience Madaan et al. (2022; 2023); Shinn et al. (2023); Zhao et al. (2024), while long-term memory systems primarily store and retrieve interaction content Packer et al. (2023); Park et al. (2023); Chhikara et al. (2025); Fang et al. (2026); Yang et al. (2026). However, these approaches generally lack explicit credit assignment from an evaluated failure to the responsible module policy and its corresponding update in a compound agent.

We introduce **Natural-Language Policy Gradients (NLPG)**, a parameter-free policy-evolution method for compound language agents. Its key mechanism is graph-backward credit assignment: NLPG traces a failed outcome through recorded execution dependencies and attributes it to the responsible module or task-family policy rather than to the entire agent or a global prompt. It keeps the benchmark program, model parameters, and retrieval backend fixed, while injecting bounded procedural instructions into the relevant invocation. After each evaluation pass, the diagnosed deviations and execution dependencies generate candidate corrections, which are aggregated across executions, filtered for safety and redundancy, and ranked according to their evidence support (cross-execution support for a correction), failure pressure (the normalized weight of the associated execution de-



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

viation), and critic confidence (the critic's confidence that the correction is reusable). Accepted corrections are activated only in a later round, enabling localized and interpretable policy evolution without changing the original agent architecture.

NLPG separates policy execution from policy improvement in a compound agent. During execution, each module uses the policy associated with its route; after evaluation, the observed failure is traced back to the module policies that could have contributed to it, enabling module-local credit assignment. A language-based critic then converts this evidence into natural-language corrections, which are applied between evaluation passes. When execution dependencies are available, the same feedback can be propagated backward through the recorded module graph to produce more localized update directions.

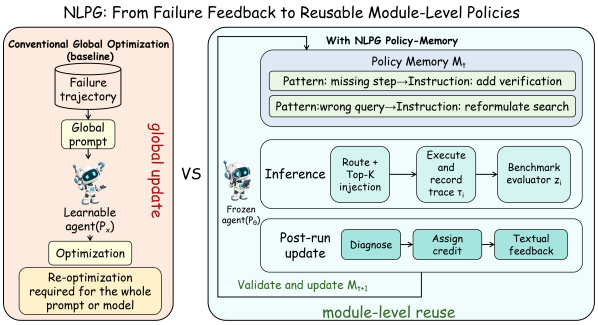



Figure 1: Conceptual comparison of global prompt-level updates and NLPG's route-indexed policy updates.

We evaluate NLPG on six benchmarks spanning long-horizon memory and transfer: LoCoMo, HaluMem, LongMemEval, HotpotQA, IFBench, and HoVer. Under protocol-matched comparisons, NLPG improves the primary metric by 2.23–23.11 points over the strongest listed baseline, depending on the benchmark. In the three-task transfer suite aligned with the GEPA evaluation setting (HotpotQA, IFBench, and HoVer) Agrawal et al. (2026), NLPG averages 59.00 with Qwen3-8B, compared to 51.09 for GEPA (+7.91 percentage points) and averages 63.33 with GPT-4.1 Mini, compared to 59.43 for GEPA+Merge (+3.90 percentage points). Across these benchmarks, the improvements target procedural demands including multi-hop composition in HotpotQA, temporal grounding in LoCoMo, evidence routing in HoVer, and explicit output constraints in IFBench. This is a cross-benchmark pattern; the contribution of individual components is analyzed separately in the ablation experiments. Our contributions and key findings are summarized as follows:

- We introduce a parameter-free framework that evolves compound language agents through external natural-language policy memory while keeping the underlying program, model parameters, and retrieval system fixed.
- We develop a natural-language gradient mechanism that attributes diagnosed execution deviations along recorded dependencies and converts them into reusable, route-local policy corrections.
- We conduct a protocol-matched evaluation across six benchmarks and show consistent gains in long-horizon memory use, multi-hop reasoning, instruction following, and evidence-grounded verification under different model settings.

## 2 PRELIMINARIES

This section introduces the execution process of the self-evolving agent and defines the notation used in the remainder of the method. A discussion of related work is provided in Appendix B.

**Self-evolving compound agent.** Let $P_\theta$ denote a self-evolving compound agent, where $\theta$ denotes the underlying model parameters and the fixed program configuration. Given a task input $x_i$ from a particular task category, the agent consults the current policy memory $M_t$ and invokes the required program modules according to its fixed execution procedure. The agent produces a task response $y_i$ and an evaluation result $z_i$ that records whether the execution satisfies the task requirements and, when available, the diagnosed failure information. The observable execution trace is denoted by $\tau_i$ (Sutton and Barto, 2018; Yao et al., 2023; Liu et al., 2024). After the current evaluation round is completed, the agent extracts effective policy instructions from the collected traces and uses them to construct the next memory version $M_{t+1}$. The model parameters and the fixed program configuration remain unchanged throughout this process.

We use the following notation:

- **Program module.** A program module $b$ is a callable unit that performs one explicit operation in the agent's execution procedure, such as retrieval, external tool use, verification, or result parsing. For the $j$-th invocation in example $i$, the module produces an explicit output $o_{i,j}$.
- **Policy memory.** The policy memory is indexed by a task category or a program module. We write $M_t = \{M_t^a\}_{a \in \mathcal{A}}$, where $M_t^a$ contains short natural-language instructions describing how the corresponding category or module should execute its operation. These instructions may specify, for example, how to construct a retrieval query, organize intermediate evidence, or formulate the final answer. Policy memory stores procedural experience rather than task-specific reference answers.
- **Execution trace.** For task $i$, the observable execution trace records the task input, the explicit outputs produced during execution, and the final response:

$$\tau_i = \left(x_i, \{o_{i,j}\}_{j=1}^{J_i}, y_i\right), \tag{1}$$

  where $J_i$ is the number of module invocations. The trace may contain generated queries, retrieved evidence, tool results, and intermediate module outputs, but does not include hidden model reasoning.
- **Failure.** A failure is an identifiable deviation between the execution or final response and the task requirements that can be attributed to one or more relevant modules. Examples include omitting a key entity from a query, failing to retrieve a necessary document, or producing an answer that is unsupported by the available evidence. A failure is represented by a failure signature $s$ and may be associated with a set of policy-memory indices.
- **Benchmark evaluation.** For benchmark $\mathcal{B}$, an evaluator uses the execution trace and the benchmark's reference information, such as annotated evidence, task constraints, a gold answer, or scoring rules, to produce $z_i$. The reference information is used to evaluate the execution and diagnose failures; it is not stored as policy memory.

The self-evolution problem studied in this paper keeps $P_\theta$ and its execution procedure fixed, while updating the external policy memory. Given evaluated executions $(\tau_i, z_i)$, the goal is to convert recurring failures into natural-language update directions associated with the relevant memory indices and to retain only updates supported by multiple examples. Formally, the next policy-memory version is constructed as

$$M_{t+1} = \mathrm{Update}\big(M_t, \{(\tau_i, z_i)\}_{i=1}^{N}\big). \tag{2}$$

where $M_t$ is fixed during the evaluation round and $M_{t+1}$ is loaded only in a subsequent round. The main challenge is therefore to determine both **where** a failure should update the policy memory and **how** the associated instruction should be changed. NLPG addresses this challenge by propagating failure information along the observed module dependencies and aggregating the resulting natural-language update directions across examples.

## 3 METHOD

As shown in Figure 2, NLPG updates the procedural policies of a fixed compound agent while keeping its model, program, and retrieval index unchanged. Section 3.1 introduces routed policy memory, Section 3.2 presents natural-language policy gradients, and Section 3.3 describes cross-example policy aggregation and validation.

### 3.1 NLPG POLICY MEMORY

To improve a compound agent without retraining its underlying models, NLPG treats procedural experience as an external, updateable source of improvement. Let $P_\theta$ denote the compound agent, where $\theta$ includes the model parameters and program configuration that remain fixed during self-evolution. Throughout this paper, a **task instance** denotes one benchmark item supplied to the agent, while an **execution** denotes the complete processing of that instance, including all module invocations, the final response, and its evaluation. At policy version $t$, the agent receives task instance $x_i$, consults the external policy memory $M_t$, and executes the fixed program.

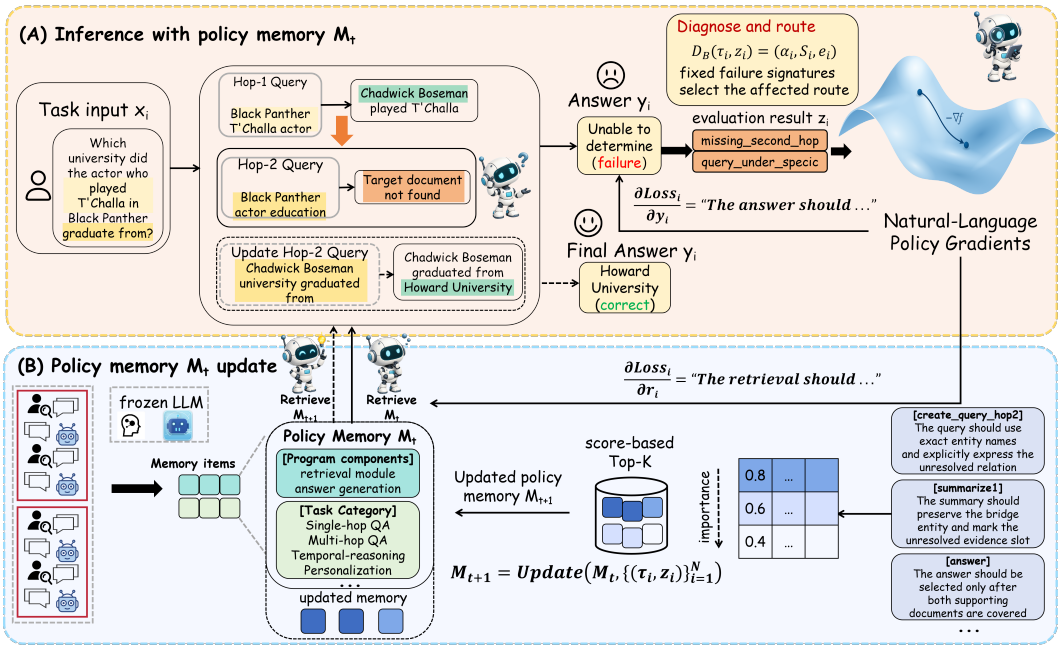



Figure 2: **Overview of NLPG. (A) Inference and failure diagnosis.** The compound agent retrieves module-local policies from the current policy memory $M_t$ and injects them into the corresponding module calls. **(B) Policy-memory update.** NLPG aggregates routed language gradients across evaluated trajectories, scores and selects reusable instructions, and stores them under the corresponding program modules and task categories to construct $M_{t+1}$.

**Policy routing and candidate construction.** For module-routed tasks, $a_{i,j}$ denotes the module invoked at step $j$ of example $i$; for task-family-routed tasks, it denotes the example's task family. The selected block $M_t^{a_{i,j}}$ contains reusable instructions for that route. When enabled, a shared block $M_t^{\text{gen}}$ supplements it with route-independent rules on evidence sufficiency, output format, and unsupported conclusions.

For the current invocation, NLPG reads the route-specific policy memory $M_t^{a_{i,j}}$ and, when available, combines it with the shared policy memory $M_t^{\text{gen}}$. We set $M_t^{\text{gen}} = \varnothing$ when a benchmark does not define shared policy memory. After removing instructions with identical normalized text, the candidate policy set is

$$\mathcal{C}_{t,i,j} = \text{Dedup}\left(M_t^{a_{i,j}} \cup M_t^{\text{gen}}\right), \tag{3}$$

where $\mathcal{C}_{t,i,j}$ is the complete candidate pool for the current invocation. It contains the instructions attached to route $a_{i,j}$ and, when defined, the instructions shared across routes; an instruction appearing in both memories is retained only once after text normalization. The pool is used for selection rather than inserted in full: candidates are ranked by their stored utility and tie-breaking scores, and only the highest-ranked $K$ instructions are formatted and added to the module context.

**Policy selection.** Each candidate $m \in \mathcal{C}_{t,i,j}$ contains an instruction and a historical priority score $w_t(m)$ derived from previous evaluated executions. NLPG ranks the candidates by this score and selects at most $K$ instructions:

$$\mathcal{P}_{t,i,j} = \text{TopK}_{w_t}(\mathcal{C}_{t,i,j}, K). \tag{4}$$

The selected set determines which policy instructions are injected into the current module invocation. The benchmark-specific values of $K$ and the policy-selection configurations used for the reported runs are provided in Appendix A.

**Policy injection and module execution.** Let $c_{i,j}^{\text{base}}$ denote the original execution context constructed by the task program without NLPG policies. It contains the module instruction, the current task input, and any outputs passed from preceding invocations. The selected instructions $\mathcal{P}_{t,i,j}$

are retrieved from policy memory, formatted as a bounded policy addendum, and appended to the original context:

$$c_{i,j}(M_t) = c_{i,j}^{\text{base}} \parallel \text{Format}(\mathcal{P}_{t,i,j}), \tag{5}$$

where $\parallel$ denotes text concatenation. The module processes the resulting context and produces output $o_{i,j}$, which may subsequently be consumed by downstream modules.

After $J_i$ invocations, the agent returns the final response $y_i$. The observable execution trace is

$$\tau_i = \left(x_i, \{a_{i,j}, c_{i,j}, o_{i,j}\}_{j=1}^{J_i}, y_i\right), \tag{6}$$

where each invocation records its policy route, execution context, and explicit output. The trace may contain generated queries, retrieved evidence, tool outputs, and other observable intermediate results. The benchmark evaluator applies the evaluation to the completed execution and returns the evaluation result $z_i$.

### 3.2 Natural-Language Policy Gradients

Task-level evaluation reveals whether an execution succeeds, but not which module caused a failure. NLPG therefore combines the evaluation result with the observed module dependencies to identify responsible modules and generate route-specific corrections. We call this natural-language description of a local error and its correction direction a **natural-language policy gradient**.

**Failure diagnosis.** We define an evaluated execution as the complete process of running one task example, from its input and module invocations to its final output and evaluation. For benchmark $\mathcal{B}$, the diagnosis function maps execution trace $\tau_i$ and evaluation result $z_i$ to

$$\mathcal{D}_{\mathcal{B}}(\tau_i, z_i) = (\alpha_i, \mathcal{S}_i, e_i), \tag{7}$$

where $\alpha_i$ is the failure weight, $\mathcal{S}_i$ is the set of diagnosed execution-deviation types, and $e_i$ is the identifier of the evaluated execution. This identifier is subsequently used to count how many distinct executions provide evidence for the same candidate correction. In our experiments, $\alpha_i = 0$ for correct executions and $\alpha_i = 1$ for failed executions; records without a valid evaluation are excluded from policy construction.

**Backward propagation of natural-language policy gradients.** Inspired by TextGrad Yuksekgonul et al. (2025), NLPG propagates task-level language feedback $\mathcal{L}_i$ through the observed module-dependency graph in reverse topological order. At a terminal module, the downstream feedback is initialized with the task-level evaluation. At an intermediate module, $F_{i,j}^{\text{down}}$ collects the feedback messages returned by its immediate successor modules. The critic examines these messages together with the current module's input, output, and execution context, produces a localized textual correction, and forwards feedback only to the module's recorded predecessors. This operation propagates feedback within one execution graph; the grouping of corrections across distinct executions is performed separately in Section 3.3. Consequently, for invocation output $o_{i,j}$, the language-level gradient is defined as

$$\frac{\partial^{\text{text}} \mathcal{L}_i}{\partial y_{i,j}} \stackrel{\text{def}}{=} \nabla_{\text{LLM}}\left(o_{i,j}, F_{i,j}^{\text{down}}\right), \tag{8}$$

where $\partial^{\text{text}}$ denotes a language-level feedback transformation rather than a numerical derivative, and $\nabla_{\text{LLM}}$ denotes the critic operation that transforms the current output and downstream feedback into a textual description of how the response $y_{i,j}$ should be improved. When an execution graph is available, NLPG propagates this feedback from terminal modules backward in reverse topological order. At each module, the critic uses the module's output and accumulated downstream feedback to formulate a local correction and pass relevant feedback to its immediate predecessors. Propagation terminates along a branch when no further feedback is assigned, so corrections follow dependencies observed in the execution.

Conditioned on the output-level feedback above, NLPG attributes the required change to the routed policy that produced the module output. We use the dependency $M_t^{a_{i,j}} \to o_{i,j} \to \mathcal{L}_i$ as a structural description of this attribution. The corresponding route-level natural-language gradient is defined as

$$g_{i,j}^{\text{text}} \equiv \frac{\partial^{\text{text}} \mathcal{L}_i}{\partial r_{i,j}} \stackrel{\text{def}}{=} \nabla_{\text{LLM}}\left(M_t^{a_{i,j}}, c_{i,j}, o_{i,j}, F_{i,j}^{\text{down}}\right), \tag{9}$$

Here, $r_{i,j}$ denotes the policy instruction retrieved for the $j$-th invocation, while $M_t^{a_{i,j}}$ denotes the route-specific policy memory from which this instruction is selected; $c_{i,j}$ and $o_{i,j}$ denote the invocation context and the resulting output, respectively, while $F_{i,j}^{\text{down}}$ carries task-level or downstream feedback. The operator $\partial^{\text{text}}$ denotes a language-level transformation rather than a numerical derivative. The critic routes each correction along the observed dependencies, after which NLPG filters and aggregates equivalent instructions across executions. Their priority reflects support, failure weight, and critic confidence, while their effectiveness is assessed by subsequent validation.

### 3.3 Cross-Execution Aggregation and Policy Update

A correction derived from a single execution may reflect an incidental property of the example or its execution path. NLPG therefore aggregates accepted natural-language policy gradients across executions before updating policy memory. Gradients are grouped by route $a$, failure type $s$, and normalized correction text $u$. For each group, let $g_{i,j}^{\text{text}}$ denote the representative gradient, selected by descending failure weight and then critic confidence, where $i, j$ identify the execution and invocation that produced it.

Let $\mathcal{E}_{a,s,u}$ denote the set of distinct evaluated-execution identifiers associated with correction $u$, and let $\mathcal{E}_a$ denote the set of distinct evidence identifiers associated with all accepted corrections on route $a$. Let $N_a$ be the number of valid evaluated records attributed to route $a$. NLPG assigns the candidate correction the score

$$w_{a,s,u} = \frac{|\mathcal{E}_{a,s,u}|}{\max(N_a, |\mathcal{E}_a|, 1)} \, \alpha_i \, \kappa_{i,j}. \tag{10}$$

The three factors represent cross-execution support, failure weight, and critic confidence. We set $\alpha_i = 1$ for failures and 0 otherwise, so the score ranks corrections for retention. The optimizer converts each grouped correction and its diagnosis into a concise route-specific instruction. After safety filtering, accepted instructions are merged with the current policy memory to form the candidate successor $\widehat{M}_{t+1}$.. NLPG promotes this candidate only if paired held-out validation satisfies the gain and regression constraints; otherwise, it retains $M_t$. The promoted memory provides the Top-$K$ policies in the next round.

Overall, NLPG turns execution feedback into validated, route-specific policy updates while keeping the agent's model and program fixed. By making each correction explicit and reusable, it supports interpretable improvement across successive rounds of agent execution.

## 4 Experiments

### 4.1 Experimental Setup

**Experimental Details.** We follow the released protocol for each benchmark. Within every comparison, all methods use matched examples, models, prompts, retrieval settings, and scorers; published results are reused only when these conditions agree and are otherwise recomputed in the NLPG evaluation harness. The policy version remains fixed within each evaluation pass. Full configurations are provided in Appendix C.

**Datasets & Baseline Methods.** We evaluate memory on LoCoMo Maharana et al. (2024), HaluMem Chen et al. (2025), and LongMemEval Wu et al. (2025), against the memory-agent baselines reported for their aligned protocols, including Mem0 Chhikara et al. (2025),MemR$^3$ Du et al. (2025), LightMem Fang et al. (2026), and ProMem Yang et al. (2026). Transfer is evaluated on the GEPA-aligned HotpotQA Yang et al. (2018), IFBench Pyatkin et al. (2025), and HoVer Jiang et al. (2020) suite against the task baseline, GRPO, MIPROv2 Opsahl-Ong et al. (2024), TextGrad Yuksekgonul et al. (2025), Trace/OptoPrime Cheng et al. (2024), and GEPA Agrawal et al. (2026). The complete baseline lists and data construction are deferred to Appendix C.

**Metrics.** LoCoMo and LongMemEval use QA accuracy; HaluMem uses Memory Integrity, Memory Accuracy, and QA Accuracy. HotpotQA uses normalized exact match, IFBench the official strict checker, and HoVer its evidence-grounded verification score. All values are percentages, and transfer-suite averages are arithmetic means over the three tasks. Auxiliary metrics and exact scoring

rules are reported in Appendix C. Complete results for HaluMem and LongMemEval are reported in Appendices C.4 and C.5, respectively.

### 4.2 Long-Horizon Conversational Memory

Table 1: LoCoMo results under the aligned evaluation protocol. Parenthesized values show improvements over the strongest baseline in each column.

| LLM | Method | 1. Multi-Hop | 2. Temporal | 3. Open-Domain | 4. Single-Hop | Overall |
|---|---|---|---|---|---|---|
| GPT-4o-mini | MemoryOS | 56.50 | 37.18 | 40.28 | 62.43 | 54.70 |
| | Mem0 | 58.16 | 55.45 | 40.62 | 66.71 | 61.00 |
| | MemU | 62.41 | 33.96 | 46.88 | 72.77 | 61.15 |
| | MemOS | 69.15 | 72.27 | 60.42 | 81.45 | 75.87 |
| | HiMem | 70.92 | 74.77 | 54.86 | 89.22 | 80.71 |
| | Zep | **71.99** | 74.45 | **66.67** | 88.11 | 81.06 |
| | TiMem | 62.20 | **77.63** | 52.08 | 81.43 | 75.30 |
| | TSM | 66.67 | 71.03 | 58.33 | 84.30 | 76.69 |
| | MemR$^3$ | 71.39 | 76.22 | 61.11 | **89.44** | **81.55** |
| | **NLPG (ours)** | **75.53** (+4.14) | **86.92** (+10.70) | **70.83** (+9.72) | **92.29** (+2.85) | **86.72** (+5.17) |

The category-level results indicate that NLPG is most useful when answering requires a sequence of memory operations rather than a single semantic match. The gains on multi-hop and temporal questions show that route-local policies help the agent combine evidence and preserve time-dependent distinctions across retrieval and answer synthesis. The smaller improvement on single-hop questions is consistent with their lower procedural complexity: these questions often require retrieving one directly relevant fact, leaving less room for module coordination to improve the result.

### 4.3 Transfer to Reasoning, Instruction Following, and Verification

We next test transfer beyond conversational memory on the GEPA-aligned HotpotQA/IFBench/HoVer suite. The comparison includes MIPROv2 Opsahl-Ong et al. (2024), TextGrad Yuksekgonul et al. (2025), Trace/OptoPrime Cheng et al. (2024), and GEPA Agrawal et al. (2026). Results are reported for Qwen3-8B and GPT-4.1 Mini.

Table 2: Cross-model generalization under the GEPA evaluation suite Agrawal et al. (2026). NLPG is evaluated with Qwen3-8B Yang et al. (2025) and GPT-4.1 Mini OpenAI (2025) on HotpotQA, IFBench, and HoVer.

| **Qwen3-8B** | | | | | |
|---|---|---|---|---|---|
| **Method** | **HotpotQA** | **IFBench** | **HoVer** | **3-task Avg.** | **Improvement** |
| Baseline | 42.33 | 36.90 | 35.33 | 38.19 | – |
| GRPO | 43.33 | 35.88 | 38.67 | 39.29 | +1.11 |
| MIPROv2 | 55.33 | 36.22 | 47.33 | 46.29 | +8.11 |
| GEPA | 62.33 | 38.61 | 52.33 | 51.09 | +12.90 |
| GEPA+Merge | 64.33 | 28.23 | 51.67 | 48.08 | +9.89 |
| NLPG | **73.67** | **44.00** | **59.33** | **59.00** | **+20.81** |
| **GPT-4.1 Mini** | | | | | |
| **Method** | **HotpotQA** | **IFBench** | **HoVer** | **3-task Avg.** | **Improvement** |
| Baseline | 38.00 | 47.79 | 46.33 | 44.04 | – |
| Trace (OptoPrime) | 60.33 | 51.19 | 46.00 | 52.51 | +8.47 |
| MIPROv2-No-Demos | 38.00 | 52.04 | 51.33 | 47.12 | +3.08 |
| MIPROv2 | 58.00 | 49.15 | 48.33 | 51.83 | +7.79 |
| TextGrad | 62.33 | 48.64 | 47.67 | 52.88 | +8.84 |
| GEPA | 69.00 | 52.72 | 51.67 | 57.80 | +13.76 |
| GEPA+Merge | 65.67 | 55.95 | 56.67 | 59.43 | +15.39 |
| GEPA-Qwen-Opt | 65.67 | 49.83 | 54.67 | 56.72 | +12.68 |
| NLPG | **72.67** | **58.00** | **59.33** | **63.33** | **+19.29** |

NLPG is best on all three tasks for both backbones, averaging 59.00 with Qwen3-8B and 63.33 with GPT-4.1 Mini. Diagnostic breakdowns are provided in Appendix C.7. The transfer results show that

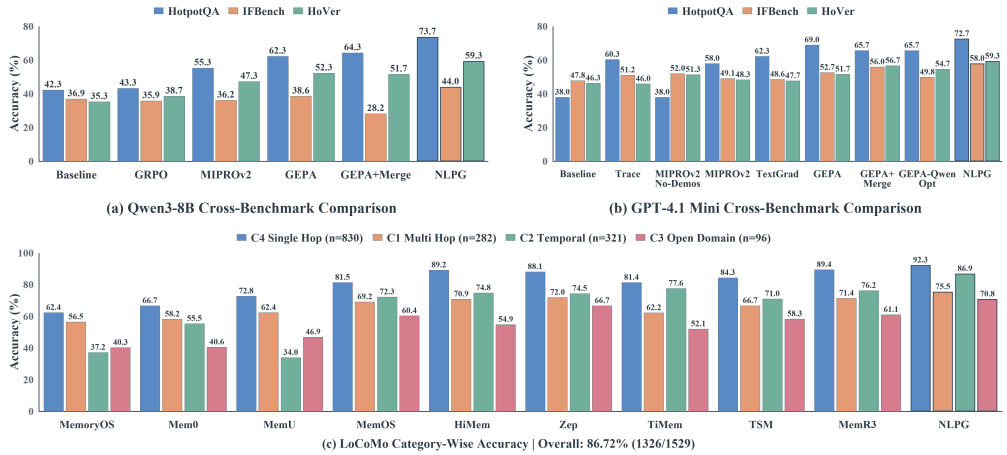



Figure 3: Cross-benchmark accuracy comparison. Panels (a) and (b) show transfer results for Qwen3-8B and GPT-4.1 Mini, respectively, while panel (c) reports category-wise LoCoMo accuracy. Consistent with Tables 2 and 1, NLPG achieves the best performance on all transfer tasks for both backbones and improves every LoCoMo category, with the largest gains on multi-hop, temporal, and open-domain questions. The strongest improvements occur on procedurally compositional questions, suggesting that route-specific policy updates improve decisions about query preservation, memory combination, and retrieval expansion rather than simply increasing context size.

the benefit is not tied to conversational-memory retrieval. NLPG improves all three tasks for both backbones, including instruction following, multi-hop retrieval, and evidence-grounded verification. Because the baseline and NLPG use the same backbone, examples, task procedure, retrieval settings, generation limits, and evaluation metric in each comparison, NLPG does not receive extra model calls or a larger inference budget. Under the matched comparison protocol, the results are consistent with a contribution from the external policy memory and update mechanism, rather than from a different answer format or more computation.

### 4.4 Cross-Benchmark Analysis

Across the six independently held-out evaluations, NLPG improves both long-horizon memory use and compound-task execution. Policies are constructed and selected without access to final-test examples.

Figure 3 summarizes the results. NLPG reaches $86.72$ on LoCoMo, with the largest gains on multi-hop, temporal, and open-domain questions; the latter category is small and should be interpreted cautiously. It also obtains $96.20$ Memory Accuracy and $85.37$ QA Accuracy on HaluMem, and $71.80$ QA Accuracy on LongMemEval. The transfer gains persist across both backbones: on HotpotQA, IFBench, and HoVer, NLPG improves over the corresponding baseline by $31.34, 6.43, 24.00$ points with Qwen3-8B and $34.67, 10.21, 13.00$ points with GPT-4.1 Mini. Because NLPG improves tasks with different evidence-routing and instruction-following requirements using the same failure-to-correction update procedure, the result supports the portability of this procedure rather than the reuse of a single benchmark-specific answer format or policy text.

### 4.5 Ablation Study

We ablate NLPG's core mechanisms and route-local memory capacity on the same 150-example IFBench split, holding the model, evaluation protocol, and inference budget fixed.

Figure 4 summarizes the contribution of each component. Removing textual gradients causes the largest drop $5.3\%$, showing that natural-language corrections provide actionable guidance beyond failure counts. Removing policy updates or failure signatures reduces accuracy by 4.0 percentage points, highlighting the importance of cross-example feedback accumulation and explicit failure




Figure 4: IFBench ablations by instruction category. **(a)** Strict accuracy (%). **(b)** Difference from full NLPG in percentage points. Hatched cells contain fewer than 10 examples.

diagnosis. Removing route credit causes a further 3.3-point drop, providing evidence for the value of module-specific credit assignment.

### 4.6 Analysis of Module-Local Policy Intervention

**Why specificity policies work.** We compare the same case with and without NLPG's route-specific specificity policy, keeping the question, backbone, retrieval setup, and QA program fixed.

**Case Study: NLPG Policy Intervention for Answer Specificity**

**Question:** Trey Anastasio and Glenn Bidmead, have which mutual occupation?
**Gold answer:** guitarist

**Without NLPG Policy**

**Diagnosis:** answer overgeneralization

**Hop-2 query:**
What are the shared occupations of Trey Anastasio and Glenn Bidmead?

→ **Predicted answer:** musician

× Broader umbrella label

**With Specificity Policy**

**Policy update:** preserve the most specific shared attribute

**Hop-2 query:**
What occupation do Trey Anastasio and Glenn Bidmead share?

→ **Predicted answer:** guitarist

✓ Specific answer recovered

Supporting-document recall remains 1.0 in both runs, so the repair does not come from additional evidence. Instead, the specificity policy changes the hop-2 query and replaces the broad label "musician" with the exact occupation "guitarist", consistent with the diagnosed overgeneralization error. As a single case with multi-module injection, it neither isolates one module's contribution nor provides a population-level causal estimate.

## 5 Conclusion

We introduced Natural-Language Policy Gradients (NLPG), a lightweight framework that converts execution failures into reusable, module-local natural-language policies for compound LLM agents. By combining failure diagnosis, route-aware policy updates, bounded selection, and offline validation, NLPG improves both conversational-memory and multi-hop reasoning tasks without changing model parameters or retrieval backends. Our results and ablations show that textual feedback and module-specific credit assignment are key to these gains, while a controlled intervention confirms that targeted policies can improve answer specificity under fixed retrieval. Future work will explore more automatic failure routing, larger-scale policy evolution, and extensions to multimodal and larger tool-using agents.

## AI USE STATEMENT

AI-based assistants were used to support language polishing, retrieval and discovery, and code-review tasks during the preparation of this manuscript. All algorithmic designs, implementations, experiments, analyses, citations, and final manuscript content were checked and verified by the authors, who remain responsible for the accuracy and integrity of this work.

## ETHICS STATEMENT

NLPG improves LLM agents by extracting natural-language policies from prior executions and applying them to subsequent tasks. Although this can improve consistency and task performance, storing execution traces and derived policies may introduce privacy risks when inputs contain sensitive information. The learned policies may also preserve or amplify biases, errors, or unsupported assumptions present in the data or model outputs. Therefore, deployment should use appropriate safeguards, including user consent, data minimization, anonymization, access control, policy validation, and human oversight. We use public benchmark datasets solely for research evaluation and do not recommend deploying NLPG for high-stakes decisions without additional safety review.

## REPRODUCIBILITY STATEMENT

To support reproducibility, we describe the NLPG algorithm and execution process in Section 3 and provide implementation details, prompt templates, model assignments, data-separation protocols, benchmark adapters, and evaluation scripts in Appendices A and C. We report the model identifiers, random seeds, data splits, policy-memory capacity, inference settings, scoring procedures, and evaluation denominators used in the experiments. The policy-construction, selection, and final-test partitions are disjoint, and the selected policy is frozen during final evaluation. We plan to release the source code and configuration files to facilitate independent verification and replication of the reported results.

## REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/0e9e708b6f48e14fd0ac29e167413f76-Abstract-Conference.html.

Ding Chen, Simin Niu, Kehang Li, Peng Liu, Xiangping Zheng, Bo Tang, Xinchi Li, Feiyu Xiong, and Zhiyu Li. HaluMem: Evaluating hallucinations in memory systems of agents. *arXiv preprint arXiv:2511.03506*, 2025. URL https://arxiv.org/abs/2511.03506.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/83ba7056bce2c3c3c27e17397cf3e1f0-Abstract-Conference.html.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. In *ECAI 2025*, volume 413 of *Frontiers in Artificial Intelligence and Applications*, pp. 2993–3000. IOS Press, 2025. doi: 10.3233/FAIA251160. URL https://ebooks.iospress.nl/doi/10.3233/FAIA251160.

Xingbo Du, Loka Li, Duzhen Zhang, and Le Song. MemR$^3$: Memory retrieval via reflective reasoning for LLM agents. *arXiv preprint arXiv:2512.20237*, 2025. URL https://arxiv.org/abs/2512.20237.

Jizhan Fang, Xinle Deng, Haoming Xu, Ziyan Jiang, Yuqi Tang, Ziwen Xu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Huajun Chen, and Ningyu Zhang. LightMem: Lightweight

and efficient memory-augmented generation. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/a05b72653ec5b473732129829ae04195-Abstract-Conference.html.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 13481–13544, 2024. URL https://proceedings.mlr.press/v235/fernando24a.html.

Yichen Jiang, Shikha Bordia, Zheng Zhong, Charles Dognin, Maneesh Singh, and Mohit Bansal. HoVer: A dataset for many-hop fact extraction and claim verification. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 3441–3460, 2020. doi: 10.18653/v1/2020.findings-emnlp.309. URL https://aclanthology.org/2020.findings-emnlp.309/.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=sY5N0zY5Od.

Aman Madaan, Niket Tandon, Peter Clark, and Yiming Yang. Memory-assisted prompt editing to improve GPT-3 after deployment. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 2833–2861, 2022. doi: 10.18653/v1/2022.emnlp-main.183. URL https://aclanthology.org/2022.emnlp-main.183/.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870, 2024. doi: 10.18653/v1/2024.acl-long.747. URL https://aclanthology.org/2024.acl-long.747/.

OpenAI. Introducing GPT-4.1 in the API. OpenAI, 2025. URL https://openai.com/index/gpt-4-1/.

Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, 2024. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. URL https://arxiv.org/abs/2310.08560.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, 2023. doi: 10.18653/

v1/2023.emnlp-main.494. URL https://aclanthology.org/2023.emnlp-main.494/.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hannaneh Hajishirzi. Generalizing verifiable instruction following. In *Advances in Neural Information Processing Systems*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/46499a0622ecf568b72d17b61e45dbd5-Abstract-Datasets_and_Benchmarks_Track.html.

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. In *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018. URL https://mitpress.mit.edu/9780262039246/reinforcement-learning/.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=pZiyCaVuti.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed H. Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *Proceedings of the First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

An Yang et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

Chengyuan Yang, Zequn Sun, Wei Wei, and Wei Hu. Beyond static summarization: Proactive memory extraction for LLM agents. *arXiv preprint arXiv:2601.04463*, 2026. URL https://arxiv.org/abs/2601.04463.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, 2018. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639:609–616, 2025. doi: 10.1038/s41586-025-08661-4. URL https://doi.org/10.1038/s41586-025-08661-4.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2211.01910.

# APPENDIX CONTENTS

**A Implementation Details** **14**

A.1 Inference with a Frozen Policy . . . . . 15

A.2 Offline Policy Construction . . . . . 15

A.3 Benchmark Adapters and Policy Injection . . . . . 16

A.4 Policy Lifecycle and Held-Out Evaluation . . . . . 17

A.5 Benchmark Execution and Retrieval . . . . . 17

A.6 Computational Overhead . . . . . 17

**B Related Work** **17**

**C Experimental Details** **18**

C.1 Data Separation and Result Provenance . . . . . 18

C.2 Benchmark and Model Alignment . . . . . 18

C.3 LoCoMo . . . . . 19

C.3.1 Paired Ablation Analysis on LoCoMo . . . . . 20

C.3.2 Category-level Analysis of Answer-stage Ablations on LoCoMo . . . . . 20

C.4 Faithful Memory Construction . . . . . 20

C.5 Long-Term Memory Question Answering . . . . . 21

C.6 GEPA-Aligned HotpotQA . . . . . 22

C.7 IFBench and HoVer . . . . . 23

C.7.1 Feedback Quality and Repair Analysis on IFBench . . . . . 23

C.8 Evaluation Metrics . . . . . 25

C.9 Policy-Memory Runtime . . . . . 25

C.10 Controlled Policy Intervention . . . . . 26

**D Prompt Templates** **26**

D.1 Application-Level Prompts . . . . . 26

D.2 Module-Local Policy Injection . . . . . 27

D.3 LLM-Based Policy Reflection . . . . . 27

D.4 Execution-Graph Diagnosis and Route Attribution . . . . . 28

D.5 Policy Initialization and Update . . . . . 29

D.6 Structured Module Outputs . . . . . 29

D.7 Prompt and Policy Budget . . . . . 30

# A IMPLEMENTATION DETAILS

NLPG adds an external procedural policy layer to a fixed compound agent. The benchmark program remains responsible for memory construction, retrieval, generation, and evaluation, while NLPG controls how reusable policy instructions are selected, updated, and injected into the program. Model parameters, program structure, retrieval indexes, and benchmark evaluators are kept

**Algorithm 1** Inference with a frozen route-indexed policy

**Require:** Task instance $x$, active policy memory $M_t$, budget $K$
**Ensure:** Response $\hat{y}$ and execution record $\tau$
1: $a \leftarrow \mathrm{Route}(x)$
2: $\mathcal{P} \leftarrow \mathrm{TopK}(\mathrm{Dedup}(M_t^a \cup M_t^{\mathrm{gen}}), K)$
3: $\tau \leftarrow \mathrm{Execute}(P_\theta, x, \mathcal{P})$
4: $\hat{y} \leftarrow \mathrm{Response}(\tau)$
5: $z \leftarrow \mathrm{Evaluate}(\tau)$
6: **return** $\hat{y}, (\tau, z)$

Table 3: Conceptual data flow of the NLPG execution and update process.

| Stage | Input → Output | Role in NLPG |
|---|---|---|
| Policy loading | $M_t \rightarrow$ route-indexed policy candidates | Load the active procedural policy version. |
| Policy selection | Task input and route → top-$K$ instructions | Deduplicate, rank, and select the policy context for the invocation. |
| Fixed execution | Task input and selected policy → execution trace | Run the original benchmark program without changing its structure. |
| Evaluation | Execution trace → $z_i$ | Apply the benchmark's released checker or judge. |
| Failure analysis | $(\tau_i, z_i) \rightarrow$ failure feedback | Identify the deviation and the routes relevant to its correction. |
| Policy update | Failure feedback → textual corrections | Generate and consolidate reusable route-specific instructions. |
| Policy validation | Candidate policy → accepted or rejected version | Evaluate the candidate on a disjoint policy-selection set. |

fixed. All reported NLPG results use the complete pipeline described below; component removals are used only in the ablation experiments.

### A.1 Inference with a Frozen Policy

During evaluation pass $t$, the agent processes each task instance using the frozen policy memory $M_t$. The memory is indexed by execution routes, which may correspond to task families, program modules, or a shared general route. For the current route $a$, NLPG removes duplicate instructions, ranks the remaining candidates, and injects at most $K$ instructions into the corresponding program context. The fixed agent then executes its original retrieval and generation procedure.

Here, $M_t^{\mathrm{gen}}$ is empty for benchmarks without a shared policy route. The selected instructions are appended to the relevant module or task context, while the ordering of the original program is unchanged. Each execution record stores the observable inputs and outputs, route information, retrieved evidence, policy instructions, execution dependencies, final response, and evaluation result. The active policy is not modified while the current evaluation pass is running.

### A.2 Offline Policy Construction

Policy construction takes place after an evaluation pass has finished. The observed execution records are converted into failure feedback associated with the routes that could have contributed to the deviation. When an execution graph is available, feedback is propagated from terminal modules toward their recorded predecessors in reverse topological order. The critic uses the local execution context, module output, and downstream feedback to produce a route-specific textual correction. Corrections from different executions are then grouped by route and normalized instruction content.

For a grouped correction $u$ associated with route $a$ and failure signature $s$, NLPG assigns the following retention score:

$$w_{a,s,u} = \frac{|\mathcal{E}_{a,s,u}|}{\max(N_a, |\mathcal{E}_a|, 1)} \alpha_{a,s,u} \kappa_{a,s,u}, \tag{11}$$

**Algorithm 2** Route-aware offline policy construction

**Require:** Completed records $\mathcal{R}$ and active policy $M_t$
**Ensure:** Candidate policy $\widehat{M}_{t+1}$
1: $\mathcal{D} \leftarrow \mathrm{Diagnose}(\mathcal{R})$
2: $\mathcal{G} \leftarrow \mathrm{Critic}(\mathcal{D}, M_t)$
3: $\mathcal{G}' \leftarrow \mathrm{Consolidate}(\mathcal{G})$
4: $\mathcal{G}_{\mathrm{safe}} \leftarrow \mathrm{Filter}(\mathcal{G}')$
5: $\widehat{M}_{t+1} \leftarrow \mathrm{Merge}(M_t, \mathcal{G}_{\mathrm{safe}})$
6: **return** $\widehat{M}_{t+1}$

where $\mathcal{E}_{a,s,u}$ is the set of distinct executions supporting the correction, $N_a$ is the number of valid evaluated executions associated with route $a$, and $\mathcal{E}_a$ is the set of executions contributing accepted corrections on that route. The terms $\alpha_{a,s,u}$ and $\kappa_{a,s,u}$ denote the normalized failure weight and critic confidence. The score is used to rank candidates for retention; it is not interpreted as a causal estimate of correction utility.

The safety stage removes empty, duplicate, overly specific, overlong, label-leaking, or unsafe instructions and limits the number of retained candidates per route. The resulting object is a candidate successor policy. It becomes active only after passing policy selection; otherwise, the current policy is retained for the subsequent evaluation pass.

### A.3 Benchmark Adapters and Policy Injection

NLPG uses a common policy memory and update procedure across benchmarks. Since the evaluated programs expose different execution modules and evaluation signals, each benchmark is connected through an adapter that translates its execution trace and evaluation result into a common NLPG representation. This representation contains the evaluation signal, observed failure information, route metadata, and execution evidence required for policy improvement.

The adapter maps each benchmark's execution record and evaluator output to the common NLPG representation, including the normalized evaluation signal, failure descriptor, task family, and eligible policy routes. It therefore performs benchmark-specific diagnosis and route assignment, but does not generate the policy text or aggregate updates across executions. The shared NLPG critic combines these normalized signals with the observable execution evidence to produce reusable route-local corrections. The shared optimizer then groups and ranks compatible corrections across executions, while the safety stage removes duplicated, overly specific, label-leaking, or unsafe instructions before the next policy version is constructed.

Route names follow the vocabulary of the fixed program. Memory-oriented benchmarks generally expose task-family routes, whereas multi-hop and instruction-following programs expose module- or constraint-specific routes. This route conditioning allows feedback to be assigned at the level where the corresponding behavior is executed, rather than pooling all corrections into a single prompt.

Table 4: Execution routes and policy injection points used by the evaluated benchmark programs.

| Benchmark | Route representation | Policy injection point | $K$ |
|---|---|---|---|
| LoCoMo | Question category | Retrieval and answer guidance | 4 |
| HaluMem | Inferred question family | Memory retrieval and answer policy | 4 |
| LongMemEval | Question type with fallback family | Retrieval and answer guidance | 4 |
| HotpotQA | Module routes in the two-hop program | Summarization, second-hop query, and answer modules | 3 |
| IFBench | Instruction identifier and instruction family | Response-generation context | 3 |
| HoVer | Hop-specific summary and query modules | Multi-hop summary and query generation | 3 |

Reference information is consumed by the benchmark evaluator and, where required by the benchmark protocol, is converted into a normalized evaluation signal or failure descriptor before policy construction. For example, supporting-document annotations are used to compute document coverage in HotpotQA and HoVer, while the official checker identifies unsatisfied instruction constraints in IFBench. These evaluator-derived signals may determine the failure signature and eligible policy routes. The critic does not receive the raw gold answer, supporting-document annotations, or reference text; it receives the observable execution record together with the normalized evaluation feedback and failure descriptors. Policy instructions are generated as reusable behavioral corrections.

### A.4 Policy Lifecycle and Held-Out Evaluation

NLPG maintains one active policy version during each evaluation pass. After the pass, the collected records are used to construct a candidate successor policy. In the held-out evaluation protocol, policy construction, policy selection, and final testing use disjoint data partitions. The candidate and its parent are compared under the same model configuration, program, and evaluation criterion on the policy-selection partition. A candidate that fails the required improvement or regression constraints is discarded; the previous policy remains active.

After policy selection is completed, the adopted policy is evaluated once on the final held-out test partition. Test executions do not contribute to policy construction or policy selection. Each policy file records its version, parent, route candidates, and the identifiers of the executions used to construct and evaluate it, allowing partition overlap to be checked directly.

### A.5 Benchmark Execution and Retrieval

Model assignments are benchmark configurations rather than components of the NLPG method. NLPG does not introduce additional model calls during ordinary policy injection and does not alter the benchmark retrieval index. For HotpotQA, the fixed program performs two retrieval stages with intermediate summarization and second-hop query generation before answer synthesis. NLPG is injected into the language-generation modules only.

HoVer uses a fixed multi-hop retrieval program with hop-specific summaries and queries. When a local corpus index is enabled, document retrieval is performed by the benchmark's configured lexical index, while language models produce summaries and queries. Supporting-document coverage and document recall are recorded as retrieval diagnostics, and the final verification score is computed by the benchmark evaluator. Complete model assignments, retrieval budgets, dataset partitions, and scoring rules are reported in Appendix C.

### A.6 Computational Overhead

For a route containing $C_a$ policy candidates, candidate ranking costs $O(C_a \log C_a)$ and formatting the selected $K$ instructions costs $O(KL)$, where $L$ is their mean length. This overhead is bounded by the policy budget and adds no mandatory retrieval operation.

Offline construction is linear in the number of inspected execution records and generated corrections, followed by route-wise candidate ranking. Critic and graph-feedback operations add language-model calls only during policy construction. The main configuration bounds the number and length of retained instructions and corrections, ensuring that the policy context remains bounded across evaluation passes.

## B Related Work

**Compound agents.** Compound language agents coordinate multiple language-model calls with retrieval, tool use, and verification. ReAct interleaves reasoning and acting (Yao et al., 2023), Toolformer studies tool use by language models (Schick et al., 2023), AutoGen supports multi-agent coordination (Wu et al., 2024), and DSPy represents language-model applications as composable programs (Khattab et al., 2024). These works establish the execution structure of compound agents,

but, to our knowledge, have paid less attention to explicit post-deployment, module-level policy updates in compound agents.

**External memory.** Memory-augmented agents preserve information across interactions through experience records, reflection, or hierarchical storage (Park et al., 2023; Packer et al., 2023). Mem0, LightMem, and ProMem further improve memory extraction, compression, retrieval, and verification (Chhikara et al., 2025; Fang et al., 2026; Yang et al., 2026). These methods mainly manage historical content or factual memories. NLPG instead stores reusable procedural instructions indexed by task family or program module and updates them from diagnosed execution failures.

**Language-based agent optimization.** Reflexion and Self-Refine use verbal feedback to improve later attempts (Shinn et al., 2023; Madaan et al., 2023). Prompt-optimization methods search over instructions through textual edits, language-model optimization, or evolutionary search (Pryzant et al., 2023; Yang et al., 2024; Opsahl-Ong et al., 2024; Fernando et al., 2024). TextGrad formalizes language feedback as a textual gradient, while Trace uses execution traces and rich feedback for generative optimization (Yuksekgonul et al., 2025). GEPA extends this direction to reflective prompt evolution for compound systems (Agrawal et al., 2026).

# C Experimental Details

This appendix records the experimental settings, model assignments, data flow, and evaluation protocols for NLPG. Comparisons within one block use the same dataset version, final-test examples, task interface, candidate corpus, retrieval budget, answer format, evaluator, normalization rule, and scoring denominator.

All NLPG results are evaluated on an independently held-out final set. Before policy construction, each benchmark is partitioned into a construction set, a selection set, and a final test set. Evaluated trajectories from the construction set are used to produce candidate policies, whereas the selection set is used only to choose between candidate and parent policy versions. The selected policy is then frozen and evaluated once on the final test set. Final- test inputs, outputs, labels, and evaluator feedback are never used to construct, select, revise, or rescore the policy reported for that set.

## C.1 Data Separation and Result Provenance

Table 5 summarizes the separation protocol. Splits are materialized before optimization and stored with stable example identifiers. For benchmarks containing multiple questions about the same conversation, user, memory record, or document history, separation is performed at that higher-level unit rather than only at the question level. Consequently, no underlying context unit can contribute feedback to policy construction or selection and later reappear in the final test set.

The same final-test manifest is used for every method in a comparison block. Published baseline values are reused only when their split and evaluation protocol match that manifest; otherwise, the baseline is rerun on the shared final set. Split manifests are fixed before any policy update, and overlap is checked using both example identifiers and the benchmark-specific grouping keys listed in Table 5.

## C.2 Benchmark and Model Alignment

We evaluate NLPG on six benchmarks covering conversational memory, faithful memory construction, long-term memory question answering, and general compound-agent optimization. The memory benchmarks are LoCoMo Maharana et al. (2024), HaluMem Chen et al. (2025), and LongMemEval Wu et al. (2025). The transfer benchmarks are GEPA-aligned HotpotQA Yang et al. (2018), IFBench Pyatkin et al. (2025), and HoVer Jiang et al. (2020).

All results are organized into comparison blocks. A comparison block is defined by a benchmark, a fixed dataset subset, a task interface, and a specified model assignment. Results from different blocks are not merged when their backbone, evaluator, or runtime protocol differs.

For LoCoMo, the shared comparison uses the same held-out first-user questions and the same memory-building, answer-generation, and judging configuration for every method in the correspond-

Table 5: Independent held-out evaluation protocol. Construction, selection, and final-test partitions are mutually disjoint under the indicated grouping key.

| Benchmark | Policy construction | Policy selection | Final test | Separation key |
|---|---|---|---|---|
| LoCoMo | Non-test conversation–user groups | Disjoint conversation–user groups | 152 first-user questions from held-out groups | Conversation and user ID |
| HaluMem | Non-test user records | Disjoint validation user records | 164 questions from held-out user records | User and record ID |
| LongMemEval | Non-test history–question groups | Disjoint validation groups | Held-out LongMemEval-S questions | History and question ID |
| HotpotQA | 150 train examples | 300 validation examples | 300 test examples | Example ID |
| IFBench | Construction portion of calibration data | Disjoint selection portion | 150 held-out examples | Prompt/example ID |
| HoVer | Construction portion of 150 train examples | Disjoint selection portion | 300 test examples | Claim ID |

ing table. No conversation or user record from this final partition is used during policy construction or selection.

For HotpotQA, IFBench, and HoVer, each model block uses one fixed task backbone for all task modules in that block. Auxiliary models used for reflection, criticism, or policy optimization are reported separately and are not treated as the task backbone. In particular, a model used by an auxiliary critic or optimizer is not used to relabel the backbone of the corresponding task result.

### C.3 LoCoMo

LoCoMo evaluates question answering over long conversational histories Maharana et al. (2024). We use the complete released LoCoMo evaluation set, including all valid questions and the corresponding conversation histories, user identifiers, and question categories. Every method receives the same conversation history and question and produces an answer through the same answer interface.

The NLPG policy is constructed and selected using conversation–user groups disjoint from the final evaluation partition. The selected policy is frozen before evaluation on the complete held-out LoCoMo test set. Conversation, user, and question identifiers are checked for overlap before scoring, and final-test trajectories and judgments do not enter any subsequent policy update.

The NLPG policy is constructed and selected using conversation–user groups that are disjoint from the final evaluation partition. The selected policy is then frozen and evaluated once on the 152 held-out first-user questions from categories 1–4. Conversation, user, and question identifiers are checked for overlap before scoring; trajectories and judgments from these 152 questions do not enter a subsequent update of the reported policy.

The comparison includes MemoryOS, Mem0 Chhikara et al. (2025), MemU, MemOS, HiMem, Zep, TiMem, TSM, MemR3, and NLPG. Baseline values are reused only when the published or released configuration matches the comparison protocol. Otherwise, the method is evaluated in the shared harness using the same questions, answer format, evaluator, and denominator.

The LoCoMo table uses the model assignment specified by the comparison block. Memory construction, answer generation, and judging use the same corresponding models for all methods, and only the method-specific memory or policy mechanism is changed.

LoCoMo accuracy is computed as the proportion of valid questions judged correct. Percentages are reported at the precision used by the corresponding result table. When a source reports a rounded percentage, the value is retained as reported rather than being reverse-converted into an integer count.

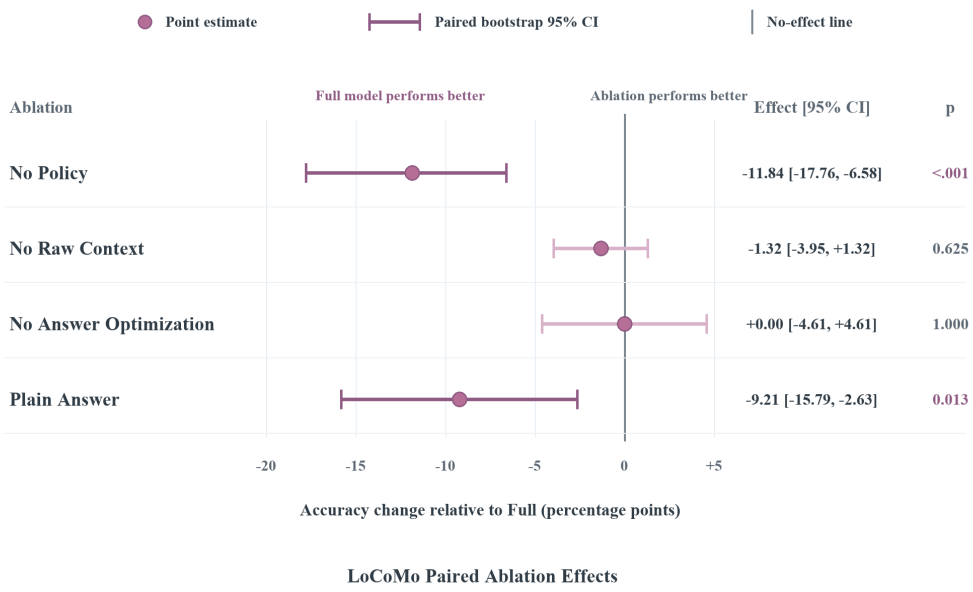



Figure 5: Paired answer-stage ablations on LoCoMo ($n = 152$). Points denote accuracy changes relative to the full model, and horizontal bars show paired-bootstrap 95% confidence intervals. Negative values favor the full model, while the vertical line denotes no effect. Reported $p$-values are from two-sided exact McNemar tests. Retrieval results are fixed and cached across all variants.

### C.3.1 Paired Ablation Analysis on LoCoMo

We further conduct paired ablations on the same 152 held-out LoCoMo questions under fixed cached retrieval. As shown in Figure 5, removing the policy causes a significant accuracy drop of 11.84 percentage points, while replacing the optimized answer stage with plain answering reduces accuracy by 9.21 points. In contrast, removing raw context or answer optimization alone does not yield a statistically significant difference. These results highlight the importance of policy and the complete answer-generation design.

### C.3.2 Category-level Analysis of Answer-stage Ablations on LoCoMo

To examine whether answer-stage components affect different question types differently, we report category-level accuracy changes relative to the full configuration under fixed cached retrieval. As shown in Figure 6, removing the raw conversational context causes the most consistent degradation across all categories, with drops ranging from 7.69 to 16.22 percentage points. Removing answer optimization produces comparatively small changes, while the effects of removing policy prompts are mixed across categories. The Open Domain results should be interpreted cautiously because this category contains only 13 questions.

## C.4 Faithful Memory Construction

HaluMem jointly evaluates the integrity and factual accuracy of constructed memories and their downstream usefulness for question answering Chen et al. (2025). We use the released examples, memory targets, and downstream questions. All methods in the comparison use the same examples, memory targets, questions, answer interface, evaluator, and scoring denominator.

HaluMem is partitioned by user record before policy learning. Policy construction and selection use disjoint non-test users, and the selected policy is evaluated on 164 questions associated only with held-out users. This user-level partition prevents the same profile, conversation history, memory target, or downstream question from appearing in more than one data role.

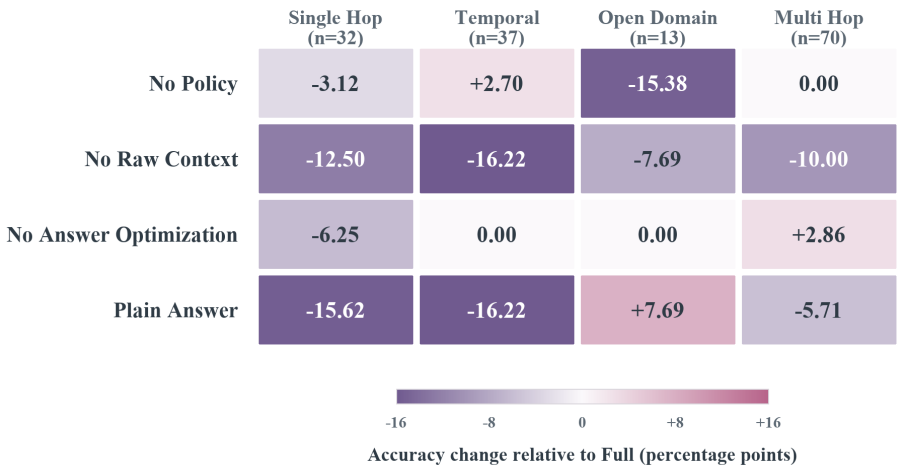



Figure 6: Category-level changes in accuracy, measured in percentage points relative to the full configuration, for answer-stage ablations on the held-out LoCoMo set with fixed cached retrieval. Negative values indicate that the ablated configuration performs worse than the full configuration. The numbers in parentheses denote the number of questions in each category. The Open Domain column is interpreted cautiously because this category contains only 13 questions.

The three reported metrics are Memory Integrity, Memory Accuracy, and downstream QA Accuracy. Memory Integrity measures whether the constructed memory preserves the required information without introducing invalid content. Memory Accuracy evaluates the correctness of the stored memory relative to the benchmark target. QA Accuracy measures whether the system answers downstream questions correctly using the constructed memory.

Table 6: Results on HaluMem under the aligned benchmark evaluation protocol. The three metrics are reported separately because they measure different properties of memory construction and downstream use.

| Method | Memory Integrity | Memory Accuracy | QA Accuracy |
|---|---|---|---|
| MemoBase | 14.55 | 92.24 | 35.53 |
| Supermemory | 41.53 | 90.32 | 54.07 |
| Mem0 | 42.91 | 86.26 | 53.02 |
| ProMem | **73.80** | 89.47 | 62.26 |
| NLPG | 70.25 | **96.20** | **85.37** |

On the independent held-out set, NLPG obtains the highest Memory Accuracy and QA Accuracy among the listed values, while ProMem obtains the highest Memory Integrity.

### C.5 Long-Term Memory Question Answering

LongMemEval evaluates question answering over extended interaction histories in which relevant evidence may be distributed across multiple sessions Wu et al. (2025). We use the same histories, questions, valid-question denominator, answer interface, evaluator, and accuracy definition for all methods in the aligned comparison. The comparison follows the protocol used for NativeRAG, Mem0, LightMem, and ProMem.

LongMemEval is partitioned before policy learning using both question and history identifiers. Construction and selection operate on mutually disjoint non-test groups, and the frozen selected policy is evaluated only on the held-out LongMemEval-S groups. All baselines are scored on the same final-test manifest and denominator.

Table 7: Results on LongMemEval under the aligned question-answering protocol.

| Method | QA Accuracy |
|---|---|
| NativeRAG | 65.09 |
| Mem0 | 43.31 |
| LightMem | 68.64 |
| ProMem | 69.57 |
| NLPG | **71.80** |

NLPG is 2.23 points above ProMem and 3.16 points above LightMem on the shared held-out evaluation set.

### C.6 GEPA-ALIGNED HOTPOTQA

**Dataset construction.** To match the referenced GEPA task setting, we load the train split of the HuggingFace HotpotQA fullwiki collection as the base pool. Following the shared splitter, the first 40% is used as test, the next 40% as validation, and the final 20% as train. A random seed of 1 is then used to retain 300 test examples, 300 validation examples, and 150 train examples.

This construction aligns the data partition and example counts with the referenced GEPA setting. Each method is evaluated with the same per-example corpus and two-hop program interface within the corresponding comparison block.

The 150-example train split is used exclusively for policy construction, the 300-example validation split is used for policy selection, and the selected policy is frozen before a single pass over the 300-example test split. No test trajectory, gold answer, supporting-document annotation, or evaluator outcome is available to the critic or optimizer before the final score is recorded.

**Runtime configuration.** The reported main run evaluates 300 test examples. Retrieval uses top-$k = 5$ for both hop 1 and hop 2. The summary, retrieval, and answer modules use the backbone specified by the relevant model block. Time filtering and explicit thinking are disabled. NLPG selects at most $K = 3$ policy instructions per module, subject to a maximum policy-context length of 1,000 characters.

Table 8: Main configuration of the 300-example HotpotQA evaluation.

| Setting | Value |
|---|---|
| Dataset | HotpotQA fullwiki, GEPA-aligned split policy |
| Evaluation split | Test; 300 examples; subset seed 1 |
| Data partition | 40% test, 40% validation, 20% train |
| Retrieval | Two-hop retrieval; top-$k = 5$ for each hop |
| NLPG policy context | At most 3 instructions per module; maximum 1,000 characters |
| Time filtering | Disabled |
| Explicit thinking | Disabled |
| Primary metric | Exact match after lowercasing, punctuation normalization, and article removal |
| Auxiliary diagnostics | Answer containment, full supporting-document coverage, and mean document recall |

**Metrics and diagnostics.** For an answer to be counted as correct, the normalized prediction must exactly match the normalized gold answer. Document recall is computed over the unique gold supporting titles. Full supporting-document coverage requires every gold supporting title to appear in the deduplicated two-hop retrieval result.

On the 300-example held-out test set, the frozen selected policy obtains 221 exact matches, corresponding to 73.67% accuracy. The same run obtains 89.67% full-document coverage and 94.33% mean document recall.

The offline two-hop analysis further separates retrieval failures from answer synthesis failures. Among 82 examples requiring a second-hop target, 31 examples miss the second-hop page, while

51 recover all missing targets. Separately, 59 examples are incorrect despite complete supporting-document coverage, indicating an answer-synthesis error rather than a retrieval miss.

### C.7 IFBench and HoVer

IFBench evaluates instruction-following behavior using its official task construction and strict checker Pyatkin et al. (2025). We use the released examples and evaluate all methods with the same instruction format, output interface, checker, and scoring denominator. A prediction is counted as correct only when it satisfies the constraints required by the official checker.

For NLPG, the calibration data are partitioned into disjoint construction and selection subsets before optimization. A separate 150-example held-out split is used only for final evaluation. The three partitions contain no shared prompt or example identifiers, and feedback from the held-out pass is not used to revise or rescore the reported policy.

HoVer evaluates evidence-grounded multi-hop verification Jiang et al. (2020). We use the released claims, evidence requirements, and verification protocol. The evidence retrieval, summary, and verification interfaces are fixed within each model block. If an auxiliary model is used for policy diagnosis or optimization, it is recorded separately from the task-model assignment.

The 150-example training subset is divided into mutually disjoint construction and selection portions. The selected NLPG policy is then frozen for a 300-example test subset. Claim identifiers have zero overlap across the three partitions, and no failure labels or trajectories from the test subset are used to select, rewrite, or rescore the reported policy.

For both benchmarks, the comparison includes the task baseline, GRPO, MIPROv2 Opsahl-Ong et al. (2024), TextGrad Yuksekgonul et al. (2025), Trace/OptoPrime Cheng et al. (2024), GEPA Agrawal et al. (2026), and the corresponding reported variants where applicable. All methods within a model block use the same task backbone and task interface.

#### C.7.1 Feedback Quality and Repair Analysis on IFBench

We evaluate whether NLPG feedback correctly identifies failed constraints and whether the resulting instructions can repair an unsuccessful response. We use the calibration split of IFBench, which contains 150 examples, and evaluate all responses using the official strict and loose IFBench checkers. The response model is Qwen3-8B, with temperature set to 0 and a maximum generation length of 512 tokens. Diagnoses are produced by the complete Agentic-NLPG pipeline, including graph-based backward attribution, structured failure diagnosis, evidence grounding, route-level credit assignment, and policy optimization.

**Diagnosis Quality** The diagnosis audit contains 98 matched records. We evaluate a diagnosis against the constraints that the official strict checker identifies as failed. The exact-signature Macro F1 is 73.81%, while the hierarchical-signature Macro F1 is 75.85%. Hierarchical matching treats a specific diagnosis as a valid refinement of its official parent constraint. For example, **count:punctuation:missing_interrobang** is treated as a refinement of **count:punctuation**. The hierarchical-signature Micro F1 is 74.14%, and the route-family Macro F1 is 82.14%.

Beyond signature matching, 77.55% of the audited diagnoses are grounded in an observed failed constraint. The optimizer retains 43 distinct diagnostic signatures, of which 88.37% are grounded in the corresponding execution evidence. These results indicate that NLPG generally attributes a failure to the correct constraint family, even when the generated signature is more specific than the identifier used by the official checker.

**Official Paired Repair Evaluation** To measure whether the generated feedback is actionable, we provide the NLPG-generated diagnosis instructions, the original request, and the original response to the same Qwen3-8B response model. The model is asked to produce a revised response, which is then evaluated using the official IFBench checkers. No approximate or model-based verifier is used for the reported repair results.

We distinguish between a failure-only analysis and a mixed paired analysis. The failure-only subset contains 91 examples for which NLPG produces a usable diagnosis. All 91 examples fail the strict

Table 9: Official paired repair evaluation on the mixed IFBench calibration subset. Repair success is computed over responses that fail the corresponding checker before repair. Regression is computed over responses that pass the corresponding checker before repair.

| Metric | Strict | Loose |
|---|---|---|
| Paired records | 143 | 143 |
| Pass before repair | 52/143 (36.36%) | 57/143 (39.86%) |
| Pass after repair | 59/143 (41.26%) | 64/143 (44.76%) |
| Repair success | 8/91 (8.79%) | 8/86 (9.30%) |
| Regression | 1/52 (1.92%) | 1/57 (1.75%) |
| Net gain | +7 | +7 |
| Failed constraints before → after | 97 → 89 | 91 → 84 |

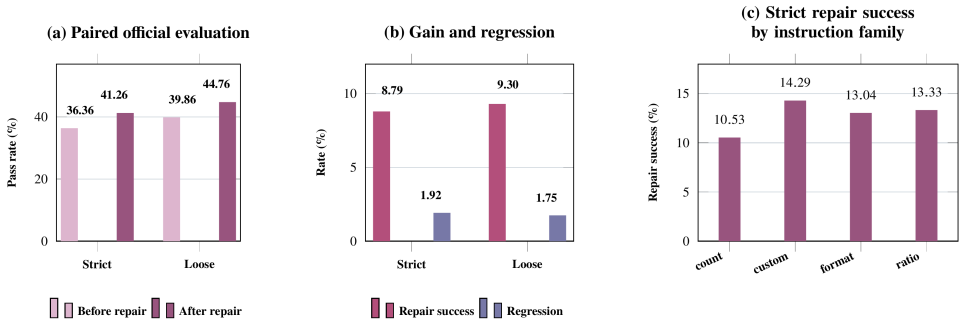



Figure 7: Feedback quality and repair evaluation on IFBench. (a) Paired official evaluation, showing pass rates before and after repair under strict and loose checkers. (b) Gain and regression, showing repair-success and regression rates. (c) Strict repair success by official instruction family.

checker before repair, while five already pass the loose checker. Therefore, the failure-only strict regression rate is undefined rather than zero.

For the regression-aware paired evaluation, we combine the 91 diagnosis-driven failures with 52 responses that pass the strict checker before repair. For the latter responses, the repair prompt contains only a preservation instruction that asks the model to retain all already satisfied constraints. Seven additional failed examples have no usable diagnosis and are therefore excluded from the repair analysis. The resulting mixed paired set contains 143 examples.

As shown in Table 9, the strict pass rate increases from 36.36% to 41.26%, while the loose pass rate increases from 39.86% to 44.76%. Under the strict checker, NLPG repairs 8 of the 91 initially failed responses and causes one of the 52 initially successful responses to fail. This produces a net gain of seven successful responses. The loose checker yields the same net gain, with eight successful repairs and one regression. The number of failed individual constraints also decreases from 97 to 89 under strict evaluation and from 91 to 84 under loose evaluation.

These results show that NLPG feedback yields a positive paired improvement, although the end-to-end repair rate remains modest. The low regression rate indicates that the preservation instruction usually retains already satisfied constraints, but the single observed regression also shows that repair is not risk-free. We therefore report repair gain and regression separately rather than presenting only the post-repair accuracy.

**Constraint-Family Analysis** Figure 7 groups the diagnosis-driven failures by the prefix of the official IFBench instruction identifier. An example can contribute to multiple families when multiple constraints fail simultaneously. For each family, the denominator is restricted to examples containing that family that fail the strict checker before repair.

The highest observed repair success is obtained for the **custom** family, with 1 of 7 examples repaired (14.29%). The **ratio** family achieves 2 of 15 repairs (13.33%), the **format** family achieves 3 of 23

repairs (13.04%), and the **count** family achieves 2 of 19 repairs (10.53%). No complete failure-to-pass repair is observed for the evaluated **repeat**, **sentence**, or **words** examples.

The family-level results highlight a distinction between diagnosis quality and response-level repair success. NLPG can identify and ground a failed constraint even when the response model cannot execute the corresponding correction reliably. This distinction is particularly important for word-level and sentence-level constraints, which often require a complete response to satisfy a global lexical or structural property. Consequently, diagnosis F1 and end-to-end repair success measure complementary aspects of feedback quality.

**Repair-Prompt Control** We additionally compare two repair-prompt variants. The **basic** variant contains the original request, the unsuccessful response, and the NLPG-generated diagnosis instructions. The **checklist** variant also exposes the official failed instruction identifiers and explicitly asks the model to verify every failed constraint while preserving constraints that the original response already satisfies.

On the failure-only subset, the basic prompt repairs 8 of 91 responses under strict evaluation, corresponding to 8.79%. The checklist prompt repairs 7 of 91 responses, corresponding to 7.69%. Under loose evaluation, the basic and checklist variants produce 13 and 12 passing responses after repair, respectively.

The checklist prompt therefore does not improve over the basic diagnosis prompt in this run. This result suggests that the main bottleneck is not merely the absence of an explicit instruction identifier. Instead, the remaining difficulty lies in executing multiple simultaneous output constraints with the response model. For this reason, we use diagnosis matching, evidence grounding, repair success, and regression as separate measurements of NLPG feedback quality.

### C.8 Evaluation Metrics

LoCoMo and LongMemEval are evaluated using benchmark question-answering accuracy. HaluMem reports Memory Integrity, Memory Accuracy, and downstream QA Accuracy under the definitions provided by the benchmark. HotpotQA uses lowercased, punctuation-normalized exact match with articles removed. IFBench uses the official strict instruction-following checker. HoVer uses its evidence-grounded verification score.

All percentages are reported in percentage points. When an auxiliary metric is computed, its denominator is fixed to the valid examples of the corresponding benchmark split. For the three-task transfer average, we compute the arithmetic mean of the HotpotQA, IFBench, and HoVer scores within the same model block.

### C.9 Policy-Memory Runtime

NLPG keeps the compound-agent program and underlying model parameters fixed while updating an external policy memory. A policy entry contains a short natural-language instruction, a utility score, and the identifiers of the examples that support the instruction. During execution, the relevant module-local policy entries are ranked by utility, deduplicated, and truncated to the configured top-$K$ budget before being inserted into the module context.

The policy version used for an evaluation pass is frozen throughout that pass. After evaluation, failure traces are grouped by failure signature and routed to the relevant task category or program module. The resulting update directions are aggregated across examples, and only updates satisfying the configured support and utility criteria are retained for the next policy version.

This separation ensures that reported task performance is measured with a fixed active policy and that policy construction does not alter the model weights, retrieval corpus, or program execution order during the same evaluation pass.

### C.10 Controlled Policy Intervention

We additionally conduct a paired intervention on one successful repair case. The question, backbone models, corpus, two-hop program, retrieval budget, and evaluation procedure are held fixed. Only a module-local specificity policy is added in the second run.

The failed trace has complete supporting-document recall but predicts the broader occupation "musician" instead of the gold answer "guitarist". We therefore diagnose an answer-overgeneralization error and construct a targeted instruction requiring the intersection of the two persons' explicitly stated occupations. The instruction is routed to the summary, second-hop query, and answer modules before rerunning the same example.

Table 10: Successful controlled policy intervention. Retrieval coverage is unchanged; only the module-local policy differs between the paired runs.

| Observable | Without policy | With policy |
|---|---|---|
| Question | Trey Anastasio and Glenn Bidmead have which mutual occupation? | |
| Supporting-document recall | 1.0 | 1.0 |
| Hop-2 query | What are the shared occupations of Trey Anastasio and Glenn Bidmead? | What occupation do Trey Anastasio and Glenn Bidmead share? |
| Predicted answer | musician | guitarist |
| Exact-match correctness | false | true |

The injected answer-module instruction is:

> Intersect the occupation attributes stated for both people and return the most specific shared answer span, rather than a broader umbrella term.

This intervention is a selected case study rather than an estimate of aggregate accuracy improvement. It provides controlled evidence that a module-local policy can repair answer specificity while leaving retrieval coverage unchanged.

## D Prompt Templates

This appendix describes the prompts used by NLPG. The fixed application prompts define the input and output interfaces of the program modules. The policy-update prompts operate on evaluated execution traces and generate module-local procedural corrections. The model parameters, program structure, retrieval interface, and evaluation procedure remain fixed during policy evolution.

### D.1 Application-Level Prompts

The application prompts specify the information consumed and produced by each module. They provide the execution interface without prescribing benchmark-specific failure corrections.

**Evidence Summarization**

You are the evidence summarization module.
Summarize the retrieved passages with respect to the current question and retain information that may be required by subsequent modules.
Return a structured response containing a concise summary and the salient entities, relations, or constraints found in the evidence.
**Question:** {question}
**Retrieved passages:**
**[1] {title_1}**
{text_1}
**[2] {title_2}**
{text_2}
...

**Query Generation**

You are the query-generation module.
Generate the next retrieval query from the task input and the preceding module output.
Return the query together with a concise rationale.
**Question:** {question}
**Previous module output:** {previous_output}

**Answer Synthesis**

You are the answer-synthesis module.
Use the retrieved evidence and intermediate outputs to produce the final answer. Return the answer and the evidence identifiers used by the program.
**Question:** {question}
**Retrieved evidence:**
**[1] {title_1}**
{text_1}
**[2] {title_2}**
{text_2}
...

The application prompts are unchanged across policy versions. NLPG modifies only the policy addendum appended to the corresponding module context.

### D.2 Module-Local Policy Injection

For module invocation $(i, j)$, NLPG retrieves a bounded set of instructions from the current policy version and appends them to the original module context. Let $c_{i,j}^{\text{base}}$ be the original context and let $\mathcal{P}_{t,i,j}$ be the selected policy instructions. The resulting context is

$$c_{i,j} = c_{i,j}^{\text{base}} \parallel \text{Format}\left(\mathcal{P}_{t,i,j}\right), \tag{12}$$

where $\parallel$ denotes text concatenation.

**Module-Local NLPG Guidance**

**NLPG procedural guidance for the {module_name} module:**
Use the following instructions to improve the current module operation while preserving the fixed input and output interface.
**1.** {instruction_1}
**2.** {instruction_2}
...

The instructions are ranked by their accumulated cross-execution support and retained within the configured policy budget. Route-specific instructions are associated with the module that produced the relevant execution evidence. Shared instructions may be included when the execution record supports a correction that applies across modules.

### D.3 LLM-Based Policy Reflection

After an execution is evaluated, the critic LLM analyzes its observable execution record. The record contains the task input, module dependencies, module inputs and outputs, retrieved evidence, generated queries, final response, and task-level evaluation feedback. The critic identifies an observable deviation, attributes it to a module present in the execution, and proposes a reusable behavioral correction.

**Natural-Language Policy Reflection**

Analyze the evaluated execution and its recorded module dependencies.
Identify the observable decision that contributed to the failure. Select the responsible module from the modules present in the execution graph. Explain the local problem and propose one concise procedural instruction that could help this module avoid the same failure on future task instances.
The instruction must describe reusable behavior. Do not copy task-specific answers, names, dates, labels, or reference information into the instruction.
Return a structured object containing:

- an execution-deviation descriptor;
- the responsible module;
- a concise critique;
- one reusable policy instruction;
- a confidence score; and
- references to the execution fields supporting the diagnosis.

**Task-level feedback:** {evaluation_feedback}
**Execution graph:** {execution_graph}
**Module records:** {module_records}

For invocation $(i, j)$, the resulting textual policy gradient is represented as

$$g_{i,j} = (a_{i,j}, s_{i,j}, u_{i,j}, \kappa_{i,j}) \,, \tag{13}$$

where $a_{i,j}$ is the receiving module selected from the observed execution graph, $s_{i,j}$ is the deviation descriptor generated by the critic, $u_{i,j}$ is the proposed natural-language correction, and $\kappa_{i,j}$ is the critic confidence. The descriptor and correction are generated from the execution evidence; they are not retrieved from a benchmark-specific correction table.

### D.4 Execution-Graph Diagnosis and Route Attribution

NLPG performs attribution over the module-dependency graph recorded during each evaluated execution. The evaluator provides task-level feedback, while the execution record provides the observable structure needed to determine where that feedback should be propagated.

The backward pass starts at the terminal module whose output contributes directly to the evaluated response. For an intermediate module, the critic collects the feedback messages returned by its immediate successors and examines them together with the module's input, output, and execution context. The critic may generate a local correction, forward a more specific feedback message to a recorded predecessor, or stop propagation when the available evidence does not justify further attribution.

This operation determines the receiving route from the observed execution graph rather than from a fixed list of benchmark-specific routes. The graph constrains where feedback may travel, while the critic determines which observable behavior should change and how the correction should be expressed. The resulting local corrections are subsequently grouped across task instances by the policy optimizer.

**Execution-Graph Backward Reflection**

Review the current module and the feedback received from its immediate successors.
Use the module input, output, and execution context to determine whether the current module contributed to the downstream error.
Return a local correction only when the execution evidence supports one. Forward feedback only to modules listed as immediate predecessors in the execution graph. Do not attribute an error to an unobserved module.
**Current module:** {module_record}
**Downstream feedback:** {downstream_feedback}
**Immediate predecessors:** {predecessors}

The backward pass is local to one execution. It should not be confused with the cross-execution aggregation described in Section 3.3, which combines corrections produced from different task instances.

### D.5 Policy Initialization and Update

To separate learned corrections from benchmark-specific behavioral knowledge, the main NLPG configuration initializes the route-local policy memory without failure-specific instructions:

$$M_0^a = \varnothing, \qquad a \in \mathcal{A}. \tag{14}$$

The fixed application prompts provide only the input and output contracts required by the program. They do not specify a particular bridge strategy, query-rewriting rule, retrieval correction, answer-selection heuristic, or failure-to-policy mapping.

After a group of task instances has been evaluated, the critic generates textual corrections from the corresponding execution records. The optimizer then groups compatible corrections, removes instance-specific content, applies the policy-safety checks, and ranks the remaining candidates by cross-execution support. The next policy version is

$$M_{t+1} = \mathrm{Update}\left(M_t, \{g_{i,j}\}_{i=1}^{N}\right), \tag{15}$$

where only corrections supported by the construction data and accepted by the policy-selection procedure are retained.

A policy instruction therefore enters the memory through the following sequence:

$$\text{evaluated execution} \rightarrow \text{LLM diagnosis} \rightarrow \text{module-local correction} \rightarrow \text{cross-execution consolidation} \rightarrow \text{policy validation}. \tag{16}$$

The final policy version is frozen before evaluation on the held-out test partition.

### D.6 Structured Module Outputs

The application modules return explicit intermediate fields so that later modules can consume observable outputs and so that NLPG can analyze the execution trace.

**Module Output Contracts**

Evidence summarization: summary and salient entities or relations.
Query generation: retrieval query and query rationale.
Answer synthesis: final answer and evidence identifiers.
Execution record: module inputs, module outputs, route identifiers, dependencies, and evaluation metadata.

These fields expose the program-level execution record without requiring access to hidden model reasoning. The policy-update process operates on these observable fields and does not modify the evaluator, scoring function, or program control flow.

### D.7 Prompt and Policy Budget

For each module invocation, NLPG retains at most $K$ policy instructions and limits the rendered policy addendum to the configured character budget. The same policy budget is used when evaluating a policy version and when constructing its successor.

The active policy version remains fixed throughout one evaluation pass. After the pass is complete, the critic analyzes the recorded executions and the optimizer constructs a candidate successor policy. The candidate is evaluated on the policy-selection partition before it can be promoted. The promoted policy is then frozen for the held-out test evaluation.

This separation ensures that a correction generated from one task instance cannot change the execution of another task instance in the same evaluation pass, and that final-test executions do not participate in policy construction or policy selection.