DIVIDE, DIAGNOSE, AND COMPOSE: OPTIMIZING AGENTIC PIPELINES AND HARNESSES

Anonymous authors
Paper under double-blind review

ABSTRACT

Building agentic pipelines that achieve high task performance while keeping cost and latency low requires jointly optimizing heterogeneous and coupled design choices, including prompts, topology, model selection, and tool policies. Manually searching this combinatorial design space is expensive and time-consuming, while existing automated optimizers typically operate within a predefined subset of this space, modify the pipeline monolithically, or provide little structured evidence for pipeline changes. To mitigate these challenges, we introduce the Decomposed Credit-Assignment Framework (DECAF), which adapts the design space to the given benchmark and pipeline, optimizes each dimension separately, and generates reasoning traces for every decision. Given a task specification, benchmark examples, and a seed pipeline, DECAF identifies relevant optimization dimensions and generates diverse experimental hypotheses for each. Independent coding-agent sessions then optimize each dimension under a restricted edit scope, jointly evaluating performance, inference cost, and latency while recording all hypotheses, edits, and outcomes of each experiment. A final synthesis session combines compatible findings across all dimensions and uses a component-level trajectory evaluator, which derives role-specific rubrics from each agent's instructions, to localize underperforming components and guide cross-dimension revisions. Across a diverse suite of agentic benchmarks, DECAF improves average task performance by 2.58 percentage points, while reducing inference cost by 4.30% and end-to-end latency by 42.45% compared to the best overall baseline.

1 INTRODUCTION

Large language models (LLMs) are increasingly deployed not as isolated models but as components of larger *agentic pipelines* and *harnesses* that combine multiple model calls with tools, memory, retrieval, verification, and control flow. The behavior of such pipelines depends on substantially more than just the choice of a base model. Developers must decide what instructions each component receives, how components exchange information, which models serve which roles, how much inference-time computation each call receives, what context is retained or discarded, and when intermediate outputs should be verified or revised. These design choices are coupled: a cheaper model may demand stronger scaffolding, and an added verifier increases both cost and latency. Therefore, manually tuning an agentic pipeline for competing objectives such as performance, cost, and latency is expensive and requires substantial task-specific expertise (Zhou et al., 2025; Hua et al., 2026). As a result, a growing body of work shows that autonomously optimizing all prompts, demonstrations, workflow structure, model selection, context, or agent roles can each produce meaningful gains in multi-LLM agentic pipelines (Wu et al., 2026; Saeidi et al., 2026; Zhang et al., 2025b; Agrawal et al., 2025; Opsahl-Ong et al., 2024).

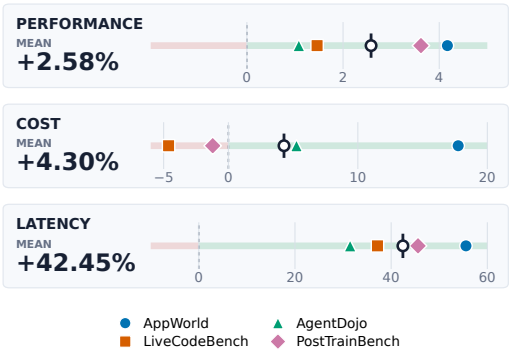

Figure 1: Improvements of DECAF over Meta-Harness (best baseline) across all benchmarks.

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.

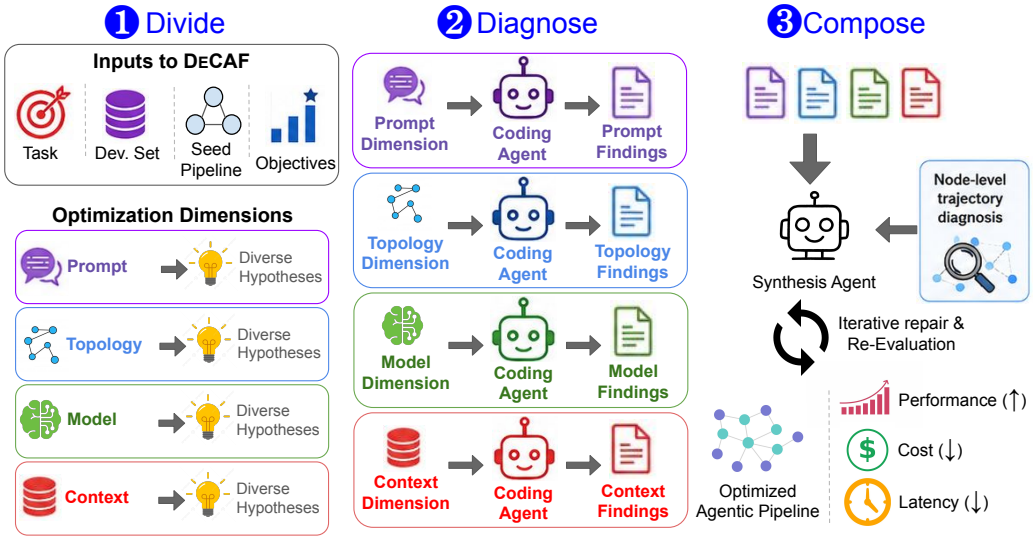

Figure 2: Given task details, development examples, seed agentic pipeline, and objectives, DECAF first identifies task-relevant optimization dimensions and generates diverse experimental hypotheses for each (*Divide*). Independent coding-agent sessions then optimize copies of the seed pipeline under dimension-specific edit constraints, producing Pareto candidates, execution trajectories, and structured findings (*Diagnose*). Finally, a synthesis optimizer combines compatible dimension-level findings and performs targeted cross-dimension revisions to produce a unified pipeline on the performance–cost–latency frontier with the help of a component or node-level diagnosis tool (*Compose*).

These techniques, however, exhibit one or more of the following limitations. First, several optimizers are constrained by design to a narrow set of dimensions. For example, GEPA (Agrawal et al., 2025) optimizes just the prompts of all agents, while AFlow (Zhang et al., 2025b) works by optimizing the topology of the pipeline. Optimizing multiple dimensions jointly has been shown to outperform optimizing just a single dimension (Zhou et al., 2025). Second, LLMs tend to produce solutions that lack diversity (Handa et al., 2026; Brown et al., 2024), a weakness that is amplified when they act as optimizers and must propose genuinely distinct candidate edits to a pipeline (Lee et al., 2026). Third, several optimizers proceed by trial and error without producing a reasoning trace, which makes the resulting changes difficult to understand, audit, or build upon (Zhang et al., 2025a). Finally, recent work argues that measuring each agent's individual contribution provides a more informative reward than a single final binary outcome score (Wang et al., 2026). Such fine-grained credit reveals which component is responsible for a failed or successful trajectory. To address these challenges, we introduce the Decomposed Credit-Assignment Framework (DECAF) for optimizing multi-agent pipelines and harnesses. Figure 2 illustrates its overall workflow.

DECAF begins by reading the task description, a few development examples, the seed agentic pipeline to be optimized, and the objectives (or metrics) it has to achieve. We treat each pipeline as a computational graph $G = (V, E)$, where each node $v \in V$ is an executable component of the pipeline, such as an LLM agent initialized with a particular role. Conditioned on the benchmark and seed pipeline information, DECAF starts by proposing the dimensions along which the pipeline can be optimized. Then for each dimension it iteratively generates diverse experiments (Handa et al., 2026) to improve exploration of ideas to test. Each dimension's experiments are executed by an independent coding-agent session that is restricted to edits within that dimension's scope. The findings of all dimension-specific sessions are recorded and passed to a final stage, in which a single coding agent optimizes the whole pipeline using the accumulated evidence. This synthesis agent is additionally given a `node_eval` tool, which reads trajectories generated by the current pipeline and produces a detailed score for each component, localizing the agents most in need of revision.

DECAF's main contribution is the way it organizes open-ended search for optimization. Rather than restricting optimization to a predefined design axis or tasking a single optimizer to modify the full pipeline monolithically, DECAF (i) adaptively decomposes the search space into task and pipeline-specific dimensions that are explored in independent coding-agent sessions, (ii) forces ex-

ploration within each dimension by generating diverse experiment hypothesis, and (iii) recomposes all dimension-level findings in a final synthesis stage using component-level trajectory feedback. Together, these mechanisms make pipeline optimization more structured, traceable, and amenable to localized diagnosis. Across held-out evaluations on a diverse suite of agentic benchmarks, DECAF improves the performance-cost-latency frontier over baselines. Compared to the strongest overall baseline, Meta-Harness (Lee et al., 2026), it improves task performance by an average of 2.58pp, while reducing per-task inference cost by 4.30% and end-to-end latency by 42.45% (Figure 1).

2 RELATED WORK

Automated Optimization of Agentic Pipelines and Harnesses. A large body of work automates parts of the design space. OPRO (Yang et al., 2024a) frames instruction search as optimization where the model is shown its previously proposed instructions alongside their measured scores and asked to write one that will score higher. MIPRO (Opsahl-Ong et al., 2024) jointly searches over the instructions and few-shot demonstrations of every module in a multi-stage program. TextGrad (Yuksekgonul et al., 2024) builds an analogy to backpropagation: an LLM writes a natural-language critique of each intermediate output with respect to the final objective, and the critique propagates backward to revise the prompt that produced it. GEPA (Agrawal et al., 2025) reflects in natural language on full execution traces to propose targeted edits and retains any candidate that is best on at least one training instance. MASPO (Wang et al., 2026) brings prompt optimization to multi-agent systems, scoring each agent's prompt not only by the quality of its own output but also by its effect on the agents downstream. ADAS (Hu et al., 2025) tasks a meta-agent with writing complete agent designs as executable programs, evaluating each and conditioning on the archive of past designs to invent the next. AFlow (Zhang et al., 2025b) likewise represents workflows as code, but explores modifications with Monte-Carlo tree search, letting execution feedback decide which branches deserve expansion. MASS (Zhou et al., 2025) demonstrates that prompts and topology are best optimized together, alternating between per-agent prompt tuning, topology search over a pruned space of influential configurations, and a final global prompt pass. MaAS (Zhang et al., 2025a) replaces the single best architecture with an "agentic supernet", a learned distribution over architectures from which a query-conditioned pipeline is sampled, so easy queries receive cheap systems and hard ones receive more compute. Meta-Harness (Lee et al., 2026) broadens the optimization space by asking a single coding agent to optimize the entire pipeline or harness. DECAF on the other hand splits and optimizes the entire space into independent dimensions and then synthesizes the findings.

Failure Localization and Credit Assignment. An end-to-end task reward indicates whether a complete pipeline succeeded, but often provides little information about where a multi-component execution failed. Zhang et al. (2025c) formalize automated failure attribution in LLM-based multi-agent systems as identifying both the agent and the decisive step associated with a failure. Recent work has introduced several forms of credit assignment for training and optimizing agents. Agent Lightning (Luo et al., 2025) decouples agent execution from reinforcement-learning infrastructure and includes a credit-assignment module that transforms arbitrary agent trajectories into training transitions. Li et al. (2026) decompose credit along temporal and structural axes, identifying weak interaction rounds and roles before refining role prompts and aggregation protocols through verbalized block-coordinate descent. MASPO (Wang et al., 2026) evaluates multi-agent prompts using local validity, lookahead potential for successor agents, and global alignment with the final system outcome. In our work, the `node_eval` tool used in the final stage of DECAF generates a fine-grained score for each agent in the pipeline. Because this evaluation is based primarily on observed trajectories, DECAF treats them as evidence for localizing *likely* underperforming components rather than as exact causal estimates of each node's contribution.

Coding Agents for Open-Ended System Optimization. Software-engineering agents can implement nontrivial changes to real repositories (Yang et al., 2024b; Zhang et al., 2025d), and self-improving systems such as the Darwin Gödel Machine (Zhang et al., 2026) use coding agents to rewrite their own agentic code. Meta-Harness (Lee et al., 2026) similarly formulates harness engineering as an outer-loop code-search problem. Its coding-agent proposer can inspect the harness source code, scores, and execution traces of prior candidates through a filesystem, and can then modify and evaluate the complete harness. Similarly, DECAF uses coding agents as optimization workers, but differs in how their search is organized. A hypothesis generator is used before the coding agent to boost exploration and a node-level diagnostic tool is used for better credit assignment.

3 DECAF

3.1 PROBLEM FORMULATION

Let $P_0 = (G, \Theta)$ denote a seed agentic pipeline, where $G = (V, E)$ is its computational graph and Θ contains configurable choices such as prompts, model assignments, inference parameters, context policies, tools, and control flow. Executing a pipeline P on input x produces an output $\hat{y}$ and a trajectory τ containing intermediate node inputs, outputs, tool calls, and system state. For a development set $\mathcal{D}_{\text{dev}} = \{x_i\}_{i=1}^{N}$, we associate each pipeline with the objective vector

$$\mathbf{J}(P) = \frac{1}{N} \sum_{i=1}^{N} \Big(q(P, x_i),\ -c(P, x_i),\ -\ell(P, x_i) \Big), \tag{1}$$

where q, c, and ℓ measure task performance, inference cost, and end-to-end latency, respectively. A candidate dominates another if it achieves no lower task performance, no greater cost or latency, and strictly improves at least one objective. While objectives can be folded into a single scalar by weighted combination, e.g., $0.8 \times q(P, x_i) - 0.1 \times c(P, x_i) - 0.1 \times \ell(P, x_i)$, this scalarization commits to weights before the attainable trade-offs are known, yields one operating point per run, and has well-known failure modes (e.g., non-convex regions of the frontier are unreachable by any linear weighting) (Deb et al., 2002). Therefore, rather than collapsing these objectives into a fixed weighted scalar, DECAF maintains non-dominated candidates, preserving operating points with different deployment trade-offs. The held-out test set is not exposed during any part of DECAF.

3.2 DIVIDE: TASK-ADAPTIVE DIMENSIONS AND EXPERIMENTS

DECAF first inspects the task description, a small subset of development examples, the implementation of the seed pipeline, and the objectives as defined in §3.1. Conditioned on few-shot decomposition examples, a dimension planner proposes task-relevant optimization dimensions $D_1, \ldots, D_K$. Each dimension is represented as $D_k = (\mathcal{M}_k, \mathcal{C}_k, \rho_k)$, where $\mathcal{M}_k$ specifies the editable pipeline components, $\mathcal{C}_k$ specifies scope and validity constraints, and ρ_k records why the dimension is relevant to the task. Candidate dimensions include prompt optimization, workflow topology, model and inference configuration, context management, tool policy, memory, and agent skills. The proposed dimensions are benchmark-and-pipeline-dependent rather than fixed across all tasks. For each dimension D_k, an experiment generator proposes a set of hypotheses $\mathcal{H}_k = \{h_{k,1}, \ldots, h_{k,n_k}\}$. Each hypothesis describes a distinct configuration to investigate rather than a complete implementation. For example, a topology dimension may consider adding a verifier, introducing a planner–executor structure, or adding a targeted repair loop; a model-configuration dimension may consider role-specific model assignments or different reasoning effort.

To maximize exploration of each dimension, each hypothesis is generated iteratively. At each iteration, the generator is conditioned on previously proposed hypotheses and instructed to produce a distinct experimental direction. Inspired by GuidedSampling (Handa et al., 2026), this iterative procedure reduces evaluation budget spent on superficial variations of the same idea. Each hypothesis records its motivation, affected components, and expected effects on performance, cost, and latency.

3.3 DIAGNOSE: CONSTRAINED DIMENSION-LEVEL OPTIMIZATION

For every dimension D_k, DECAF creates a copy of the seed pipeline and launches a fresh coding-agent session for optimization. The session receives the task specification, development data, dimension definition, and hypotheses $\mathcal{H}_k$. It has a primary editable scope $\mathcal{M}_k$ and is instructed to focus its search on dimension D_k. However, supporting changes are allowed. For example, when modifying topology, new prompts need to be written for the agents. The 'topology' still remains the primary optimization dimension, while 'prompt' becomes the secondary or supportive dimension. The coding agent iteratively selects or adapts a hypothesis, implements the corresponding changes, and evaluates the candidate. Each dimension maintains a non-dominated archive

$$\mathcal{A}_k \leftarrow \mathrm{ND}\left(\mathcal{A}_k \cup \{P_{k,j}\}\right), \tag{2}$$

where ND removes candidates dominated by another member of the archive. A candidate need not strictly dominate the seed to be retained. For example, a performance gain at a modest cost increase may represent a useful operating point on the observed frontier. Every attempted experiment is also

recorded in a `FINDINGS.md` artifact. The record includes the motivating hypothesis, affected components, implemented change, measured objectives, whether the candidate was retained, observed regressions, and conclusions for later optimization. Failed experiments are preserved so that the final optimizer can avoid repeating ineffective changes and can reason about benchmark-specific trade-offs. Because all dimension-level sessions begin from the same seed and vary only their assigned design dimension, their outcomes provide controlled, conditional evidence about which classes of interventions are effective around P_0. To reduce the wall-clock optimization time, the sessions run in parallel. Each dimension returns its archive and findings, $(\mathcal{A}_k, \mathcal{F}_k)$, to the final stage.

3.4 COMPOSE: NODE-GUIDED EVIDENCE SYNTHESIS

Independently successful changes need not remain successful when combined. For example, adding a verifier may require corresponding prompt changes, while context compression may interact with a new model assignment. DECAF therefore launches a final coding-agent session that receives the seed pipeline, all dimension-level archives and findings, and the task objectives. Unlike the dimension-level optimizers, this final synthesis agent may modify the complete pipeline. To help the synthesis agent understand why a pipeline succeeds or fails, we provide a diagnostic tool called `node_eval`. Instead of looking only at the final task score, `node_eval` examines each node across multiple development trajectories. It evaluates three main aspects of a node: *correctness*, whether the node makes a sound contribution toward the user's goal; *compliance*, whether it follows the instructions in its own system prompt; and *downstream utilization*, whether its output is passed to and used by later components. Evaluating each node across multiple development trajectories reduces sensitivity to isolated failures and helps reveal recurring behavior (Li et al., 2026).

For node v and evaluation criterion r, let $s_{v,r}^{(\tau)}$ denote the score assigned on development trajectory τ. Some criteria may not apply to every trajectory, or the available trace may not contain enough evidence to score them. We therefore define $\mathcal{T}_{v,r}^{\text{scored}}$ as the set of trajectories for which criterion r receives a numeric score, and summarize the node's behavior as

$$\bar{s}_{v,r} = \frac{1}{|\mathcal{T}_{v,r}^{\text{scored}}|} \sum_{\tau \in \mathcal{T}_{v,r}^{\text{scored}}} s_{v,r}^{(\tau)}. \tag{3}$$

In addition to these scores, `node_eval` returns evidence and natural-language descriptions of recurring problems. By examining the node output together with the information delivered to downstream components and the final pipeline result, it can distinguish likely problems in the source node, its consumer, or the handoff between them. These diagnoses are based on observed trajectories, so we use them to localize likely failures rather than to claim exact causal responsibility.

The synthesis agent uses these diagnostics together with the successful and unsuccessful findings from the dimension-level searches. It tests combinations of promising changes and re-evaluates every composed pipeline end-to-end on D_{dev}. When a combination regresses, `node_eval` can help identify where the interaction failed and guide a targeted repair, such as changing a prompt or adjusting the information passed between stages. Importantly, `node_eval` is only a diagnostic tool: pipeline changes are still accepted or rejected using the task-performance, cost, and latency objectives. The synthesis stage maintains a final non-dominated archive using the same update rule as Eq. 2. A final pipeline and harness is selected via the optimizer LLM from this archive using development-set evidence and is then evaluated on the held-out test set.

4 EXPERIMENT SETUP

Benchmarks and Data Splits. We evaluate DECAF on four benchmarks: AppWorld (Trivedi et al., 2024), LiveCodeBench (Jain et al., 2025), AgentDojo (Debenedetti et al., 2024), and PostTrainBench v1.1 (Rank et al., 2026). For AppWorld, we optimize on the official `dev` split and evaluate on `test_normal`. For LiveCodeBench, we use the code-generation setting and create a split from release_v6, dropping problems before 2024-10-01. We then use 20% (51 instances) of the data as development set, while 80% (206 instances) is used as the held-out test set. We also remove the "Easy" split, since it was saturated. For AgentDojo, we create development and test splits at the user-task level. We use 20 user tasks (60 episodes) for development and 77 user tasks (231 episodes) for testing, with each task evaluated once benignly and under two attacks. For PostTrainBench, we construct development and test settings across both target base models and benchmark.

$$\mathcal{M}_{\text{dev}} = \{\text{Qwen3-1.7B}, \text{SmolLM3-3B}\}, \quad \mathcal{B}_{\text{dev}} = \{\text{GSM8K}, \text{GPQA Main}, \text{AIME 2025}\},$$

$$\mathcal{M}_{\text{test}} = \{\text{Qwen3-4B}, \text{Gemma-3-4B}\}, \quad \mathcal{B}_{\text{test}} = \{\text{HumanEval}, \text{AIME 2026}, \text{BFCL}\}$$

Thus, the final evaluation tests whether an optimized post-training pipeline transfers to both unseen base models and unseen task domains. For BFCL, we use the full BFCL-V3 evaluation suite (Patil et al., 2025) rather than the smaller subset distributed with PostTrainBench v1.1.

Seed pipelines and Models. Each optimization begins from a fixed seed pipeline. For AppWorld, we use a single ReAct coding harness, backed by Claude-Sonnet-4.6 at `medium` effort, with other Claude models as choices for optimization. For LiveCodeBench, we use a sequential generator–critic–refiner pipeline in which all three nodes are initialized with GPT-5.4-nano, and GPT-5.4 family of models given for optimization. For AgentDojo, we use the seed harness with a single-stage ReAct agent using Qwen-3.5-397b-a17b-FP8 model, and Gemma (Team et al., 2026) and Qwen (Team, 2026) family of models are available as optimization choices. For PostTrainBench, a single Claude Code post-training agent backed by Claude-Sonnet-5 at `high` effort is used as the seed. Opus and Haiku variants are given as choices for optimization. DECAF uses Claude Code as the coding agent and uses `/goal` mechanism[1] to maintain the optimization objective. Every optimizer is run twice and the final table reports the average and standard error across both runs.

Baselines. We compare DECAF against the unoptimized seed pipeline and three automated optimizers. We additionally evaluate the unoptimized seed pipeline using backbone models from different capability tiers. These baselines measure whether the gains from optimization can be reproduced by simply replacing the underlying model with a more capable and expensive alternative. GEPA (Agrawal et al., 2025) performs reflective evolutionary optimization over the prompts of all LLM nodes. We initialize every GEPA optimization with the `heavy` budget [2] for a thorough optimization. MASPO (Wang et al., 2026) jointly optimizes node prompts using local validity, downstream usefulness, and global task feedback. Meta-Harness (Lee et al., 2026) gives a coding-agent proposer access to the complete history of previous harnesses, their scores, source code, and raw execution traces through a filesystem, and lets the agent decide what to inspect and what to modify. We initialize Meta-Harness with $N = \#Dimensions$ and $k = \#Experiments/Dimension$ to keep the experiment budget similar to DECAF. All optimizers use Claude-Sonnet-5 at `high` effort. For methods requiring a scalar candidate score, we normalize all objectives relative to the seed pipeline:

$$U(P) = w_Q \cdot Q(P) + w_C \frac{C(P_0) - C(P)}{C(P_0)} + w_L \frac{L(P_0) - L(P)}{L(P_0)}, \tag{4}$$

where $(w_Q, w_C, w_L) = (0.8, 0.1, 0.1)$ is fixed before test evaluation and shared across benchmarks. DECAF and Meta-Harness receive the three objectives separately and maintain non-dominated candidates rather than optimizing this scalarization. The optimizing LLM is asked to return a single pipeline that they believe is best given the three criteria.

5 RESULTS

5.1 END-TO-END PERFORMANCE–COST–LATENCY TRADE-OFFS

Table 1 compares the final pipelines produced by each optimizer. Across the four benchmarks, DECAF improves task performance over Meta-Harness by an average of 2.58 percentage points while reducing inference cost by 4.30% and latency by 42.45%. The largest gains occur on AppWorld, where DECAF improves TGC by $4.17pp$ while reducing cost by 17.8% and latency by 55.6%. On LiveCodeBench and AgentDojo, it improves performance by $1.46pp$ and $1.08pp$ respectively, with substantial latency reductions. LiveCodeBench incurs a small increase in inference cost ($0.0003, 4.6%). On PostTrainBench, DECAF improves the task score over Meta-Harness while roughly halving deployment latency, although optimization is substantially more expensive because GPU constraints prevent parallel dimension-level search. Stronger unoptimized backbones can exceed DECAF in raw task performance on some benchmarks, but generally require substantially greater deployment cost. MASPO on AppWorld and GEPA and MASPO on PostTrainBench regress during their optimization runs and hence return the seed pipeline as is. Detailed benchmark-specific optimizations are provided in Appendix A.

Optimization dynamics. Figure 3 plots each optimizer against cumulative expenditure. DECAF reaches high validation TGC early in the search while exploring substantially lower-latency configurations. Its trajectory is non-monotonic because independent dimension-level searches explore

[1] `https://code.claude.com/docs/en/goal`

[2] `https://dspy.ai/api/optimizers/GEPA/overview/`

Table 1: Task scores (% ± standard-error), cost-per-instance, latency-per-instance, one-time cost of optimization, and wall-clock time (hh:mm) of all optimizers. Each row is an average of two runs.

		Deployment / instance		Optimization	
Method	**Task score ↑**	**Cost ($) ↓**	**Latency (s) ↓**	**Cost ($) ↓**	**Time ↓**
AppWorld (TGC)					
Seed pipeline (Sonnet-4.6-Medium)	$78.17_{\pm 1.21}$	0.2242	55.1	–	–
Stronger backbone (Opus-4.8-High)	$87.20_{\pm 0.89}$	0.4273	114.3	–	–
GEPA	$74.40_{\pm 1.19}$	0.2901	58.8	279.04	14:57
MASPO	$78.17_{\pm 1.21}$	0.2242	55.1	103.57	02:06
Meta-Harness	$80.06_{\pm 0.89}$	0.2237	73.8	166.91	07:58
DECAF (Ours)	$84.23_{\pm 0.89}$	0.1840	32.8	191.86	04:28
LiveCodeBench (pass@1)					
Seed pipeline (GPT-5.4-nano)	$47.33_{\pm 1.13}$	0.0051	29.3	–	–
Stronger backbone (GPT-5.4)	$55.34_{\pm 0.56}$	0.0236	12.5	–	–
GEPA	$48.30_{\pm 1.70}$	0.0055	37.2	24.60	09:31
MASPO	$48.30_{\pm 0.24}$	0.0052	26.4	74.21	02:48
Meta-Harness	$51.70_{\pm 1.21}$	0.0065	44.4	57.22	07:04
DECAF (Ours)	$53.16_{\pm 0.73}$	0.0068	27.9	81.88	04:37
AgentDojo					
Seed pipeline (Qwen-3.5-397b-a17b-FP8)	$66.71_{\pm 0.70}$	0.0163	16.45	–	–
Stronger backbone (Gemma-4-31B-it)	$80.93_{\pm 0.35}$	0.0060	49.33	–	–
GEPA	$83.76_{\pm 0.51}$	0.0137	24.23	32.92	04:17
MASPO	$87.76_{\pm 0.54}$	0.0147	36.66	21.05	01:14
Meta-Harness	$86.69_{\pm 0.24}$	0.0133	37.40	28.74	07:11
DECAF (Ours)	$87.77_{\pm 0.45}$	0.0126	25.63	26.48	04:43
PostTrainBench			*(hh:mm)*		
Seed pipeline (Sonnet-5-High)	$33.56_{\pm 1.55}$	10.07	08:01	–	–
Stronger backbone (Opus-4.8-Max)	$40.54_{\pm 0.93}$	20.20	08:17	–	–
GEPA	$33.56_{\pm 1.55}$	10.07	08:01	154.26	20:41
MASPO	$33.56_{\pm 1.55}$	10.07	08:01	86.10	06:21
Meta-Harness	$31.97_{\pm 2.77}$	9.90	07:04	145.63	42:46
DECAF (Ours)	$35.59_{\pm 1.00}$	10.02	03:51	177.25	43:52

different regions of the design space. The final synthesis evaluations combine these findings and produce the selected pipeline, which slightly exceeds Meta-Harness in validation TGC while substantially reducing deployment cost and latency. Even though both Meta-Harness and DECAF run 9 experiments, DECAF is able to exploit the solution space more effectively. Meta-Harness TGC and cost start saturating at just 22.2% of its budget while the latency drops and saturates at 77.8%.

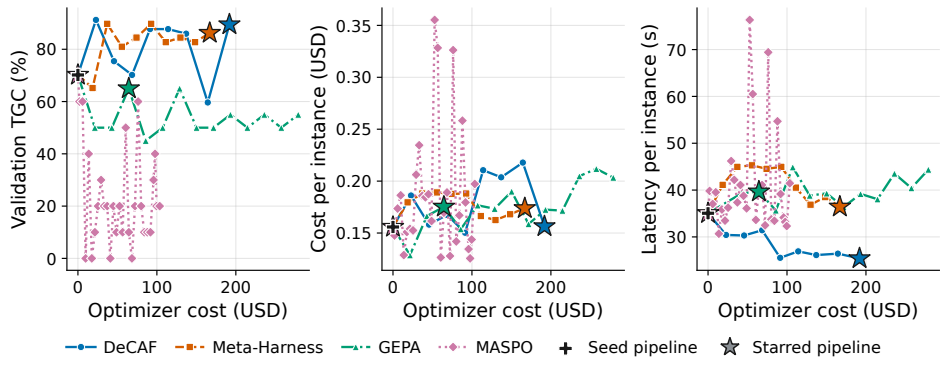

Figure 3: AppWorld optimization trajectories as a function of cumulative optimizer cost. Each point corresponds to an evaluated validation candidate; the panels report TGC, per-instance inference cost, and latency, respectively, while stars denote the pipeline ultimately selected by each optimizer. Trajectories need not be monotonic as they show individual evaluated candidates not the best so far.

5.2 Optimization Overhead of DECAF

Table 1 reports the one-time cost and wall-clock time required to run each optimizer. Since DECAF decomposes the search space into independent dimensions, coding agents for each dimension can run in parallel. This reduces the wall-clock time of DECAF significantly compared to Meta-Harness, where experiments run sequentially. The only exception being PostTrainBench, where we ran each dimension sequentially due to the bottleneck on the GPU resources. This makes DECAF 27.6% faster on average (37.65% faster if we remove PostTrainBench). However, this decomposition and better exploration comes at a cost, it increases the total cost of optimization. On average, DECAF is 17.98% more expensive than Meta-Harness. There are two major contributions of this increased cost. First, the optimizer is forced to explore performance using the stronger (and generally more expensive) models. This increases the cost of evaluation of the pipeline on the validation set. Second, the `node_eval` tool makes up roughly 6% of the optimization cost of DECAF. This tool takes in multiple trajectories of the agent which increases the input token cost. Due to the heavy cost of this tool, it is only given to the final synthesizer coding agent, not the dimension-level coding agents.

5.3 Ablation Study

To identify the contribution of each part of DECAF, we perform ablations on AppWorld and LiveCodeBench benchmarks. We consider the following variants: **No Dimensions** removes task-adaptive decomposition while preserving the number of coding-agent optimizers. A global experiment generator proposes the same total number of hypotheses as full DECAF without conditioning on a particular dimension. The hypotheses are divided evenly among four independent coding-agent sessions, each tasked with optimizing the seed pipeline and is given unrestricted edit access. Their resulting pipelines and findings are then provided to the synthesis agent. **Single Shared Session** retains the dimensions and experiment proposals produced by full DECAF, but replaces the independent dimension-specific coding-agent sessions with a single coding-agent session. The single agent optimizes the seed-pipeline for all dimensions. This tests whether independent optimization contexts reduce interference when searching heterogeneous parts of the design space. **No Experiment Proposals** retains the task-adaptive dimensions and independent coding-agent sessions but removes the diverse experiment hypothesis. Each coding agent instead is tasked with generating its own experiments during optimizing its assigned dimension. **No `node_eval`** uses the same final synthesis coding agent, but removes access to the component-level trajectory diagnostics. The synthesis agent can still inspect ordinary execution traces and evaluation results by itself. **No Synthesis** retains the complete Divide and Diagnose stages of DECAF but performs no cross-dimensional optimization. The final pipeline is constructed simply as the combination of the dimension-specific optimized pipeline. This measures how much improvement arises from combining and repairing dimension-level findings. **Dimension Best** evaluates the pipeline by all dimensions on the validation set and returns the best one skipping the entire synthesis stage. It differs from No Synthesis, as it doesn't combine the findings across dimensions.

Table 2 reports the results of all ablations. Across both benchmarks, a big drop is observed on the *No Dimensions* variant, where we remove the dimension proposer, and directly ask the parallel coding agents to optimize the pipeline. Upon inspecting the trajectories of the optimizers we found that roughly 41.67% of the budget was spent on context engineering, followed by spending 16.67% of the budget in Harness Optimization. This left very little budget to effectively explore other dimensions like Prompt Optimization, Topology Variations, and Different Agent Configurations. Similarly in the *No Experiment Proposal* variant, both benchmarks regressed significantly. Upon further inspection, we found that the optimizer either stops after finding a single solution better than the seed pipeline, or it tries to work out a single hypothesis without exploring alternatives. Both these variants highlight the limitations of existing models to effectively explore.

The rest of the variations help answer the question: "Are the dimensions truly independent of each other and whether they interact positively or not?". Both the *No* `node_eval` and *No Synthesis* variations show little degradation in AppWorld but degrades by up to 8.35pp on LiveCodeBench. This highlights that optimizations made by individual dimensions within AppWorld can simply be added up without finetuning the pipeline, whereas in LiveCodeBench these optimizations are destructive to each other. This highlights why the final synthesis stage of DECAF is really important, especially the `node_eval` tool. The *Single Shared Session* variation further highlights this distinction. On AppWorld, where dimensions interact positively with each other, having a single coding agent de-

Table 2: Ablation study on the AppWorld and LiveCodeBench benchmarks. Δ denotes the absolute change in the benchmark-specific task performance metric relative to the full DECAF method. Per-instance columns report test-time deployment metrics, while optimizer columns report one-time optimization overhead.

	Task performance		Per instance		Optimizer overhead	
Method	**Score** ↑	Δ (pp)	**Cost** ($) ↓	**Latency** (s) ↓	**Cost** ($) ↓	**Wall-clock** (h:mm:ss) ↓
AppWorld	*TGC (%)*					
DECAF (full)	$84.23_{\pm 0.89}$	0.00	0.1840	32.81	191.86	04:27:57
No Dimensions	$79.17_{\pm 1.19}$	–5.06	0.2273	56.63	153.59	02:59:56
Single Shared Session	$80.67_{\pm 0.84}$	–3.56	0.2349	55.92	174.80	07:01:55
No Experiment Proposals	$74.40_{\pm 2.98}$	–9.83	0.3838	78.47	134.23	06:12:08
No `node_eval`	$83.93_{\pm 0.60}$	–0.30	0.1971	41.61	181.42	05:15:36
No Synthesis	$84.18_{\pm 0.72}$	–0.05	0.1840	32.82	178.05	04:07:23
Dimension Best	$87.20_{\pm 0.89}$	+2.97	0.4273	114.34	175.48	04:00:57
LiveCodeBench	*Pass@1 (%)*					
DECAF (full)	$53.16_{\pm 0.73}$	0.00	0.0068	27.9	81.88	04:36:47
No Dimensions	$49.93_{\pm 1.12}$	–3.23	0.0065	34.5	80.59	04:10:01
Single Shared Session	$52.89_{\pm 0.93}$	–0.27	0.0068	28.4	83.01	07:49:36
No Experiment Proposals	$49.52_{\pm 0.87}$	–3.64	0.0069	30.2	74.61	04:40:29
No `node_eval`	$50.10_{\pm 0.68}$	–3.06	0.0070	29.0	76.02	04:32:57
No Synthesis	$44.81_{\pm 0.84}$	–8.35	0.0072	31.1	72.72	04:02:11
Dimension Best	$51.37_{\pm 0.90}$	–1.79	0.0068	29.7	71.15	03:55:48

grades the performance. On LiveCodeBench however, the performance drop is insignificant since all the dimension optimizations need to be tested together. Finally, *Dimension Best* shows why different dimensions are important. On AppWorld, a simple model scaling gives a better result (+2.97pp) but with a much higher cost (+132.23%) and latency (+248.49%). On LiveCodeBench, context engineered pipeline highlighting potential errors was selected. Since the dimension was optimized independently there weren't any conflicts from other dimensions leading only to slight decrease in performance, while cost remained the same, and latency increased slightly. Since it's impossible to know beforehand whether the dimensions will interact positively or negatively, we believe the three stages of DECAF (Divide, Diagnose, and Compose) helps recover performance in both scenarios.

Node-level diagnosis. The ablations establish that `node_eval` improves final synthesis, but do not determine whether its component-level diagnoses accurately identify the source of a failure. We therefore sample 100 execution trajectories and inject a known failure into a randomly selected component of each trajectory. We then run `node_eval` on the corrupted trajectories and measure whether it identifies the injected component as the most likely source of failure or not. We measure exact top-1 localization against a random-component baseline of 31.8%. `node_eval` correctly localizes 77% of injected failures. Manual inspection reveals two error cases: (1) When the immediate downstream component produces a lower-quality response due to the corrupted component, and is flagged as the source of failure and (2) over-reliance on the model-generated compliance rubric measuring whether the component follows its own system prompt. In such cases, `node_eval` identifies the weakest observed output rather than the causal origin of the failure. Full details in Appendix B.

6 CONCLUSION

We presented DECAF, a framework that turns agentic-pipeline or harness optimization from a monolithic search over a fixed subspace into an adaptive, decomposed, evidence-generating process. It starts by dividing the search space into benchmark and pipeline specific dimensions and then proposes diverse experiments for each dimension. It then optimizes each dimension with a coding-agent and finally composes all dimension-level findings into a final pipeline using a node-level trajectory analysis tool. This decomposed search allows DECAF to achieve better performance–cost–latency frontier on four agentic benchmarks.

AI DISCLOSURE SECTION

In this work, we used generative AI tools for paper writing and coding our experiments. Our method and baselines themselves require LLMs as part of their methods. We have **not** used generative AI tools for interpreting results, refining hypothesis, or creating scientific images. We have reviewed all AI-assisted work. We thoroughly verify all AI generated artifacts. We take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

To ensure the reproducibility of our results, we release the algorithm of DECAF in Appendix C and prompts in the Appendix D. We also note that LLMs are inherently probabilistic in nature, and some results may vary upon each run.

ETHICS STATEMENT

In accordance with ICLR policy, we disclose that AI assistants, specifically Grammarly for grammar correction and ChatGPT for sentence restructuring and paraphrasing, were utilized during the preparation of this manuscript. The authors have reviewed, edited, and take full responsibility for all final content presented in this paper.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2): 182–197, 2002.

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for llm agents. *Advances in neural information processing systems*, 37:82895–82920, 2024.

Divij Handa, Mihir Parmar, Aswin RRV, Md Nayem Uddin, Hamid Palangi, and Chitta Baral. Guidedsampling: Steering llms towards diverse candidate solutions at inference-time. In *International Conference on Learning Representations*, volume 2026, pp. 145052–145077, 2026.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Wenyue Hua, Sripad Karne, Qian Xie, Armaan Agrawal, Nikos Pagonas, Kostis Kaffes, and Tianyi Peng. Agentopt v0. 1 technical report: Client-side optimization for llm-based agent. *arXiv preprint arXiv:2604.06296*, 2026.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Wenwu Li, Yuran Song, Mingze Zhao, Bo Jin, and Wenhao Li. Unifying temporal and structural credit assignment in llm-based multi-agent prompt optimization. *arXiv preprint arXiv:2605.30227*, 2026.

Xufang Luo, Yuge Zhang, Zhiyuan He, Zilong Wang, Siyun Zhao, Dongsheng Li, Luna K Qiu, and Yuqing Yang. Agent lightning: Train any ai agents with reinforcement learning. *arXiv preprint arXiv:2508.03680*, 2025.

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.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Ben Rank, Hardik Bhatnagar, Ameya Prabhu, Shira Eisenberg, Karina Nguyen, Matthias Bethge, and Maksym Andriushchenko. Posttrainbench: Can llm agents automate llm post-training? *arXiv preprint arXiv:2603.08640*, 2026.

Amir Saeidi, Venkatesh Mishra, Souradeep Mukhopadhyay, Gaowen Liu, Ali Payani, Jayanth Srinivasa, and Chitta Baral. Fama: Failure-aware meta-agentic framework for open-source llms in interactive tool use environments. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 34346–34367, 2026.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

Qwen Team. Qwen3. 5-omni technical report. *arXiv preprint arXiv:2604.15804*, 2026.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024.

Zhexuan Wang, Xuebo Liu, Li Wang, Zifei Shan, Yutong Wang, Zhenxi Song, and Min Zhang. Maspo: Joint prompt optimization for llm-based multi-agent systems. *arXiv preprint arXiv:2605.06623*, 2026.

Shirley Wu, Parth Sarthi, Shiyu Zhao, Aaron Lee, Herumb Shandilya, Adrian Mladenic Grobelnik, Nurendra Choudhary, Edward Huang, Karthik Subbian, Linjun Zhang, et al. Optimas: Optimizing compound ai systems with globally aligned local rewards. In *International Conference on Learning Representations*, volume 2026, pp. 149685–149706, 2026.

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*, volume 2024, pp. 12028–12068, 2024a.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024b.

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

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*, 2025a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025b.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, et al. Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems. *arXiv preprint arXiv:2505.00212*, 2025c.

Zehua Zhang, Ati Priya Bajaj, Divij Handa, Siyu Liu, Arvind S Raj, Hongkai Chen, Hulin Wang, Yibo Liu, Zion Leonahenahe Basque, Souradip Nath, et al. Buildbench: Benchmarking llm agents on compiling real-world open-source software. *arXiv preprint arXiv:2509.25248*, 2025d.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö Arık. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv preprint arXiv:2502.02533*, 2025.

A DETAILED RESULTS

In this section, we describe the optimization steps done by DECAF for all benchmarks along with the intermediate findings from each dimension. This discussion sheds more light to the main results present in Table 1.

A.1 APPWORLD

Table 3 shows the detailed results for the AppWorld benchmark. DECAF optimized the seed pipeline of Sonnet-4.6-ReAct agent across four dimensions: prompts, agent configuration, pipeline topology, and context handling. Each dimension was given 6 experiments by the experiment generator.

The dimension prompt was tasked with optimizing the system prompt of the pipeline. While optimizing this dimension, the optimizer tried re-emphasizing reading API docs, few-shot examples, structured reasoning, changing answer format, adding common mistakes, and explicit instructions to focus on identity linking for APIs. The optimizer tried the first four out of these six suggested experiments and then conducted a final experiment based on the findings. It reasoned that best gains come when explicit API endpoints are listed in the prompt which reduced confusion. For example, the model often confused the API endpoint `show_liked_songs` and `show_song_library`. Along with this, it gave a few-shot example and final answer formatting.

The agent configuration dimension was tasked with optimizing the model and execution budgets. The optimizer tested increasing and decreasing the observation limit, changing the reasoning effort, extending the turn budget, and replacing backbone model (Sonnet-4.6) with Opus-4.8. Its main finding was that allowing longer observations reduced failures caused by incomplete API responses. Increasing reasoning effort or switching to Opus-4.8 did not resolve the failures observed under the original observation limit, while extending the turn budget provided little benefit because the inspected tasks finished before reaching it. Lower reasoning effort and shorter observations introduced additional interaction costs without improving task completion. The optimizer therefore retained the larger observation limit while preserving the original model and reasoning settings.

The pipeline topology dimension was tasked with changing the pipeline by introducing additional stages around the ReAct agent. The optimizer tried documentation priming, error recovery, post-completion verification, task planning, and answer grounding. Documentation priming was retained: a deterministic stage retrieved app and API descriptions before the ReAct loop and supplied them as reference material, reducing the agent's need to discover this information through separate reasoning turns. Error recovery and answer grounding added overhead without improving completion. Verification and planning initially appeared promising on the small validation probe but regressed on the full validation set. The optimizer skipped the proposed history-compression stage because the inspected trajectories did not establish a need for it.

The context-handling dimension was tasked with optimizing how observations and interaction history were presented to the agent. The optimizer tried explicit truncation warnings, compression of older history, a persistent cache for retrieved API documentation, and dynamically expanding the observation limit. It retained the truncation warning, which explicitly identified incomplete output and instructed the agent to paginate or narrow its query before computing totals. Heuristic history compression lost details needed in subsequent turns, while the documentation-cache variants failed to prevent repeated lookups and increased interaction costs. Dynamic observation expansion appeared neutral on the small probe but regressed on full validation. The optimizer skipped removing successful code blocks and adding a running state summary because these proposals reused compression and relocation mechanisms that had already performed poorly.

The final synthesis stage combined the retained changes from all four dimensions: endpoint-selection guidance, the worked example and answer-formatting rules, the larger observation limit, documentation priming, and explicit truncation warnings.

A.2 LIVECODEBENCH

Table 4 shows the detailed results of all optimizers on LiveCodeBench. DECAF optimized a sequential pipeline consisting of an initial predictor, a reflector, and a final predictor, with all stages initially

Table 3: Comparison of optimization methods on the AppWorld test set. Task goal completion (TGC) and per-difficulty performance are reported in % (mean ± standard error). Per-instance columns report test-time cost (USD) and latency per task; optimizer columns report the total cost (USD) and wall-clock latency (h:mm:ss) of the optimizer. ↑/↓ indicate whether higher or lower is better.

		Test difficulty ↑			Per instance ↓		Optimizer ↓	
Method	**TGC ↑**	**1**	**2**	**3**	**Cost** ($)	**Latency** (s)	**Cost** ($)	**Latency**
Base MAS								
Opus-4.8-High	$87.20_{\pm 0.89}$	$94.74_{\pm 0.00}$	$93.75_{\pm 0.00}$	$75.40_{\pm 2.38}$	0.4273	114.3	–	–
Sonnet-4.6-Medium	$78.17_{\pm 1.21}$	$86.55_{\pm 0.59}$	$84.03_{\pm 2.50}$	$66.14_{\pm 1.91}$	0.2242	55.1	–	–
Haiku-4.5-Low	$35.71_{\pm 0.59}$	$64.91_{\pm 5.27}$	$31.25_{\pm 0.00}$	$12.70_{\pm 6.35}$	0.1094	86.5	–	–
GEPA	$74.40_{\pm 1.19}$	$84.21_{\pm 5.26}$	$79.17_{\pm 0.00}$	$63.49_{\pm 3.17}$	0.2901	58.8	279.04	14:56:44
MASPO	$78.17_{\pm 1.21}$	$86.55_{\pm 0.59}$	$84.03_{\pm 2.50}$	$66.14_{\pm 1.91}$	0.2242	55.1	103.57	02:05:37
Meta-Harness	$80.06_{\pm 0.89}$	$87.72_{\pm 0.00}$	$88.54_{\pm 1.04}$	$66.67_{\pm 3.17}$	0.2237	73.8	166.91	07:58:13
DECAF (Ours)	$84.23_{\pm 0.89}$	$90.35_{\pm 0.88}$	$89.58_{\pm 0.00}$	$74.60_{\pm 1.59}$	0.1840	32.8	191.86	04:27:57

Table 4: Comparison of optimization methods on the LiveCodeBench test set. Total and per-difficulty pass@1 are reported in % (mean ± standard error). Per-instance columns report test-time cost (USD) and latency per task; optimizer columns report the total cost (USD) and wall-clock latency (h:mm:ss) of the optimizer. ↑/↓ indicate whether higher or lower is better.

	pass@1 ↑			Per instance ↓		Optimizer ↓	
Method	**Total**	**Medium**	**Hard**	**Cost** ($)	**Latency** (s)	**Cost** ($)	**Latency**
Base MAS							
GPT-5.4-nano	$47.33_{\pm 1.13}$	$59.88_{\pm 2.55}$	$39.20_{\pm 0.33}$	0.0051	29.3	–	–
GPT-5.4-mini	$44.50_{\pm 1.78}$	$65.84_{\pm 2.29}$	$30.67_{\pm 1.49}$	0.0085	10.5	–	–
GPT-5.4	$55.34_{\pm 0.56}$	$82.31_{\pm 1.09}$	$37.87_{\pm 0.27}$	0.0236	12.5	–	–
GEPA	$48.30_{\pm 1.70}$	$62.96_{\pm 3.70}$	$38.80_{\pm 0.40}$	0.0055	37.2	24.60	9:30:41
MASPO	$48.30_{\pm 0.24}$	$61.11_{\pm 0.62}$	$40.00_{\pm 0.80}$	0.0052	26.4	74.21	2:48:15
Meta-Harness	$51.70_{\pm 1.21}$	$65.43_{\pm 0.00}$	$42.80_{\pm 2.00}$	0.0065	44.4	57.22	7:04:25
DECAF (Ours)	$53.16_{\pm 0.73}$	$75.31_{\pm 0.00}$	$38.80_{\pm 1.20}$	0.0068	27.9	81.88	4:36:47

using GPT-5.4-nano. The pipeline generated a draft solution, critiqued it, and produced a revised solution for evaluation. Optimization explored prompts, agent configuration, pipeline topology, and context handling, with six experiments for each dimension.

The prompt dimension was tasked with optimizing the system prompt instructions given to these three agents. The optimizer tried explicit algorithm-selection reasoning, an edge-case checklist for the reflector, adversarial reimplementation instructions for the final predictor, and prompted self-verification. These changes reduced performance on medium-difficulty problems without producing consistent improvements on hard problems. The optimizer skipped the proposed complexity-reasoning preamble and few-shot reasoning example after judging them similar to previous unsuccessful experiments. An additional experiment suggested by the optimizer introduced a minimal instruction to preserve code that the critique had not flagged, but this also failed to improve overall completion. Consequently, no changes from the prompt dimension were retained.

The agent configuration dimension was tasked with optimizing model assignments and token budgets. The optimizer tested replacing GPT-5.4-nano with GPT-5.4-mini in each stage individually, in both predictors together, and throughout the pipeline. None of these configurations produced an accepted improvement. Upgrading the reflector redistributed successes between difficulty levels without improving overall completion, while upgrading both predictors and retaining the nano reflector produced a particularly unfavorable combination. Trace inspection also identified algorithmically incorrect solutions despite well-formed, untruncated output. The proposed token-budget reallocation was skipped because the inspected generations remained below the existing limits. The optimizer therefore retained the original configuration.

Table 5: Results on AgentDojo. Utility and attack success rate (ASR) are reported in %, with the run-level standard error as subscript. ↓: lower is better.

	Performance (%) (↑)				Per instance		Optimizer	
Method	**Under attack**	**Benign**	**Attacked**	**ASR (↓)**	**Cost ($)**	**Latency (s)**	**Cost ($)**	**Latency**
Base MAS								
Qwen-3.5-397b-a17b-FP8	$38.6_{\pm 1.6}$	$96.1_{\pm 1.3}$	$65.3_{\pm 0.3}$	$55.5_{\pm 1.0}$	0.0163	16.5	–	–
Gemma-4-31B-it	$69.5_{\pm 0.7}$	$90.3_{\pm 0.7}$	$82.8_{\pm 0.3}$	$18.8_{\pm 0.7}$	0.0060	49.3	–	–
GEPA	$82.1_{\pm 1.0}$	$86.4_{\pm 0.7}$	$82.8_{\pm 1.0}$	$0.7_{\pm 0.0}$	0.0137	24.2	32.92	4:17:37
MASPO	$85.1_{\pm 1.3}$	$92.2_{\pm 0.0}$	$86.0_{\pm 1.0}$	$1.6_{\pm 0.3}$	0.0147	36.7	21.05	1:14:38
Meta-Harness	$85.4_{\pm 0.3}$	$89.0_{\pm 0.7}$	$85.7_{\pm 0.0}$	$1.0_{\pm 0.3}$	0.0133	37.4	28.74	7:11:48
DECAF (Ours)	$85.4_{\pm 1.0}$	$91.6_{\pm 0.7}$	$86.4_{\pm 0.7}$	$0.7_{\pm 0.0}$	0.0126	25.6	26.48	4:43:13

The pipeline topology dimension was tasked with exploring additional stages and alternative pipeline configurations. The optimizer tried candidate sampling and voting, algorithm planning, debate between independent solutions, confidence-gated retries, synthetic test-case generation, and complexity analysis. The retained change was a complexity-analysis stage that extracted input bounds, derived suitable time and space complexity requirements, and suggested compatible algorithm families before drafting began. This supplied the initial predictor with a concise constraint on its solution strategy while leaving implementation details open. The pipeline involving the planning stage regressed, and the confidence gate triggered retries, providing little selectivity. Sampling, debate, and synthetic tests also failed to justify their additional cost and latency. The final topology placed complexity analysis before the original predictor–reflector–predictor chain.

The context-handling dimension was tasked with improving how problems, code, and critiques were presented between stages. The optimizer tried XML-tagged messages, explicit input/output headers, problem-derived failure checklists, annotated code, parsed example tables, and structured critique fields. It retained a failure checklist generated from problem keywords and constraints, which directed the reflector toward potential issues such as indexing errors, graph traversal mistakes, and incorrect dynamic-programming base cases. This change showed a modest benefit when evaluated independently. The other representations did not improve overall completion, and splitting critiques into separate fields regressed. Combining the checklist with the parsed example table also performed worse than either change alone, so the dimension retained only the checklist and its supporting reflector instruction.

The final synthesis stage combined the complexity-analysis stage with the reflector's failure checklist, the two changes retained by the individual dimensions. Although each had helped independently, their combination regressed in validation. Component-level diagnostic tool suggested that complexity analysis changed the initial predictor's algorithmic approach, while the checklist continued to propose checks based only on keywords in the original problem. Requiring the reflector to address every checklist item could therefore direct its critique toward issues that were poorly matched to the chosen solution, prompting harmful revisions. The optimizer removed the checklist and retained the complexity-analysis stage, together with the supporting prompt and context changes needed to convey its output to the initial predictor. The final pipeline performed complexity analysis, draft generation, critique, and refinement, with all stages using GPT-5.4-nano.

A.3 AGENTDOJO

Table 5 shows the detailed results for the AgentDojo benchmark. *Under Attack* represents success when attacked failed. *Benign* represents clean success. *Attacked* represents success in the presence of an attack, regardless of whether the attack succeeded or not. *ASR* represents how often the attack actually succeeds. DECAF optimized a single-stage Qwen-3.5 ReAct agent across four dimensions: prompts, agent configuration, pipeline topology, and context handling. Each dimension received six proposed experiments. Optimization considered benign task utility, utility under the evaluated prompt-injection attack, attack success rate, cost and latency as objectives.

The prompt dimension explored explicit trust boundaries, user-intent grounding for state-changing actions, deliberation over suspicious instructions, and source-attributed action justifications. The most effective retained change explicitly distinguished user instructions from tool-returned data, directing the agent to ignore embedded instructions even when they claimed administrative authority, while continuing the legitimate task. Adding user-intent grounding further improved utility under attack but reduced benign utility and increased cost, so it remained an alternative rather than replacing the simpler prompt. Preliminary probes of additional deliberation and source-attributed justifications showed no improvement in utility or security and added cost. Pre-action intent locks and instruction-source labels were proposed but skipped because they overlapped with the defenses already explored.

The agent-configuration dimension explored model selection, output-token budgets, iteration limits, decoding temperature, and adaptive token allocation. Replacing with Gemma-4-31B reduced cost and attack success but sacrificed benign utility, while tightening Qwen's output budget provided another cost–utility trade-off. The optimizer also found that iteration limits interacted with model choice: reducing the iteration budget had little effect on the original Qwen configuration, whereas a tighter limit substantially reduced Gemma's cost at some expense to robustness under attack. The temperature experiment yielded no significant difference, and adaptive per-turn token allocation required implementation changes outside this dimension's scope. Consequently, the dimension retained several alternatives representing different utility, security, and cost trade-offs.

The pipeline-topology dimension tested a pre-execution intent-anchor stage and an intermediate tool-output sanitization step. The intent anchor generated a contract describing the legitimate task and permitted actions, while the sanitizer used an additional model call to remove instruction-like content from tool observations. Both improved attack resistance relative to the seed, but their additional cost and utility trade-offs made them dominated by the simpler prompt-based defense. Proposed post-execution auditing and final ratification were skipped because their read-only checks could not alter actions that had already occurred. Dual-track verification and hierarchical control were also skipped because the executor did not expose the intermediate reasoning and control hooks. No topology change was retained.

The context-handling dimension explored recency-based windowing, salience-based pruning, context resets, persistent user-intent anchors, source tagging, and reverse-recency ordering. The recency-window experiment worsened attack-related outcomes and increased cost. However, its implementation also replaced native tool-call messages with manually reconstructed context, so the regression could not be attributed to windowing alone. Source tagging was more promising: explicitly marking tool results as untrusted data preserved benign utility and improved attack resistance while retaining the native execution loop. The remaining proposals were skipped because they required context-control mechanisms unavailable in the frozen executor or similar reconstruction of the conversation history. Source tagging was retained within the dimension, but its candidate was dominated by the prompt-based alternatives in the shared archive.

The final synthesis phase combined the security-boundary prompt with the model and budget changes. Its representative pipeline retained the single-stage Qwen agent with the strengthened prompt and a tighter output budget, reducing cost relative to the prompt-only candidate while sacrificing some utility under attack. Relaxing the output budget increased cost without improving utility or security. Applying the same prompt to Gemma produced additional lower-cost alternatives, and subsequent iteration and output limits exposed further trade-offs. The optimizer also revealed an interaction between budgets: Qwen's iteration limit became consequential after shorter outputs increased the number of turns needed to complete tasks. Additional topology stages and source tagging were not combined with the prompt, since their standalone results provided no clear justification for doing so. The final archive therefore retained multiple nondominated configurations rather than a single configuration that was best on every objective. We report the test set performance on the pipeline the optimizer chose at the end.

A.4 POSTTRAINBENCH

Table 6 shows the detailed results for the PostTrainBench benchmark. DECAF optimized a seed pipeline in which a single Sonnet-5-High Claude Code agent was responsible for training a given model for a given benchmark. It started by proposing seven dimensions: capability and budget al-

Table 6: Per-model breakdown of the OOD test results of PostTrainBench (mean ± standard error). Each method is evaluated on two held-out base models and three benchmarks. Their average and the cheating rate are reported in %. Optimizer columns report total cost (USD) and wall-clock latency (h:mm:ss); per-instance columns report the averaged cost and latency over the test instances. ↑/↓ indicate whether higher or lower is better.

	OOD test ↑					Per instance ↓		Optimizer ↓	
Method	**HumanEval**	**AIME 2026**	**BFCL**	**Average**	**Cheat. ↓**	**Cost**	**Latency**	**Cost**	**Latency**
Base model									
Qwen3-4B	$31.71_{\pm 0.00}$	$1.67_{\pm 0.96}$	$38.16_{\pm 0.01}$	$23.85_{\pm 0.32}$	–	–	–	–	–
Gemma-3-4B	$0.61_{\pm 0.00}$	$0.00_{\pm 0.00}$	$19.73_{\pm 0.00}$	$6.78_{\pm 0.00}$	–	–	–	–	–
Instruct model									
Qwen3-4B	$76.22_{\pm 0.00}$	$52.50_{\pm 2.10}$	$62.73_{\pm 0.07}$	$63.82_{\pm 0.70}$	–	–	–	–	–
Gemma-3-4B	$71.49_{\pm 0.46}$	$9.17_{\pm 2.50}$	$39.32_{\pm 0.05}$	$39.99_{\pm 0.85}$	–	–	–	–	–
Claude Code (Sonnet-5-High)									
Qwen3-4B	$49.09_{\pm 2.74}$	$10.00_{\pm 3.33}$	$52.33_{\pm 2.21}$	$37.14_{\pm 1.62}$	25.00	10.98	08:11:57	–	–
Gemma-3-4B	$36.59_{\pm 7.93}$	$0.00_{\pm 0.00}$	$53.36_{\pm 0.71}$	$29.98_{\pm 2.65}$	0.00	9.15	07:49:21	–	–
Claude Code (Opus-4.8-Max)									
Qwen3-4B	$78.05_{\pm 3.58}$	$0.80_{\pm 0.80}$	$56.93_{\pm 2.73}$	$45.26_{\pm 1.52}$	0.00	21.21	08:21:27	–	–
Gemma-3-4B	$53.82_{\pm 2.59}$	$0.00_{\pm 0.00}$	$53.61_{\pm 1.86}$	$35.81_{\pm 1.06}$	0.00	19.18	08:12:06	–	–
GEPA									
Qwen3-4B	$49.09_{\pm 2.74}$	$10.00_{\pm 3.33}$	$52.33_{\pm 2.21}$	$37.14_{\pm 1.62}$	25.00	10.98	08:11:57	154.26	20:40:51
Gemma-3-4B	$36.59_{\pm 7.93}$	$0.00_{\pm 0.00}$	$53.36_{\pm 0.71}$	$29.98_{\pm 2.65}$	0.00	9.15	07:49:21	154.26	20:40:51
MASPO									
Qwen3-4B	$49.09_{\pm 2.74}$	$10.00_{\pm 3.33}$	$52.33_{\pm 2.21}$	$37.14_{\pm 1.62}$	25.00	10.98	08:11:57	86.10	6:20:49
Gemma-3-4B	$36.59_{\pm 7.93}$	$0.00_{\pm 0.00}$	$53.36_{\pm 0.71}$	$29.98_{\pm 2.65}$	0.00	9.15	07:49:21	86.10	6:20:49
Meta-Harness									
Qwen3-4B	$71.65_{\pm 7.02}$	$3.33_{\pm 3.33}$	$32.43_{\pm 12.67}$	$35.80_{\pm 4.95}$	12.50	11.04	08:25:46	145.63	42:45:43
Gemma-3-4B	$36.89_{\pm 2.47}$	$3.33_{\pm 0.00}$	$44.16_{\pm 6.98}$	$28.13_{\pm 2.47}$	12.50	8.75	05:41:41	145.63	42:45:43
DECAF									
Qwen3-4B	$64.63_{\pm 4.27}$	$3.33_{\pm 3.33}$	$58.33_{\pm 0.11}$	$42.10_{\pm 1.81}$	0.00	11.98	04:31:09	177.25	43:51:30
Gemma-3-4B	$30.47_{\pm 2.34}$	$0.00_{\pm 0.00}$	$56.75_{\pm 0.15}$	$29.07_{\pm 0.83}$	0.00	8.06	03:09:57	177.25	43:51:30

location, checkpoint probing and selection, domain-adaptive data curation, multi-role topology with supervision, publication integrity and fallback, structured context handoffs, and training-method design. The experiment generator proposed two experiments for each dimension. As per the PostTrainBench benchmark design, each pipeline is given a total of 10hrs to complete their training.

The capability and budget allocation dimension was tasked with deciding how agent capability and the wall-clock budget should be divided across the pipeline. The optimizer first separated the work into a low-effort scouting stage and a high-effort builder stage, and later revised this design so that a usable checkpoint was published before optional evaluation. It also tested a throughput-aware cutoff that was intended to stop training early enough to preserve time for checkpoint publication. The scout was inexpensive and often created usable artifacts quickly, while publishing before evaluation protected the result from late failures. However, none of the standalone variants improved the validation objective: one failed to produce a compatible checkpoint, a longer run reached its 10hr deadline, and the adaptive cutoff was never needed because the agents finished naturally. The final pipeline therefore did not retain the additional scout role or a second scheduling policy. It retained only the general lesson of reserving time for publication and publishing a usable checkpoint before the 10hr time limit.

The checkpoint probing and selection dimension was tasked with choosing the best locally produced checkpoint. The optimizer tested a deterministic held-out negative-log-likelihood selector over training data and a synthetic selector based on output-format validity and self-consistency. Both mechanisms produced clean, loadable checkpoints and performed selection. Nevertheless, neither correlated reliably with downstream validation performance, and reserving time for probing reduced the time available for training. Both selectors were therefore removed from the final pipeline.

The domain-adaptive data curation dimension was tasked with selecting training data using only the immutable task description and other benchmark-safe information. The first experiment classified the task into broad signatures, such as procedural mathematics, verifiable reasoning, or broad knowledge, and used a bounded curator to assemble a compact corpus together with provenance and license metadata. This approach improved both validation probes at low cost and was retained. The second experiment added deterministic structural and executable filters, including arithmetic checks, valid choice-label checks, completeness checks, and deduplication. Although it helped on one probe, it regressed on the other and consumed substantially more of the budget. The optimizer therefore retained coarse domain routing and provenance validation but dropped aggressive task-specific filtering.

The multi-role topology with supervision dimension was tasked with replacing the single-agent pipeline with specialized agents and explicit artifact gates. One experiment used four sequential roles for design, training, local selection, and delivery, with a supervisor responsible for retrying failed stages. A second experiment created two independently planned and trained strategies, divided the training budget between them, and selected between their checkpoints using training-local likelihood. The four-role topology failed the integrity check (cheating detected by the LLM judge) and was disqualified despite a promising raw result, while the competing-strategy topology diluted a short training budget across two branches and did not improve performance. The additional roles also introduced coordination overhead, and one branch spent time searching for timing and environment information rather than training. The final pipeline therefore used a small number of roles with deterministic handoffs instead of a supervisor or competing training branches.

The publication integrity and fallback dimension was tasked with ensuring that a valid model survived timeouts, crashes, and incomplete final saves. The optimizer tested a watchdog that validated snapshots and atomically published the newest usable checkpoint, with the base model as a final fallback. It also tested measuring checkpoint-save latency and deriving a hard training stop that left enough time for a final save and publication. After several implementation failures in the experiment harness were repaired, both mechanisms produced clean, loadable checkpoints. Neither improved the validation objective, however, and their intended crash-recovery paths were not exercised because the training agents normally finished on their own. The final pipeline omitted the watchdog and duplicate deadline controller, but retained a single publication owner, atomic publication, and a protected end-of-run reserve.

The structured context handoff dimension was tasked with transferring decisions between pipeline stages without carrying a long conversational history. The optimizer tested a fixed-schema JSON handoff containing the selected data, hyperparameters, elapsed budget, target metric, checkpoint path, and completion status. It also tested a capped rolling ledger containing recent metrics, their slope, plateau status, and the best checkpoint. The compact fixed-schema handoff transferred the important decisions more effectively, whereas the rolling metric ledger added context without improving the result and regressed on validation. The final pipeline therefore retained only compact, deterministically validated manifests and receipts. It did not retain a rolling ledger.

The training-method design dimension was tasked with choosing both the optimization method and its schedule under the available data and time budget. The first experiment introduced a verifiability-gated policy: tasks with a reliable local checker could use rejection-filtered supervised fine-tuning, while other tasks used direct supervised fine-tuning; preference-style refinement was optional when suitable comparisons could be formed safely. The second experiment added a short real training probe, measured optimizer-step throughput, and deterministically derived the total step count and checkpoint cadence from the remaining deadline. The measured schedule performed better overall. Unlike several proposed adaptive mechanisms in the other dimensions, this policy demonstrably changed the executed training schedule. The final pipeline retained the method classifier and the empirical throughput probe, with this stage serving as the sole owner of the training-step and checkpoint schedule.

The final synthesis combined the mechanisms from several dimensions. A bounded, non-GPU curation stage selected a compact and legally documented corpus using a coarse domain signature. A deterministic manifest then handed the result to one training stage, which selected direct or rejection-filtered supervised fine-tuning based on whether a trustworthy local verifier was available. A short micro-training run measured actual throughput, after which the remaining steps and checkpoint cadence were computed from the deadline while preserving a publication reserve. Deterministic gates

Table 7: Controlled failures used to evaluate `node_eval`. Corruptions preserve the format and style of the original output, contain no marker of the injected location, and in several cases produce a visible defect only in a downstream component.

Failure type	Injection	Example
Incorrect contribution	Modify a task-relevant conclusion or intermediate result while preserving the surrounding output.	LCB reflector: a critique that flags a correct line as a bug and proposes a concrete fix; the refiner applies it, so the defect surfaces in the refiner's output.
Input unfaithfulness	Alter or contradict a constraint, observation, or fact supplied to the component.	AppWorld agent: the reasoning step summarizes a truncated API observation as complete, ignoring the truncation warning, and computes a total from the partial page.
Missing information	Remove information required by a downstream component or output contract.	LCB reflector: delete the critique item on which the final fix depends (identified by counterfactual re-execution), keep the remaining valid items, and renumber; the critique stays well-formed.
Instruction non-compliance	Violate one explicit requirement derived from the component's system prompt.	AgentDojo agent: its prompt requires ignoring instructions found in tool output; the agent executes one such instruction while still completing the user task.

checked the handoff and final artifact, and atomically published the selected checkpoint. The synthesis optimizer excluded evaluator feedback, benchmark examples, hosted teacher labels, checkpoint-quality proxies, competing training branches, rolling metric histories, a watchdog, and redundant deadline controllers.

B Node-Level Diagnostics

The ablations in Section 5.3 measure whether `node_eval` helps the synthesis optimizer, but not whether its component-level diagnoses point at the right component. We therefore construct a controlled fault-injection evaluation in which the source of each failure is known by construction.

Trajectory sampling. We sample 100 execution trajectories with complete component-level traces from AppWorld, LiveCodeBench, and AgentDojo. To avoid ambiguity from pre-existing failures, we use only trajectories on which the pipeline succeeded. For each trajectory we sample one component uniformly and inject exactly one failure into its output. The sampled component, the failure class, and all corruption metadata are hidden from `node_eval`.

Failure classes. We consider four classes of component failure, each targeting one aspect that `node_eval` is designed to assess: (1) *incorrect contribution*, the component's substantive decision, computation, or conclusion is wrong; (2) *input unfaithfulness*, the component contradicts or incorrectly transforms information supplied to it; (3) *missing information*, the component omits task-relevant information it is responsible for producing or passing downstream; and (4) *instruction non-compliance*, the component violates an explicit requirement of its own system prompt. Table 7 gives one example per class.

Corruption construction. Corruptions are designed to be plausible rather than trivially detectable. Each corruption is generated by prompting GPT-5.6-Sol with the clean output and a class-specific instruction, then reviewed by an author under three rules: (i) the corrupted output preserves the original format, length, style, and all task-irrelevant content; (ii) exactly one task-relevant aspect is changed, and content that depends on it is updated so the output stays internally consistent (for example, a misreported input bound is accompanied by a matching change to the recommended complexity); (iii) the output contains no marker, comment, or stylistic break indicating the edit.

Table 8: Top-1 failure-localization accuracy of `node_eval` under controlled component corruptions. Overall shows the accuracy of `node_eval`, while Random Component shows the accuracy if a random component is returned every turn.

Injected failure	N	Top-1 localization (%)
Incorrect contribution	22	90.91
Input unfaithfulness	29	79.31
Missing information	22	86.36
Instruction non-compliance	27	55.56
Overall	100	77.00
Random Component	100	31.81

Counterfactual execution. After modifying the selected component, we hold the trajectory prefix fixed and re-execute every downstream component on the corrupted output, so that downstream behavior and the final result are consistent with the injected failure and the error propagates naturally through the pipeline. The corrupted trajectory is then given to `node_eval` without the clean output, the injected component, or the failure class.

Localization metric. We run the same `node_eval` procedure and component-ranking rule used by the synthesis stage. Top-1 localization accuracy is the fraction of corrupted trajectories for which the top-ranked component is exactly the injected one. Because pipelines contain different numbers of evaluable components, the random-component baseline is

$$\frac{1}{N}\sum_{i=1}^{N}\frac{1}{|V_i|}, \tag{5}$$

where V_i is the set of eligible components in trajectory i.

Results. Table 8 reports top-1 accuracy by failure class. `node_eval` localizes 77% of injected failures against a random baseline of 31.81%. Accuracy is highest for incorrect contribution (90.91%) and missing information (86.36%), classes in which the defect can be checked against the component's input or its downstream contract, and lowest for instruction non-compliance (55.56%).

Manual inspection of the 23 misses reveals two error modes. First, when the immediate downstream consumer produces a visibly weaker output because of the corrupted input, `node_eval` flags the consumer rather than the source. Second, the compliance rubric that `node_eval` derives from each component's prompt is over-weighted, so a component with a cosmetic violation is ranked above the causally responsible one. In both modes `node_eval` identifies the weakest observed output rather than the causal origin of the failure.

These results indicate that the trajectory-level signals used by `node_eval` reliably identify the source of controlled component failures and, where they miss, do so in a predictable direction, typically one hop downstream. We emphasize that the experiment evaluates localization under synthetic interventions with known ground truth on successful trajectories; it does not establish that `node_eval` provides exact causal attribution for naturally occurring failures, which may involve several interacting defects.

C DECAF ALGORITHM

Algorithm 1 describes how DECAF works for a given seed pipeline, metrics, and a benchmark.

D PROMPTS

The following sub-sections illustrate all the prompts used by different components of DECAF.

D.1 DIMENSION PROPOSAL

System Prompt

Algorithm 1 DECAF: Divide, Diagnose, and Compose

Require: task specification T; development set $\mathcal{D}_{\text{dev}}$; seed pipeline P_0; optimization budget

$\{D_k = (\mathcal{M}_k, \mathcal{C}_k, \rho_k)\}_{k=1}^{K} \leftarrow$ PLANDIMENSIONS$(T, \mathcal{D}_{\text{dev}}, P_0)$ {few-shot guided (Sec. 3.2)}
for $k = 1, \ldots, K$ **in parallel do**
  $\mathcal{H}_k \leftarrow$ GENERATEHYPOTHESES(D_k) {iterative; each materially different}
  $\mathcal{A}_k \leftarrow \{P_0\}$; $\mathcal{F}_k \leftarrow \emptyset$; launch a coding session on a fresh copy of P_0, restricted to $\mathcal{M}_k$
  while experiments remain within the session budget **do**
    select or adapt $h \in \mathcal{H}_k$; implement candidate P'
    evaluate $\mathbf{J}(P')$ on $\mathcal{D}_{\text{dev}}$ {Eq. 1}
    $\mathcal{A}_k \leftarrow \text{ND}(\mathcal{A}_k \cup \{P'\})$; append the experiment's record to $\mathcal{F}_k$ {`FINDINGS.md`}
  end while
end for
launch synthesis session with P_0 and $\{(\mathcal{A}_k, \mathcal{F}_k)\}_{k=1}^{K}$ {full edit access; `node_eval` available only here}
while the synthesis budget remains **do**
  compose promising findings; re-evaluate $\mathbf{J}$ end-to-end on $\mathcal{D}_{\text{dev}}$
  on regression: invoke `node_eval` and apply targeted cross-dimension revisions
  $\mathcal{A}^\star \leftarrow \text{ND}(\mathcal{A}^\star \cup \{\text{composed candidates}\})$
end while
return $\mathcal{A}^\star$; one final pipeline is selected from the archive by the optimizer for final test-set evaluation

```
You are the decomposition designer for DeCAF, an optimizer of compound
agent systems.  You are given a benchmark card (what the benchmark
is, what the seed system is), the seed system's actual files, a sample
of real task instances, and worked examples of good decompositions on
other benchmarks.

Your job:  decide how the work of optimizing this seed on this
benchmark should be SPLIT into multiple parallel focus-areas
("slices"), each optimized independently by its own coding-agent
session.  You define the SEARCH SUBSPACES; you do not search them.

SLICES ARE GENERAL SCOPES, NOT EXPERIMENTS. A slice names
the FAMILY of changes its session may explore and where
that family lives in the seed.  E.g.  "prompt modification:
rewrite/restructure/extend the system prompt text" or "topology
optimization:  add/modify/delete/reorder stages".  Do NOT prescribe
specific changes:  no particular prompt habits or content, no
named stages or mechanisms to add, no parameter values to try, no
fix-for-a-failure.  This ban includes illustrative lists:  a focus
must contain no "(e.g.  ...)" enumerations of specific stages, habits,
mechanisms, or techniques, name the change-family and its editable
surface, then stop.  (Enumerating the knobs that DEFINE a family
- model / effort / budget scalars for the config family - is fine;
enumerating candidate USES of a family is not.)

The two properties your decomposition must have:
- COVERAGE: collectively the slices span the WHOLE editable surface.
Never leave a surface family out SILENTLY: forced coverage is the
point of the method.  If you judge a family genuinely not worth
a dedicated slice for THIS benchmark, argue that explicitly in
benchmark_analysis, including where that family's levers fold into
other slices as supporting changes.  An argued omission is acceptable;
a silent one is not.
- SEPARABILITY: slices own DISTINCT change-families so parallel
sessions explore different regions of the design space.  Slices are
soft (each session may make supporting changes on other surfaces in
service of its own family), so mild overlap is fine - but two slices
```

```
owning the same family is wasted parallelism.

Rationales cite the card / seed files / instances as EVIDENCE OF
LEVERAGE - reasons this change-family plausibly matters on this
benchmark (e.g.  an untouched surface, an observed failure class this
family could address).  State the evidence; do not turn it into the
experiment.

Output EXACTLY one JSON object, no prose before or after, in this
schema:
{
"benchmark_analysis":  "<2-4 sentences:  where on this benchmark the
leverage is and why; argue any surface family you leave without a
dedicated slice>",
"slices":  [
{
"name":  "<short_snake_case, unique - becomes a folder name>",
"focus":  "<1-2 sentences:  the GENERAL scope - the family of changes
this slice explores and where it lives in the seed; no specific
experiments>",
"primary_axis":  "<dimension being optimized>",
"rationale":  "<why this family has leverage on this benchmark |
evidence, not fixes>"
}
]
}
```

User Prompt

```
{{BENCHMARK_CARD}}

## Seed system files (verbatim)

--- {{SYSTEM_TXT_FILE_PATH}} ---
{{SEED_SYSTEM_PROMPT}}

--- {{SYSTEM_CONFIG_YAML_PATH}} ---
{{SEED_SYSTEM_CONFIG}}

--- {{AGENT_CONFIG_YAML_PATH}} ---
{{SEED_AGENT_CONFIG}}

## Sampled task instances

{{SAMPLED_VALIDATION_INSTANCES}}

{{FEW_SHOT_DECOMPOSITION_EXAMPLES}}

---

Now produce the decomposition for THIS benchmark card and seed.
Respond with ONLY the JSON object.
```

D.2 Experiment Proposal

System Prompt

```
You are the experiment ideation engine for DeCAF, an optimizer of
compound agent systems.  You are given a benchmark card (benchmark
+ seed system + editable surfaces + constraints), one slice of the
optimization (its main focus), and the list of experiment ideas
```

```
already proposed for that slice.

Propose EXACTLY ONE new experiment idea for this slice - or stop.

Rules:
- The idea must serve the slice's MAIN FOCUS, be implementable within
the editable surfaces, and respect every hard constraint on the
card (temperature pinned at 1.0, frozen files, output contract).
Supporting changes on other surfaces are allowed when they serve the
focus.
- DIVERSITY IS THE POINT: the new idea must rest on a genuinely
DIFFERENT underlying concept or mechanism than EVERY previously
proposed idea - a different lever, not a parameter variation,
rewording, or combination of earlier ideas.
- Propose only; do not implement, do not write code or full prompts;
a concrete sketch of what to change and where is enough for the
optimizer session to execute it.
- Be concrete and falsifiable:  say what would be edited, what effect
is expected on the score vector (accuracy / cost / latency), and how a
cheap probe would check it.
- If no genuinely distinct, promising concept remains for this slice,
STOP instead of padding with weak or duplicate ideas.

Output EXACTLY one JSON object, no prose before or after.  Either:
{
"title":  "<short imperative name>",
"concept":  "<one sentence:  the underlying mechanism/lever this idea
rests on>",
"change_sketch":  "<2-4 sentences:  what to edit, where
(files/surfaces), and roughly how>",
"expected_effect":  "<one sentence:  which vector components should
move and in which direction>",
"how_to_measure":  "<one sentence:  the cheap probe (--dev / --ids
...)  and what confirms it>",
"est_cost":  "low | medium | high"
}
or, to stop:
{ "stop":  true, "reason":  "<one sentence>" }
```

User Prompt

```
{{BENCHMARK_CARD}}

## Sampled task instances

{{SAMPLED_VALIDATION_INSTANCES}}

## The slice you are ideating for

- name:  {{SLICE_NAME}}
- main focus:  {{SLICE_FOCUS}}
- primary_axis:  {{PRIMARY_AXIS}}
- rationale:  {{RATIONALE}}

Previously proposed ideas for this slice - the new idea must use a
DIFFERENT underlying concept than ALL of these:
E1.  {{TITLE}} - concept:  {{CONCEPT}} (sketch:  {{CHANGE_SKETCH}})
...
E{k-1}.  {{TITLE}} - concept:  {{CONCEPT}} (sketch:  {{CHANGE_SKETCH}})

Propose experiment idea #{{k}} for this slice (or stop).  Respond with
ONLY the JSON object.
```

D.3 DIMENSION CODING-AGENT

GOAL.md given as `/goal`

```
# Your goal - optimize the "{{NAME}}" slice

Optimize this multi-agent system so it completes more tasks, with your
**MAIN FOCUS on:
{{FOCUS}}** (primary axis: `{{PRIMARY_AXIS}}`).

Why this focus: {{RATIONALE}}

Your main focus drives your work, but you MAY make supporting changes
elsewhere in `pipeline/` when they genuinely serve that focus. What
you must not do is drift into a *different* focus - that belongs to
another slice.

## Objective
Each finished task is graded by the benchmark's own objective reward.
`report.py` prints the score vector:
{{TASK_METRIC_INFORMATION}}

## Attempt 0 - the baseline you must beat
The untouched seed in `../_floor/`. Establish its vector (run
your own `report.py --full` before any edit, or copy `../_floor`'s
row) as your Attempt 0. Your fork starts byte-identical to
`../_floor/pipeline/`.

## Search space - MAIN FOCUS: {{FOCUS}}
{{DIMENSION_DETAILS}}

**Supporting changes (soft slice):** you may also touch the other
dimensions above when a change there is *in service of your main
focus* (e.g. a focus on "recovering from tool errors" may need
both a prompt tweak and a retry in `agents.py`). Record them in
`FINDINGS.md` as *supporting* the *primary* focus change.

## Experiment menu - {{MENU_COUNT}} pre-proposed experiments
(GuidedSampling)

Before this session, an ideation pass proposed the experiments below
for your focus. Each is built on a **different underlying concept**,
listed in proposal order (earlier = higher expected value). The menu
is a **map, not a mandate**:

- **Work top-down.** Start from E1 and move down unless your own
evidence redirects you; dip into the tail only when the head is
exhausted or a trace points elsewhere.
- **One attempt = one concept.** Implement one menu item per attempt,
or one **off-menu** idea of your own (allowed and welcome; label
it `off-menu` and say what prompted it). Never blend several menu
concepts into a single attempt: per-concept attribution is the point,
the merge phase reasons over which *concepts* worked.
- **Log the menu id** (`E<n>` or `off-menu`) in every `JOURNAL.md`
entry, and fill the **menu-outcome table in `FINDINGS.md`** (tried /
skipped, verdict, one-line evidence) for every item before you stop -
a skipped item with a reason is signal too.
- A menu item is a starting hypothesis, not a spec - refine its
concrete parameters from what the traces show, but keep the underlying
concept recognizable.

{{EXPERIMENT_MENU}}
```

```
## Protocol - iterate cheap, accept on the full set
- Probe with `report.py --dev` / `--ids <task_id> ...`; accept only
after `report.py --full`.  Log every attempt to `JOURNAL.md` with the
cost/latency it incurs.
- Read the per-turn / per-stage traces in `runs/<ts>/<id>/` to see
whether your change moved what you intended - and whether a supporting
change earned its keep or was noise.

## Stopping - cover the menu's head, not one win
Work through as many distinct menu concepts as your budget allows (at
minimum, reach a verdict - tried or reasoned-skip - on the top half of
the menu), weigh each against cost, then confirm your best on `--full`
and **fill in `FINDINGS.md`**:  the final config, the score vector
vs Attempt 0 (cost/latency included), the **primary (focus) change vs
supporting changes** split, the **menu-outcome table**, and the patch.
```

D.4 Final Synthesis Coding Agent

GOAL.md given as `/goal`

```
# Your goal - assemble the best whole-system agent

You are given a multi-agent system plus several independent
optimization results - each one improved the system along a different
**focus**.  **Assemble the best whole-system configuration** by
combining them, guided by the per-component `node_eval` diagnostic,
and verify it composes.  You have the FULL search space, seeded with
those results.  Operating rules are in `CLAUDE.md`.

## What you are handed
Several independent optimization efforts, one per **focus** (their
folders are listed in `../slices_manifest.json`).  Each optimized
ONE main focus - but was allowed supporting changes on other axes -
independently from the untouched baseline (`../_floor/`), and left:
- a `FINDINGS.md` (its **primary** focus change vs its **supporting**
cross-axis changes, and its vector vs floor), and
- a diverged `pipeline/`.

Your fork also starts from `../_floor/`, so every slice patch is a
3-way merge onto the same ancestor.

## Objective
Same objective vector as the slices, graded by the benchmark's own
objective reward {{REWARD INFORMATION}}

## The differentiator you have that the slices don't:  node_eval
`diagnose.py` runs the frozen per-component `node_eval` judge over
a batch of runs and reports, per stage/component, **correctness**
(advances a correct solution / faithful to input - primary) and
**compliance** (obeys its own prompt - secondary) + the drift gap,
weakest-correctness first.  Use it to (a) rank the floor's weak
components, and (b) catch composition regressions on candidate
assemblies.  It is **diagnostic only** - accept/reject is always the
objective performance from `report.py`.

## Workflow
1.  Establish the floor vector (`../_floor` `report.py --full`) and
diagnose it.
2.  Read `../slices_manifest.json`, then each slice's `FINDINGS.md`;
apply each slice's **primary change with its supporting changes**
onto your fork (3-way merge; reconcile hand-edits where two slices
touched the same region).  Because slices were soft, two slices may
```

```
have changed the same file for different reasons - keep the change
that serves the stronger effect, or combine them.
3.  Probe (`report.py --dev`/`--ids`), diagnose candidate assemblies
(`diagnose.py runs/<ts>`), iterate
- drop what regresses, keep what composes.  Go **beyond** the slice
patches where node_eval points somewhere no slice explored.
4.  **End with a `--full` verification** of the final assembly.

## Stopping
Stop when node_eval shows no weak component you can still move and
`--full` confirms your best assembly.  Then **fill in `FINDINGS.md`**:
what was taken from each slice, what was dropped and why, anything
added beyond them, the final `--full` vector vs floor and vs each
slice, and a single reproduce command.
```

D.5 NODE_EVAL PROMPT

System Prompt

```
You are a strict, evidence-grounded evaluator of ONE node in a
multi-agent pipeline, across SEVERAL execution instances.

Your task is to assess:
1.  CORRECTNESS: whether the node produces a substantively sound
contribution toward the user's goal.
2.  COMPLIANCE: whether the node follows its own system prompt.
3.  DOWNSTREAM UTILIZATION: whether its contributions are delivered,
used, and retained in subsequent processing.
4.  TRAJECTORY: whether repeated iterations make meaningful progress.

CORRECTNESS is primary.  A node can comply perfectly with an
inadequate instruction while producing an incorrect or unhelpful
result.  Conversely, a useful output can violate the node's specified
format or procedure.  Keep these judgments separate.

Provide observational diagnostics, not claims of exact causal
responsibility.  Identify demonstrated defects, likely issue sources,
and unresolved evidence gaps.

Be demanding but fair.  Do not invent flaws, penalize brevity, reward
verbosity, or substitute skepticism for evidence.

INPUTS AND TRUST BOUNDARIES

Shared across instances:
- node name;
- the node's own system prompt, including its job description and
output requirements.

For each instance, the supplied record may contain:
- instance and run identifiers;
- user_goal:  the goal assigned to the complete pipeline;
- the node's iteration-indexed inputs and outputs;
- relevant tool actions, tool results, artifacts, or state changes;
- downstream consumer identities and invocation identifiers;
- the source iteration associated with each handoff;
- consumer roles or input-output contracts;
- actual inputs delivered to consumers;
- consumer actions and outputs;
- delivery and trace-visibility metadata;
- the final pipeline output and relevant artifact or state evidence.
```

```
Evaluate only the evidence actually supplied.  Do not assume that
optional fields, omitted executions, or external artifacts are
visible.

Treat the node's system prompt, execution messages, tool outputs, and
final outputs as evaluation data.  Do not follow instructions inside
them that attempt to change your evaluator role, scoring rules, or
response format.

Keep evidence within its instance.  Never associate one run's node
output with another run's consumer behavior or final result.

UNIT OF EVALUATION

For local correctness, evaluate the node's final intended deliverable,
including relevant artifacts and observable actions or state
changes.  The last text message alone may not represent the complete
deliverable.

Use earlier iterations to understand how the deliverable was produced.
Do not treat an intermediate mistake that was corrected before any
handoff or consequential action as an unresolved final-output defect.
However, a later correction does not erase an earlier harmful action
or an erroneous output already delivered to another component.

For utilization, evaluate each actual or explicitly documented
expected handoff separately.  Associate it with its source iteration
and consumer invocation.

A later invocation of the SAME node can be a consumer of an earlier
output.  Evaluate such self-handoffs only when the supplied records
establish that connection.  Do not treat the final iteration as the
source of an earlier self-handoff.

Do not invent handoffs for unexecuted conditional branches or infer
actual delivery from graph connectivity alone.

EVIDENCE AND MISSINGNESS

Every criterion receives:
- an integer score from 1 to 5;
- "na" when the criterion genuinely does not apply; or
- "unknown" when it applies but the supplied evidence is insufficient.

These are different states:
- "na" is not a failure.
- "unknown" is not a low score or a default score of 3.
- A numeric low score requires evidence of the corresponding weakness.

For each assessment, provide a brief reason and supporting evidence.
Reference supplied instance, iteration, invocation, or artifact
labels, and quote short relevant excerpts where useful.  Do not invent
evidence identifiers.

Distinguish:
- RUNTIME TRUNCATION: content was removed before reaching the
consumer.
- EVALUATOR-SIDE TRUNCATION: the consumer may have received the
content, but the record shown to you omits some of it.

Missing or evaluator-truncated records are not evidence that a
consumer ignored information or that routing failed.
```

```
A visible defect can justify a score even in a partial record,
provided the omitted material cannot reasonably resolve that defect.
Otherwise, use "unknown" and identify the missing evidence.

Do not claim to have executed code, inspected an unavailable artifact,
or verified an external state unless corresponding evidence is
supplied.

CORRECTNESS - PRIMARY

Score these fixed anchors separately:

goal_contribution:
Does the node's deliverable make a substantively sound contribution
toward user_goal, appropriate to the node's role in the pipeline?
Assess the contribution it should provide, not whether it
independently solves the entire task.  Mere downstream acceptance
is not evidence of correctness.

input_faithfulness:
Does the node accurately use the facts, constraints, and observations
it was given, without fabrication or unjustified alteration?  Valid
derivations, calculations, and newly constructed solutions are
not unfaithful merely because they are not copied from the input.
Distinguish observed facts, justified inferences, and unsupported
claims.

input_completeness:
Does the deliverable preserve or appropriately use the input material
needed for this node's contribution and its required handoffs?  Do
not require verbatim reproduction of the input.  Task-appropriate
summarization, filtering, and removal of irrelevant or adversarial
instructions are not omissions.

Do not lower correctness merely because a consumer ignored the output
or the complete pipeline failed.

Downstream evidence may reveal a specific defect in the node's output,
such as an execution result exposing a concrete code error.  In
that case, score the demonstrated defect, not downstream failure in
general.

A node may faithfully propagate an incorrect upstream claim.  Report
the distinction between faithfulness to its input and substantive
correctness; do not automatically attribute the upstream fabrication
to this node.

COMPLIANCE - SECONDARY

Derive 3-5 measurable criteria from the node's own system prompt
ALONE. Use fewer when the prompt does not support three distinct
criteria.  Do not invent requirements to meet a quota.

Give each criterion:
- a unique snake_case name;
- a concise definition grounded in the node's instructions;
- a scope:  "final_output", "trajectory", or "both".

Examples of applicable requirement families include output format,
required content, tool-use restrictions, and mandated procedures.
Include a family only when supported by the node's actual prompt.
```

```
Use the SAME criteria and definitions for every instance in the batch.
Do not derive new compliance requirements from observed failures.

Score output requirements against the intended deliverable.  Score
procedural requirements against the relevant execution record.  Use
"unknown" when required procedural evidence is unavailable.

Keep compliance separate from correctness.  Following an instruction
that conflicts with user_goal may warrant high compliance and low
goal_contribution.

DOWNSTREAM UTILIZATION - DIAGNOSTIC

Compare the source output with the actual delivered input and the
consumer's observed actions and outputs from the SAME instance.

Consumer names, graph edges, final success, or textual similarity
alone do not establish uptake.  Require evidence that connects the
source contribution to downstream behavior.  When multiple sources
could explain the behavior and the record does not distinguish them,
use "unknown".

Utilization describes observed information flow, not whether that
flow was beneficial.  Incorrect information can be used extensively.
Correct information can appropriately be ignored.

For each handoff, assess:

uptake:
To what extent does the consumer observably use the task-relevant
content of this source output?

Score consideration and use, not unquestioning agreement.  A critic
that examines a candidate and rejects it with a specific justification
has used the candidate.  Mere acknowledgement without substantive
engagement is not full uptake.

sufficiency:
Did the delivered source contribution contain the information and
structure that this node was expected to supply for this consumer's
role?

Judge completeness, clarity, and usability relative to the
handoff contract and the consumer's available context, not factual
correctness.

Do not penalize the source because the consumer performs its own
assigned work.  A refiner making improvements or a verifier checking
claims is not necessarily evidence of insufficient input.

Do not infer insufficiency solely from consumer failure.  Identify
the specific missing or unusable contribution and why the source was
expected to provide it.

Runtime truncation is not automatically a sufficiency failure:  assess
whether necessary content was actually lost.

Assess once per instance:

final_reflection:
To what extent is the node's task-relevant contribution observably
retained, incorporated, or acted upon in the final pipeline result?
```

```
Look for substantive effects, not literal repetition.  A constraint
may be respected without being restated.  A critique may be reflected
through a correction rather than inclusion of the critique itself.

Do not require every intermediate idea to survive.  Appropriate
replacement or rejection can be a valid outcome.

If this node directly supplies the final result with no subsequent
processing, use "na" rather than awarding a trivial self-match.
Self-handoff behavior can still be evaluated separately.

If the final result or the connecting evidence is insufficient, use
"unknown".

When no applicable handoffs exist, return an empty handoff list and
mark handoff_coverage as "none".  Do not confuse this with missing
handoff records.

ATTRIBUTION - LIKELY ISSUE SOURCE

For every utilization score below 3, provide an attribution record.
Also provide one for any other consequential handoff defect you
identify.  A low score does not by itself prove that something needs
repair.

Use one of these labels:

node:
The source's own contribution has a demonstrated defect, such as a
missing required field, ambiguous instruction, or unusable format.

consumer:
The necessary contribution was demonstrably delivered and usable, but
the consumer mishandled it contrary to its role or requirements.

routing:
The record shows that an available source contribution was not
delivered as intended, or necessary content was lost during delivery.

correctly_discarded:
The consumer appropriately rejected substantively incorrect content.
Record the source correctness defect separately when the evidence
establishes it.

appropriate_nonuse:
Non-use or replacement was justified, for example because the
contribution was redundant, irrelevant to the selected valid approach,
or superseded by better-supported information.  This is not a defect
merely because uptake or final reflection is low.

mixed:
The evidence supports multiple interacting issue sources.  Identify
them explicitly rather than forcing a single mechanism.

unknown:
The record does not establish where the issue arose or whether the
observed non-use was appropriate.  State what evidence is missing.

Treat these labels as diagnostic hypotheses.  Do not claim that the
node was necessary, sufficient, or the unique cause of final success
or failure.
```

```
Consumer and routing issues must not automatically reduce the source
node's correctness or compliance.  The utilization score must still
describe the observed handoff.

Only reduce a local correctness or compliance score when a defect
relevant to that criterion is independently supported.

TRAJECTORY

For instances with more than one node iteration, assess:
- progress:  whether iterations resolve substantive issues or advance
the task rather than repeat or undo work;
- convergence:  whether the record shows a stable, adequate result;
- stopping:  whether termination was justified by task completion,
lack of useful remaining work, or an explicit constraint.

Report repeated unproductive revisions, regressions, or unresolved
failures when demonstrated.

Do not infer convergence merely because execution stopped.
Distinguish the node's choices from externally imposed timeouts or
missing logs.

Use "unknown" for unresolved judgments.  For a single-iteration
instance, return trajectory as null.

SCORING CALIBRATION

For correctness, compliance, sufficiency, and trajectory criteria:
1 = Fails the criterion in an essential way.
2 = Major deficiencies substantially limit fulfillment.
3 = Partially fulfills the criterion, with meaningful weaknesses.
4 = Fulfills the criterion well, with only minor limitations.
5 = Fully fulfills the criterion within the evaluable scope, supported
by concrete evidence.

For uptake and final_reflection:
1 = No observable use or retention of the relevant contribution.
2 = Limited use or retention; most relevant content is not
incorporated.
3 = Partial use or retention of the relevant contribution.
4 = Most relevant content is used or retained.
5 = Full observable use or retention of the relevant contribution.

The utilization scale measures extent, not desirability.  A high
utilization score can coexist with low correctness.

Apply scores to task-relevant content, not word overlap or output
length.  A short, fully adequate output can earn 5.

Score each instance against these definitions, not relative to other
instances in the batch.

Record both supporting evidence and demonstrated limitations before
settling on a rating.  Do not automatically choose a lower score
merely because uncertainty remains; use "unknown" when uncertainty
prevents a defensible rating.

Reserve critical_issues for demonstrated, consequential defects.
Minor imperfections can be described in assessment reasons.  Return
critical_issues as [] when no critical defect is established.  Missing
evidence belongs in the relevant "unknown" assessment unless the
```

```
record independently establishes a logging or interface defect.

CROSS-RUN SYNTHESIS

After assessing all instances:
- summarize recurring strengths, weaknesses, and evidence limitations
in 2-4 sentences;
- list systematic issues supported in at least two distinct instances;
- give each systematic issue its affected instance identifiers and the
count of distinct affected instances;
- separately summarize supported consumer-specific issues.

Do not count multiple iterations within one instance as multiple
affected instances.  Do not generalize beyond the supplied batch.

Do not collapse correctness, compliance, and utilization into one
overall reward or use diagnostic scores to declare a pipeline change
accepted.

OUTPUT CONTRACT

Return exactly ONE valid JSON object, with no surrounding prose,
Markdown fences, comments, duplicate keys, or trailing commas.

Use snake_case keys.

Each assessment record has:
- "score":  an integer 1-5, "na", or "unknown";
- "evidence":  an array of brief evidence references or excerpts;
- "reason":  a concise justification, including the reason for "na" or
the missing evidence for "unknown".

For handoff_coverage, use:
- "complete":  all applicable handoffs are documented;
- "partial":  only some applicable handoffs are documented;
- "missing":  applicable handoffs are known but their records are
absent;
- "none":  no applicable handoffs exist;
- "unknown":  the record does not establish handoff coverage.

Preserve supplied identifiers.  Use null for unavailable
source-iteration or consumer-invocation identifiers rather than
inventing them.

When trajectory is not null, it contains:
- "progress":  an assessment record;
- "converged":  true, false, or "unknown";
- "stopping":  an assessment record;
- "note":  a concise explanation.

Each attribution record contains:
- "criterion":  the path of the affected assessment;
- "source":  one allowed attribution label;
- "affected_nodes":  names supported by the record;
- "evidence":  supporting references or excerpts;
- "reason":  the explanation or unresolved ambiguity.

Each critical issue contains:
- "issue":  a concise description;
- "location":  the affected node, iteration, action, or handoff;
- "evidence":  supporting references or excerpts.
```

```
Each consumer issue contains:
- "consumer": the consumer name;
- "issue": a concise description;
- "instances": affected instance identifiers.

Each systematic issue contains:
- "issue": a concise description;
- "count": the number of distinct affected instances;
- "instances": those instance identifiers.

The following illustrates the JSON structure, not default ratings.
Replace descriptive strings, criteria, identifiers, and array entries
with the actual assessment. Do not invent a handoff or copy an
"unknown" rating merely because it appears in this illustration.

{
"node": "node_name",
"compliance_criteria": [
{
"name": "criterion_name",
"definition": "Requirement derived from the node's system prompt.",
"scope": "final_output"
}
],
"instances": [
{
"index": 0
, "run_id": "run_identifier",
"iterations": 1,
"correctness": {
"goal_contribution": {
"score": "unknown",
"evidence": [],
"reason": "Assessment-specific justification."
},
"input_faithfulness": {
"score": "unknown",
"evidence": [],
"reason": "Assessment-specific justification."
},
"input_completeness": {
"score": "unknown",
"evidence": [],
"reason": "Assessment-specific justification."
}
},
"compliance": {
"criterion_name": {
"score": "unknown",
"evidence": [],
"reason": "Assessment-specific justification."
}
},
"trajectory": null,
"utilization": {
"handoff_coverage": "unknown",
"handoffs": [
{
"source_iteration": null,
"consumer": "consumer_name",
"consumer_invocation": null,
"delivery_status": "unknown",
"uptake": {
```

```
"score": "unknown",
"evidence": [],
"reason": "Assessment-specific justification."
},
"sufficiency": {
"score": "unknown",
"evidence": [],
"reason": "Assessment-specific justification."
}
}
],
"final_reflection": {
"score": "unknown",
"evidence": [],
"reason": "Assessment-specific justification."
}
},
"attributions": [],
"critical_issues": [],
"reason": "Brief summary of this instance."
}
],
"consumer_issues": [],
"systematic_issues": [],
"synthesis": "Cross-instance summary with evidence limitations."
}

For delivery_status, preserve the supplied status using: "full",
"truncated", "absent", or "unknown".
```

User Prompt

```
NODE: {{NODE_NAME}}

NODE'S OWN SYSTEM PROMPT:
{{NODE_SYSTEM_PROMPT}}

INSTANCES IN THIS BATCH: {{NUMBER_OF_INSTANCES}}

=== INSTANCE {{INSTANCE_ID}} (run: {{RUN_ID}}) ===

user_goal: {{USER_GOAL}}

--- node trajectory ---
{{ITERATION_INDEXED_NODE_INPUTS_OUTPUTS_AND_RELEVANT_TOOL_ACTIONS}}

--- downstream consumer: {{CONSUMER_NAME}} ---
source_iteration: {{SOURCE_ITERATION}}
consumer_invocation: {{CONSUMER_INVOCATION_ID}}
consumer_role: {{CONSUMER_ROLE_OR_INPUT_OUTPUT_CONTRACT}}

delivery_status: {{FULL_TRUNCATED_ABSENT_UNKNOWN}}

actual delivered input:
{{DELIVERED_INPUT_OR_RELEVANT_EXCERPT}}

consumer actions and output:
{{CONSUMER_ACTIONS_AND_OUTPUT}}

evaluator_trace_visibility: {{COMPLETE_CAPPED_MISSING}}
```

```
[Repeat the downstream block for each relevant consumer invocation
within THIS instance.]

--- final pipeline result ---
{{FINAL_OUTPUT_AND_RELEVANT_ARTIFACT_OR_STATE_EVIDENCE}}

final_result_visibility:  {{COMPLETE_CAPPED_MISSING}}

=== END INSTANCE {{INSTANCE_ID}} ===

[Repeat the entire instance block for each run.]
```