DAGLOOP: ALIGNING METHOD DESIGN WITH EXPERIMENTAL FEEDBACK IN AUTOMATED AI RESEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Automated AI research agents aim to improve machine learning methods by iteratively proposing, implementing, and evaluating candidate methods. Existing work uses the outcome of each evaluation to determine how the next candidate should be revised for performance improvement. However, a candidate often combines multiple design decisions, and its overall outcome alone does not reveal which decisions produced their expected effects, whether their required conditions were satisfied, or which decisions should be retained or revised. Inspired by ablation studies in the machine learning community, we propose DAGLoop to test the expected effect of a design decision by comparing implementations with and without that decision while preserving the other decisions it requires. Specifically, DAGLoop translates each design decision's rationale into a testable expectation and pairs it with a targeted experiment, forming an experimental specification for assessing the expected effect under its required conditions. Using these specifications, we formulate ablation planning as the construction of a directed acyclic graph (DAG) from the candidate method's design: nodes associate decisions with tests, and edges encode dependencies. The DAG specifies which decisions each experiment must retain and supports joint tests when an expected effect requires several decisions to act together. Outcomes are recorded with the tested decisions, expectations, and conditions to form structured research memory. This memory guides the agent in retaining experimentally supported decisions, revising unsupported ones, and planning further experiments when the evidence is unresolved. Experiments on dozens of research tasks demonstrate substantial performance improvements achieved by DAGLoop across diverse executable research settings.

1 INTRODUCTION

Improving a machine learning method requires repeated decisions about what to change, what to test, and how to interpret the result. Recent research agents automate this process by proposing candidate methods, implementing them, running experiments, and using the outcomes to guide subsequent revisions. MLAgentBench studies agents that conduct machine learning experiments, while The AI Scientist extends automation to longer research workflows and agentic tree search (Huang et al., 2024; Lu et al., 2026; Yamada et al., 2025). AlphaEvolve uses executable evaluators to guide program and algorithm discovery (Novikov et al., 2025). We study the executable method-improvement setting: given a research task, a baseline, an experimental environment, and a finite budget, an agent seeks to improve performance under a fixed task evaluator.

Experimental feedback determines which candidates the agent explores next. Optimization by PROmpting (OPRO) generates candidates from scored solutions; FunSearch selects evaluated programs for further search (Yang et al., 2024; Romera-Paredes et al., 2024). Richer feedback includes MetaReflection's memory of past reflections and Trace's execution traces for parameter updates (Gupta et al., 2024; Cheng et al., 2024). Our focus is the relation between this feedback and the design decisions being revised. A candidate may combine a data transformation, a representation, an objective, and an optimization rule. Its overall performance measures their combined effect, but does not establish which decisions produced their expected effects or whether their required

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.

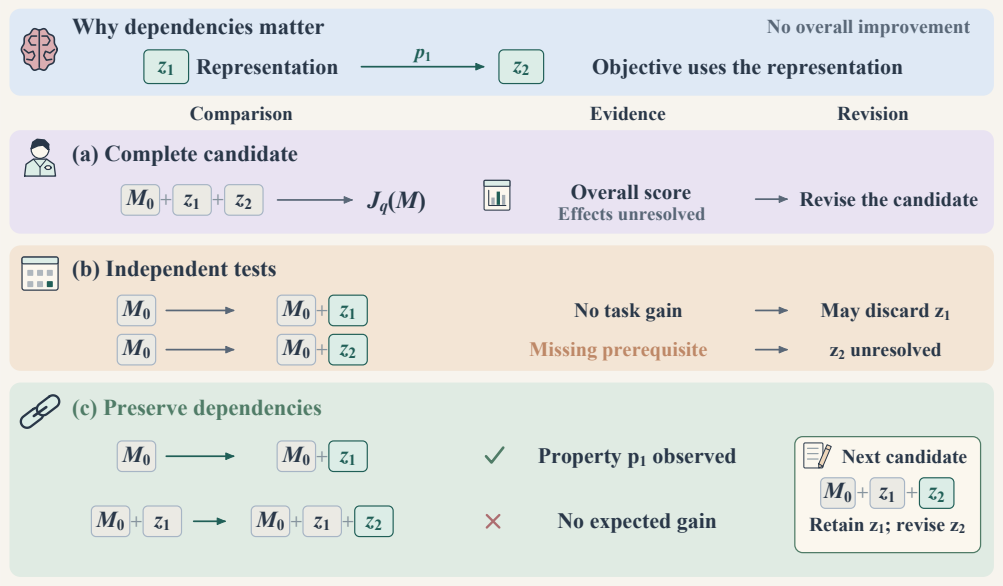

Figure 1: Why dependencies matter. Candidate-level feedback combines decision effects, while independent tests can remove prerequisites. DAGLoop preserves prerequisites in the control. Here M_0 is the baseline, J_q the task metric, and p_1 a property supplied by z_1.

conditions held. The evaluator scores the complete candidate, whereas revision must determine which constituent decisions to retain, revise, or test further.

Ablation studies examine decisions by comparing implementations with and without them; recent benchmarks study whether language models can plan informative ablations (Zhao et al., 2025; Abramovich & Chechik, 2026). However, testing each decision independently can remove the conditions needed for its effect. Suppose z_1 constructs a representation and z_2 introduces an objective that uses it. The representation may provide a necessary intermediate property without improving the task metric alone, while the objective may be meaningless without it. Rejecting z_1 for lacking standalone gain prevents a meaningful test of z_2; evaluating only their composition leaves each decision's contribution unresolved. Decision-level evidence therefore requires experiments that preserve the prerequisites of the decision being tested (Figure 1).

We present **DAGLoop**, a general-purpose research system that addresses executable method-improvement problems through proposal, implementation, experimentation, and revision. It constructs targeted experiments from a candidate method's design. For each decision, it translates the rationale into a testable expectation: the effect the decision should produce and its required conditions. A targeted experiment specifies what to compare and measure, forming an experimental specification that connects design reasoning to evidence. These specifications define a DAG, whose nodes associate decisions with tests and whose edges encode their dependencies. Each experiment compares a control implementation containing the prerequisites with a treatment implementation that additionally applies the decision: $z_1 + z_2$ against z_1 in the example. A decision that provides a prerequisite, such as z_1, is assessed through the intermediate property it is intended to provide. Effects requiring several decisions are tested jointly, retaining their external prerequisites and attributing experimental support to the group of decisions tested together.

After checking execution validity, DAGLoop records each outcome alongside the decision, expected effect, required conditions, and saved control implementation. This structured research memory guides the agent to retain supported decisions, revise unsupported ones, and plan further experiments when evidence is unresolved. Failed implementations provide debugging information. Revised decisions are assembled into a subsequent candidate and evaluated with the same official evaluator. Targeted experiments guide revision; complete-candidate evaluation measures improvement.

We evaluate DAGLoop on 36 AutoLab tasks spanning CPU optimization, CUDA kernels, program synthesis, and model development (Xu et al., 2026), comparing with Terminus-2 using the same language model. Our contributions are threefold:

- We identify a mismatch between evaluating complete candidates and revising individual decisions, motivating experiments that retain prerequisites for decision-level evidence.
- We propose DAGLoop, a research system that translates design rationales into experimental specifications, constructs a dependency DAG to plan targeted ablations, and uses structured research memory to guide evidence-based revision and further experimentation.
- Experiments on 36 AutoLab tasks demonstrate method improvement across diverse executable problems. DAGLoop achieves a mean score of 0.6113, compared with 0.5831 for Terminus-2 using the same language model, with higher means on both CPU and GPU tasks.

2 Related Work

Automated AI Research. Recent work automates method proposal, implementation, experimentation, and revision. MLAgentBench studies language agents that conduct machine-learning experiments; The AI Scientist, its extension using agentic tree search, and Agent Laboratory address longer research workflows (Huang et al., 2024; Lu et al., 2024; Yamada et al., 2025; Lu et al., 2026; Schmidgall et al., 2025). AlphaEvolve uses executable evaluators to guide program and algorithm discovery (Novikov et al., 2025).

Executable benchmarks make this research loop measurable. MLE-bench, MLGym, and RE-Bench evaluate agents on executable tasks; MLR-Bench assesses open-ended research (Chan et al., 2025; Nathani et al., 2025; Wijk et al., 2025; Chen et al., 2025a). ScienceAgentBench evaluates scientific code using execution results and cost (Chen et al., 2025b). ResearchGym, AutoLab, and AutoResearchEval examine longer research trajectories, artifact improvement, and failures linking experiments to claims (Garikaparthi et al., 2026; Xu et al., 2026; Fei et al., 2026). Our work studies the evidence used to revise design decisions within these executable research settings.

Evaluator Feedback and Research Memory. OPRO conditions candidate generation on score histories; FunSearch selects evaluated programs for further search; Automated Design of Agentic Systems (ADAS) and AFlow optimize agent designs and workflows (Yang et al., 2024; Romera-Paredes et al., 2024; Hu et al., 2025; Zhang et al., 2025). MetaReflection uses past reflections; TextGrad and Trace optimize variables using textual and execution feedback (Gupta et al., 2024; Yuksekgonul et al., 2025; Cheng et al., 2024). The AI Scientist maintains an experimental journal, Meta-Harness exposes earlier candidates' code, scores, and traces, and ResearchLoop organizes questions, evidence, and claims as durable research state (Lu et al., 2026; Lee et al., 2026; Xia & Wang, 2026). AutoResearchEval examines links between implementations, observations, and claims (Fei et al., 2026). Our memory associates each outcome with the tested decision, expected effect, and prerequisite context, so revision can use evidence at the level of the decision being changed.

Ablation Planning and Experimental Attribution. AbGen and AblationBench evaluate whether language models can design informative ablations, and The AI Scientist includes ablations in its experiment schedule (Zhao et al., 2025; Abramovich & Chechik, 2026; Lu et al., 2026). Software composition also requires explicit component dependencies. Cordis represents these dependencies, motivating our description of the properties that one decision requires from another (Shi et al., 2026). DAGLoop uses these dependencies to construct the control implementation for each targeted experiment during revision. It distinguishes task improvement from a property required by another decision and uses joint tests when an expected effect requires several decisions. The evidence guides decision retention, revision, and further testing.

3 Method

DAGLoop organizes method proposal, implementation, experimentation, and revision into a research loop. The system turns a candidate's design into experimental specifications, uses their dependencies to construct targeted comparisons, and records the outcomes for revision (Figure 2). The assembled candidate is evaluated by the official task evaluator.

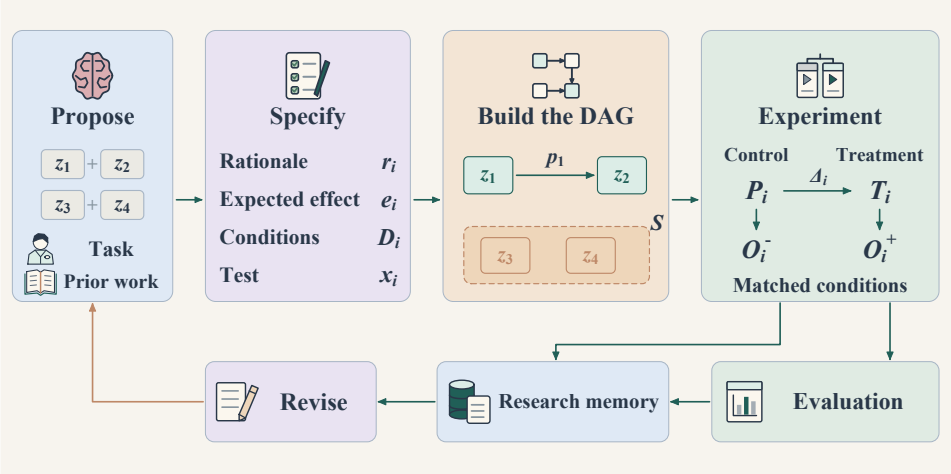

Figure 2: DAGLoop research loop. Each DAG node represents a decision or jointly tested group; S denotes such a group. Control P_i preserves prerequisites; treatment T_i adds the change Δ_i. Their observations O_i^- and O_i^+ enter research memory and guide revision. Complete candidates are evaluated separately by the task evaluator, with feedback returned for revision.

3.1 Task Setup and Experimental Specifications

Starting from the user's research question, DAGLoop clarifies objectives and constraints with the user, retrieves relevant literature, and proposes a method. The user confirms the research direction and method. With an executable baseline, evaluator, environment, and budget established, the system proceeds to implementation, experimentation, and revision.

An executable research task is

$$\mathcal{T} = (q, M_0, \mathcal{E}, J_q, B), \tag{1}$$

where q describes the task, M_0 is an executable baseline, $\mathcal{E}$ is the experimental environment, and B is the research budget. The official evaluator returns a raw (native) task metric $m_q(M)$, such as accuracy or execution time, and a task score $R_q(M)$. We define J_q so that larger values indicate better performance: $J_q(M) = m_q(M)$ for maximization tasks and $J_q(M) = -m_q(M)$ for minimization tasks. The score R_q follows the official reward rule, including its thresholds and clipping.

Let τ be the research trajectory, including proposals, implementations, experiments, and revisions, and let $\mathcal{C}(\tau)$ contain the complete executable candidates it evaluates. We seek

$$\widehat{M}(\tau) \in \arg \max_{\substack{M \in \mathcal{C}(\tau) \cup \{M_0\} \\ M \text{ passes the task checks}}} J_q(M), \qquad \mathrm{Cost}_{\mathcal{E}}(\tau) \le B. \tag{2}$$

Task checks enforce correctness and task constraints. Cost includes planning, implementation, targeted tests, repair, repeated measurements, and final evaluation. Improvement means $J_q(\widehat{M}) > J_q(M_0)$. The baseline remains available when no evaluated candidate improves it.

A candidate method M consists of design decisions $Z = \{z_i\}_{i=1}^n$. The complete-candidate metric $J_q(M)$ evaluates their combined performance, whereas revision requires deciding which z_i to retain or change. To test the effect expected from a particular decision, we associate it with an experimental specification linking its rationale, expected effect, conditions, and test:

$$s_i = (r_i, e_i, D_i, x_i), \qquad x_i = (\Delta_i, \rho_i, o_i, v_i), \tag{3}$$

where r_i is the rationale, e_i the expected effect, and D_i the required conditions. The experiment x_i specifies the implementation change Δ_i, execution procedure ρ_i, quantity to measure o_i, and validation rule v_i. The specification defines the measurement and criterion before execution; the experiment report supplies the observed value. A decision can provide an intermediate property p_i required by another decision, with a corresponding property check.

Each node in the experimental DAG represents an executable component: one decision or a group of decisions tested together. Each component has an implementation and a specification. Tests assess either improvement in the task metric or an intermediate property required by another decision. We refer to these as *performance tests* and *property checks*, respectively. A checkpoint stores the implementation and artifacts needed to reconstruct a control or candidate.

From a rationale to a test. In the representation–objective example, the rationale for z_1 identifies a property p_1 needed by z_2. Its specification makes that property the expected effect and selects an observable that checks whether the representation supplies it. The specification for z_2 instead declares a task-metric improvement, with p_1 included in its required conditions. These declarations determine both what is measured and which implementation is a meaningful control. A successful property check establishes the context for testing z_2; the downstream experiment then assesses its own declared effect. The planner records these roles before execution so that a decision that supplies a prerequisite is assessed by its intended property even when its standalone task metric does not improve in isolation under the same evaluation protocol.

The planner uses a language model to propose decisions, dependencies, and tests from the method, task, literature summary, and baseline evidence. A review agent (the critic) checks whether the tests assess the expected effects under the required conditions. The planner revises the proposal using this feedback. The plan is accepted after review finds no outstanding issues and graph validation passes; Section 3.2 describes the graph and the prerequisite checks used to validate it.

When an effect requires several decisions to act together, the planner groups them into one component with a shared specification and test. The plan records the original decisions and their external dependencies. The result is attributed to the group.

3.2 Ablation Planning

Ablation analysis compares algorithm configurations to assess particular changes (Abramovich & Chechik, 2026). Fix the baseline, implementations, and evaluation conditions, and let $f(C)$ denote J_q for the implementation containing a feasible set C of decision indices. For C and $C \cup \{i\}$ satisfying the declared dependencies, the marginal effect of adding decision i is

$$\delta_{i|C} = f(C \cup \{i\}) - f(C), \qquad i \notin C. \tag{4}$$

This difference measures the marginal contribution of decision i to configuration C (Olsen et al., 2024). For z_2 requiring z_1, setting $C = \{1\}$ recovers $\delta_{2|1} = f(\{1,2\}) - f(\{1\})$. Baseline and complete-candidate values do not determine its sign because they leave $f(\{1\})$ unobserved (Proposition 1). Testing z_1 supplies this control measurement; the property check assesses whether this implementation provides the declared intermediate property.

Let V index the components. The conditions D_i describe what component i requires; d_i^0 identifies the other components whose outputs or properties it directly needs. An edge (j, i) denotes such a dependency. Other conditions, such as the data split and measurement settings, are specified in the experimental protocol. The set K_i contains all ancestors of i, including indirect prerequisites:

$$\begin{aligned} A_0 &= \{(j,i) : j \in d_i^0\}, \qquad G_0 = (V, A_0), \\ K_i &= \bigcup_{j \in d_i^0} \big(\{j\} \cup K_j\big). \end{aligned} \tag{5}$$

Graph validation checks identifiers, acyclicity, and agreement between K_i and the declared prerequisites (Appendix A). The control P_i retains prerequisites; treatment T_i adds the change:

$$P_i = \mathrm{Build}(M_0, K_i), \qquad T_i = P_i \oplus \Delta_i. \tag{6}$$

Build combines saved implementations of the required components with the baseline. The operator $\oplus$ applies the tested component's implementation change to this control. Implementation review checks that the resulting code follows the specification. Appendix A details this merge.

For $z_1 \to z_2$, the test compares $z_1 + z_2$ with z_1, preserving the representation required by the objective. Testing z_c in $z_a \to z_b \to z_c$ retains both z_a and z_b. Thus, the control contains every required component together with its own prerequisites.

For a component u representing a group S, the experiment applies the decisions together to a control containing their external prerequisites. The shared specification defines the expected effect of this combined change. The result is attributed to the group, and the record identifies all participating decisions without assigning separate gains to individual members. The revised graph must remain acyclic. Appendix A gives the construction algorithm and examples; Appendix D proves that baseline and complete-candidate evaluations alone cannot determine the improvement in equation (4), and that K_i is the smallest set satisfying the declared dependencies.

3.3 Executing and Assessing Experiments

The procedure ρ_i specifies how to run the experiment; the protocol ω_i specifies the evaluation conditions and measurement settings shared by control and treatment:

$$O_i^- = \mathrm{Run}(\mathcal{E}, P_i; \rho_i, \omega_i), \qquad O_i^+ = \mathrm{Run}(\mathcal{E}, T_i; \rho_i, \omega_i). \tag{7}$$

The records O_i^- and O_i^+ link observations to their implementations and execution attempts. Both runs use the validation rule specified in s_i. Matching conditions controls data and execution differences; training randomness also affects performance comparisons (Bosma et al., 2024).

For valid executions with finite measurements and correctly matched reports, let m_i^- and m_i^+ be the control and treatment metrics, and let $d_q \in \{-1, 1\}$ denote minimization or maximization. The numerical test criterion is expressed by the predeclared predicate $\phi_i(y) = \mathbf{1}[y \bowtie_i \theta_i]$, with $\bowtie_i \in \{<, \leq, =, \geq, >\}$ and threshold θ_i. The resulting criterion separates performance tests from property checks according to the declared experimental objective:

$$\begin{aligned} \delta_i &= d_q(m_i^+ - m_i^-), \\ g_i &= \begin{cases} \mathbf{1}[\delta_i > \max(0, \epsilon_i)]\, \phi_i(y_i^{\mathrm{test}}), & \text{performance test,} \\ \phi_i(a_i^{\mathrm{obs}}), & \text{property check.} \end{cases} \end{aligned} \tag{8}$$

A performance test requires improvement beyond zero and the protocol threshold ϵ_i, and applies ϕ_i to $y_i^{\mathrm{test}} \in \{\delta_i, m_i^+\}$ as specified before execution. A property check applies ϕ_i to the reported intermediate observable a_i^{obs}. An observable such as the number of failed property checks can summarize several assertions. Appendix A gives the full numerical and report-matching checks.

Execution validity and effect assessment are separate. A checkpoint is *qualified* for retention after successful execution, implementation review, task checks, and a passed test, with no pending plan revision. A component is *accepted* when its search ends with a retained qualified checkpoint. Failed execution triggers repair; a valid test that fails its criterion informs revision of the expected effect or design. See Appendices A and C for details.

3.4 Research Memory and Revision

For each component, DAGLoop records the specification version, prerequisites, checkpoints, code changes, observations, validation results, and execution artifacts. Each observation remains associated with the decision, expected effect, conditions, and specification version tested. These records supply the evidence for subsequent implementation and method revision.

At research step t, let $\mathcal{L}_t(j)$ record component j's history. For active component i, the agent receives

$$\mathcal{H}_t(i) = (\{\mathcal{L}_t(j) : j \in K_i \cup \{i\}\}, \quad \{\mathrm{Summary}(\mathcal{L}_t(j)) : j \notin K_i \cup \{i\}\}). \tag{9}$$

Detailed histories of the active component and its ancestors let the agent inspect how the control was built, which prerequisite checks passed, and how the current attempt differs from earlier ones. Summaries of other branches supply context for revising the method without including their full execution traces in each component's input. Failed attempts remain available for debugging; method revision receives the accumulated research memory.

For example, when a valid test of z_2 does not support its expected gain, the record identifies the control containing z_1. Revision can reconsider the objective while retaining the supported representation. If revision changes z_1, the earlier result for z_2 remains a record of the previous experiment. The revised representation defines a new control for the objective; testing it again determines whether the expected effect persists after the prerequisite changes.

DAGLoop reuses a saved implementation only when its specification, prerequisite implementations, and recorded execution context remain compatible. Otherwise, the earlier result stays in memory for reference. Appendix C gives the equality and checkpoint checks used for reuse.

Search processes an active component first; a pending component becomes ready after its prerequisites are accepted. The agent drafts the first implementation, debugs execution errors or code that does not follow the specification, and reassesses the expectation when a valid test fails its criterion. Otherwise, it improves the implementation. Each operation uses research memory. Replanning can revise the decomposition or group decisions within the revision budget. When no component is ready, the system reports which prerequisites remain unresolved.

Component search retains the best qualified checkpoint. Performance tests rank checkpoints by the task metric; property checks use the specified comparison rule, with task performance breaking ties. Search continues to the attempt limit, retaining the best checkpoint found (Appendix C).

3.5 Complete-Candidate Evaluation

Once all components are accepted, DAGLoop assembles their checkpoints and runs the official evaluator. Targeted tests assess effects in the prerequisite context, whereas the assembled candidate also includes decisions from other branches. These combinations can change task performance. Complete-candidate evaluation checks whether the supported changes improve the full method, and its result enters research memory to guide the next revision. Only complete candidates that pass the task checks can replace the current best:

$$\widehat{M}_{t+1} \in \arg \max_{M \in \{\widehat{M}_t\} \cup \mathcal{C}_t^{\text{valid}}} J_q(M), \tag{10}$$

Here $\mathcal{C}_t^{\text{valid}}$ contains newly evaluated complete candidates. Component measurements guide revision through memory. Algorithm 1 summarizes the loop; Appendices A and C detail the procedure and resource limits used during execution.

Algorithm 1 DAGLoop: targeted experiments and revision

Input: Task, baseline M_0, proposed candidate M, budget B

Initialize research memory $\mathcal{L}$ and best candidate $\widehat{M} \leftarrow M_0$
while budget and search limits permit **do**
  Construct or revise specifications and the dependency DAG using $(M, \mathcal{L})$
  for each ready component i in topological order **do**
    Search component i: build P_i, implement and test T_i under the declared protocol
    Record validity and effect evidence in $\mathcal{L}$; debug or revise using this memory
    Retain the best qualified checkpoint when component search ends
  end for
  If all components are accepted, assemble and evaluate M^+; update $\widehat{M}$
  Revise M from $\mathcal{L}$ if further search is permitted; otherwise stop
end while
return $\widehat{M}, \mathcal{L}$

4 Experiments

We evaluate DAGLoop on diverse executable research tasks. We use AutoLab as the main benchmark and compare with locally run Terminus-2 using the same language model. We also evaluate on FML-Bench; its results are reported in Appendix G.

4.1 Benchmark and Task Protocol

We use all 36 official AutoLab tasks: 25 CPU tasks and 11 GPU tasks covering CPU optimization, CUDA kernels, program synthesis, and model development (Xu et al., 2026). Each task provides a

baseline repository, evaluator, reward rule, correctness checks, and protected task files. DAGLoop reproduces the baseline locally, proposes a candidate, constructs the dependency DAG, runs component experiments, and evaluates candidate checkpoints.

Both DAGLoop and Terminus-2 use `glm-5.2`. All AutoLab GPU experiments for both agents run on NVIDIA H800 GPUs. The comparison uses the official Terminus-2 agent in Harbor 0.3.0, configured with `openai/glm-5.2` and the original task verifiers (Harbor Framework Team, 2026). Both agents run to completion without agent wall-clock truncation; task-verifier timeouts remain unchanged for both agents throughout evaluation.

For each task, we report the score from one DAGLoop run and the arithmetic mean of two local Terminus-2 runs. Section 3.5 describes how the system selects its final complete candidate. Appendix F reports all task scores for completeness.

Higher scores are better. We report task scores and means for the full suite, CPU tasks, and GPU tasks. Reward thresholds and clipping can produce ties despite different native metrics. The local native-metric baselines are listed in their original units; the paired comparison uses task scores.

4.2 Overall and Per-Task Results

Table 1: Mean task scores with `glm-5.2`.

Split	Tasks	**DAGLoop**	Terminus-2
All	36	0.6113	0.5831
CPU	25	0.6344	0.6247
GPU	11	0.5588	0.4885

DAGLoop has higher mean scores than Terminus-2 on the full suite and in both resource groups (Table 1). Each mean includes every task in its split. Figure 3 shows paired task scores and resource-level means. Table 2 reports all 36 task scores; Appendix F gives native metrics and local baselines.

The largest DAGLoop advantages occur in online language-model serving (1.0000 versus 0.0006) and game-playing program minimization (0.9491 versus 0.4740). Terminus-2 leads on instruction-following data selection (0.8581 versus 0.3774) and sorted string table (SSTable) compaction (0.3479 versus 0.0696). Four tasks tie at 1.0000, including BM25 and multilingual optical character recognition in the full AutoLab benchmark suite.

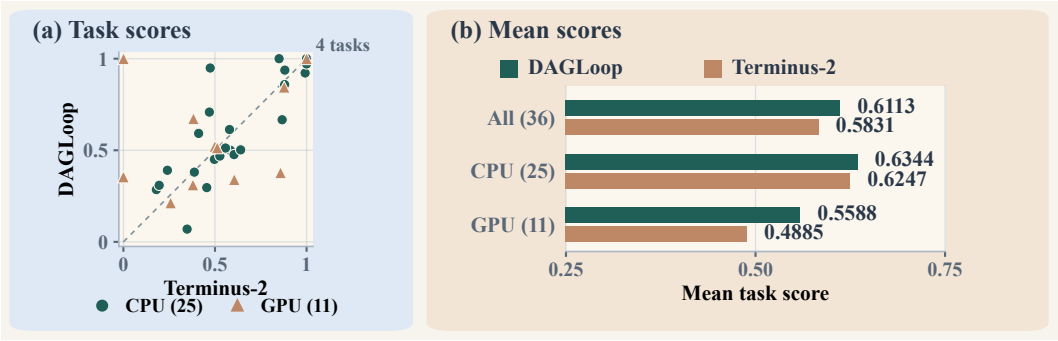

Figure 3: DAGLoop and Terminus-2 with the same `glm-5.2` model. (a) Paired task scores; points above the diagonal favor DAGLoop. (b) Mean scores for the full suite and CPU/GPU splits; the horizontal axis spans 0.25–0.75. All task scores appear in Table 2.

4.3 CPU and GPU Results

The CPU results cover sorting, comparator construction, and spatial indexing. DAGLoop scores 0.5921 on radix sort and 0.3087 on Z-order scans, compared with 0.4108 and 0.1953 for Terminus-2. On AES-128 CTR and concurrent key-value storage, the corresponding task scores are 0.6132 versus 0.5794 and 0.4764 versus 0.6040, respectively.

Table 2: AutoLab task scores. T2: local Terminus-2 mean. Higher scores and ties are bold.

Task	**DAGLoop**	T2	Task	**DAGLoop**	T2
Adaptive compression	0.4502	**0.4968**	Levenshtein distance	0.3805	**0.3874**
Adversarial splay	0.5000	**0.5841**	LLM online serving	**1.0000**	0.0006
AES-128 CTR	**0.6132**	0.5794	Moving MNIST	0.2126	**0.2579**
Agent tool routing	**0.3908**	0.2403	MSM Pippenger	**0.6725**	0.3824
BM25 retrieval	**1.0000**	**1.0000**	Multilingual OCR	**1.0000**	**1.0000**
BVH raytracing	0.2961	**0.4551**	NTT butterfly	**0.5133**	0.5125
Concurrent KV WAL	0.4764	**0.6040**	Radix sort	**0.5921**	0.4108
IFEval data selection	0.3774	**0.8581**	Regex engine	0.8587	**0.8803**
Sorting discovery	**1.0000**	0.8500	ResNet bit flip	**0.9375**	0.8813
FFT	0.4684	**0.5266**	Safety router	0.9218	**0.9921**
FlashAttention	**0.7084**	0.4695	Scaling law	0.3402	**0.6054**
FLUX.2 LoRA	**0.3545**	0.0000	SHA-256	**0.2853**	0.1799
Fredkin network	0.6667	**0.8667**	Game-player size	**0.9491**	0.4740
Gaussian blur	0.5122	**0.5574**	SSTable compaction	0.0696	**0.3479**
Visual reasoning	0.8444	**0.8778**	Stack-machine golf	**1.0000**	**1.0000**
Hash join	0.5024	**0.6402**	Toy ISA	**1.0000**	**1.0000**
Huffman CUDA	0.3109	**0.3798**	VLIW scheduling	0.9712	**1.0000**
ICP CUDA	**0.5213**	0.4994	Z-order scan	**0.3087**	0.1953

GPU tasks cover kernel latency and model quality. DAGLoop's multi-scalar multiplication kernel takes 27.5941 ms and scores 0.6725 versus 0.3824 for Terminus-2. Its number-theoretic transform (NTT) butterfly kernel takes 1.136992 ms and scores 0.5133 versus 0.5125. For visual reasoning, an accuracy of 0.58 corresponds to a score of 0.8444 versus 0.8778.

5 Limitations

Decision-level evidence assesses the declared effect under the tested conditions. Its reliability depends on the decomposition, prerequisite graph, observable, and the experiment's statistical power. It does not identify a unique causal explanation, and joint tests attribute results to the merged component when member-level isolation is not meaningful. The task-level comparison measures performance reached during research. The contributions of tests that retain prerequisites and research memory remain unmeasured; this requires matched-budget comparisons over repeated runs.

DAGLoop addresses candidate-method improvement under executable task constraints. Evidence reuse depends on recorded context: changes to data, hardware, evaluation, or upstream decisions can require new experiments. Repeated optimization against a fixed evaluator can overfit task artifacts. Broader scientific generalization and revision of the research agent itself require separate evaluation.

6 Conclusion

We present DAGLoop as a general-purpose research system for executable method improvement. It connects candidate-method revision to targeted experiments on its constituent design decisions. Experimental specifications link each decision's rationale and expected effect to tests that preserve its prerequisites. Structured research memory carries the outcomes into subsequent revisions, while complete-candidate evaluation measures task performance. Preserving prerequisites lets the agent assess intermediate properties and downstream effects in their intended context. Joint experiments assess effects that require several decisions to act together, while the accumulated records guide further tests when the evidence remains unresolved. On 36 AutoLab tasks, DAGLoop obtains a mean task score of 0.6113, compared with 0.5831 for Terminus-2 using the same language model. DAGLoop scores higher on 14 tasks, ties on four, and scores lower on 18. The research output records which expectations were supported, under which prerequisites, and which remain unresolved.

Ethics Statement

Automated research consumes compute and can amplify errors in experimental interpretation. DAGLoop retains evaluator requirements, experiment records, and the context of each result. The authors remain responsible for resource use, interpretation, and reported claims.

Reproducibility Statement

Section 3 describes control construction, validation, memory, and revision; Appendices A and C provide the full procedure and evidence-reuse rules.

AI Use Statement

Generative AI tools were used for literature search and screening, method and experiment planning, code development and research execution, assistance with mathematical proofs, data analysis and figure preparation, and manuscript drafting, translation, and language editing. The authors review the text, citations, and experimental claims and take responsibility for the final work.

References

Talor Abramovich and Gal Chechik. AblationBench: Evaluating automated planning of ablations in empirical AI research, 2026. URL https://arxiv.org/abs/2507.08038.

AutoLab Authors. AutoLab: Official benchmark repository, 2026. URL https://github.com/autolabhq/autolab/tree/4127da3dde8449be61a1cf9859473b9fbbd51751. Task definitions at commit 4127da3dde8449be61a1cf9859473b9fbbd51751.

Joeran Sander Bosma, Dré Peeters, Natália Alves, Anindo Saha, Zaigham Saghir, Colin Jacobs, and Henkjan Huisman. Reproducibility of training deep learning models for medical image analysis. In *Medical Imaging with Deep Learning*, volume 227, pp. 1269–1287. PMLR, 2024. URL https://proceedings.mlr.press/v227/bosma24a.html.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, volume 2025, pp. 50466–50494, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/7e3767db483c942b883eb4f8cfb74e31-Paper-Conference.pdf.

Hui Chen, Miao Xiong, Yujie Lu, Wei Han, Ailin Deng, Yufei He, Jiaying Wu, Yibo Li, Yue Liu, and Bryan Hooi. MLR-Bench: Evaluating AI agents on open-ended machine learning research. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025a. doi: 10.52202/085713-3945. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/ab8dd000d6f87f40061a73f8bca7fae4-Paper-Datasets_and_Benchmarks_Track.pdf.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. ScienceAgentBench: Toward rigorous assessment of language agents for data-driven scientific discovery. In *International Conference on Learning Representations*, volume 2025, pp. 96934–96990, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/f12b4df26344f3be803c06b555252efe-Paper-Conference.pdf.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 71596–71642. Curran Associates, Inc., 2024. doi: 10.52202/079017-2287. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/83ba7056bce2c3c3c27e17397cf3e1f0-Paper-Conference.pdf.

Yanlin Fei, Nazhou Liu, Xinmiao Yu, Shaolong Chen, Lei Li, Rahul Thapa, Madalina Ciobanu, Navan Preet Singh, Qingqing Mao, and Ritankar Das. How do agents fail on AutoResearch: End-to-end diagnostic evaluation on 100 real-world frontier research tasks, 2026. URL https://arxiv.org/abs/2608.14905v3.

Aniketh Garikaparthi, Manasi Patwardhan, and Arman Cohan. ResearchGym: Evaluating language model agents on real-world AI research, 2026. URL https://arxiv.org/abs/2602.15112.

Priyanshu Gupta, Shashank Kirtania, Ananya Singha, Sumit Gulwani, Arjun Radhakrishna, Gustavo Soares, and Sherry Shi. MetaReflection: Learning instructions for language agents using past reflections. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 8369–8385. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.477. URL https://aclanthology.org/2024.emnlp-main.477/.

Harbor Framework Team. Terminus-2 in harbor (version 0.3.0), 2026. URL https://github.com/harbor-framework/harbor/tree/1d97c20a1729f068a2bb95af6100f0f818ef0595. Official agent implementation, release v0.3.0; accessed September 26, 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. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/36b7acf6f6010652b3f2a433774a66fe-Paper-Conference.pdf.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. MLAgentBench: Evaluating language agents on machine learning experimentation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 20271–20309. PMLR, 2024. URL https://proceedings.mlr.press/v235/huang24y.html.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist: Towards fully automated open-ended scientific discovery, 2024. URL https://arxiv.org/abs/2408.06292.

Chris Lu, Cong Lu, Robert Tjarko Lange, Yutaro Yamada, Shengran Hu, Jakob Foerster, David Ha, and Jeff Clune. Towards end-to-end automation of AI research. *Nature*, 651(8107): 914–919, 2026. doi: 10.1038/s41586-026-10265-5. URL https://doi.org/10.1038/s41586-026-10265-5.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, Dieuwke Hupkes, Ricardo Silveira Cabral, Tatiana Shavrina, Jakob Foerster, Yoram Bachrach, William Yang Wang, and Roberta Raileanu. MLGym: A new framework and benchmark for advancing AI research agents, 2025. URL https://arxiv.org/abs/2502.14499.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Lars Henry Berge Olsen, Ingrid Kristine Glad, Martin Jullum, and Kjersti Aas. A comparative study of methods for estimating model-agnostic shapley value explanations. *Data Mining and Knowledge Discovery*, 38(4):1782–1829, 2024. doi: 10.1007/s10618-024-01016-z. URL https://doi.org/10.1007/s10618-024-01016-z.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang,

Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using LLM agents as research assistants. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 5977–6043. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.320. URL https://aclanthology.org/2025.findings-emnlp.320/.

Yifan Shi, Wei Zhang, and Tianyi Cui. A programming paradigm for spatiotemporal composability, 2026. URL https://arxiv.org/abs/2608.25512.

Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Josh Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Holden Karnofsky, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. RE-Bench: Evaluating frontier AI r&d capabilities of language model agents against human experts, 2025. URL https://arxiv.org/abs/2411.15114.

Yihan Xia and Taotao Wang. ResearchLoop: An evidence-gated control plane for AI-assisted research, 2026. URL https://arxiv.org/abs/2605.28282.

Zhangchen Xu, Junda Chen, Yue Huang, Dongfu Jiang, Jiefeng Chen, Hang Hua, Zijian Wu, Zheyuan Liu, Zexue He, Lichi Li, Shizhe Diao, Jiaxin Pei, Jinsung Yoon, Hao Zhang, Mengdi Wang, Radha Poovendran, Misha Sra, Alex Pentland, and Zichen Chen. AutoLab: Can frontier models solve long-horizon auto research and engineering tasks?, 2026. URL https://arxiv.org/abs/2606.05080.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist-v2: Workshop-level automated scientific discovery via agentic tree search, 2025. URL https://arxiv.org/abs/2504.08066.

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, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/3339f19c5fcee3ad74502947a32be9e6-Paper-Conference.pdf.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025. doi: 10.1038/s41586-025-08661-4. URL https://doi.org/10.1038/s41586-025-08661-4.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, XiongHui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5492ecbce4439401798dcd2c90be94cd-Paper-Conference.pdf.

Yilun Zhao, Weiyuan Chen, Zhijian Xu, Manasi Patwardhan, Chengye Wang, Yixin Liu, Lovekesh Vig, and Arman Cohan. AbGen: Evaluating large language models in ablation study design and evaluation for scientific research. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12479–12491. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.611. URL https://aclanthology.org/2025.acl-long.611/.

Qiran Zou, Hou Hei Lam, Wenhao Zhao, Tingting Chen, Yiming Tang, Samson Yu, Yingtao Zhu, Srinivas Anumasa, Zufeng Zhang, Tianyi Zhang, Chang Liu, Zhengyao Jiang, Anirudh Goyal, and Dianbo Liu. Fml-bench: A controlled study of AI research agent strategies from the perspective of search dynamics, 2026. URL https://arxiv.org/abs/2605.17373.

A Detailed Experimental Procedure

The executable task and budgeted objective are defined in equations (1) and (2). The procedures below implement its targeted experiments and method revision.

A.1 Graph Construction and Comparison Configurations

Let N convert component names to a consistent identifier format (Appendix C). The declared dependencies d_i^0 and ancestor sets K_i are defined in equation (5).

The constructor rejects duplicate identifiers, unknown or self-referential dependencies, and cycles. Kahn's topological sort, using a first-in, first-out (FIFO) queue for eligible nodes, gives the order π in which the ancestor sets are computed, preserving the input traversal order when nodes become eligible together. The plan also declares a_i, the set of components that must be accepted before testing component i. The constructor removes redundant direct edges, retaining the parents d_i, and requires a_i to equal all ancestors:

$$\begin{aligned} d_i &= d_i^0 \setminus \{j \in d_i^0 : \exists k \in d_i^0 \setminus \{j\},\ j \in K_k\}, \\ a_i &= K_i, \qquad A = \{(j,i) : j \in d_i\}. \end{aligned} \tag{11}$$

The equality is a validation requirement: a missing ancestor or an unrelated component causes rejection. Semantic adequacy of the proposed dependencies and tests is assessed by the critic.

Algorithm 2 Construction of the dependency DAG

Input: Declared components $\{(i, d_i^0, a_i, s_i)\}_{i=1}^n$
Normalize identifiers with N; reject duplicates and invalid references
$A_0 \leftarrow \{(j,i) : j \in d_i^0\}$; $\pi \leftarrow \text{FIFO-Kahn}(V, A_0)$
if $|\pi| \neq |V|$ **then**
  return cycle error
end if
for i in π **do**
  $K_i \leftarrow \bigcup_{j \in d_i^0} (\{j\} \cup K_j)$
end for
for i in π **do**
  $d_i \leftarrow \{j \in d_i^0 : \nexists k \in d_i^0 \setminus \{j\},\ j \in K_k\}$
  if a_i is missing or $a_i \neq K_i$ **then**
    return prerequisite mismatch
  end if
end for
return $G = (V, \{(j,i) : j \in d_i\}),\ \pi,\ \{K_i, s_i\}_{i \in V}$

Build restores the checkpoints of the required components and combines their files with the baseline. When multiple branches contribute changes, it uses a shared earlier version as the merge base where available and reports conflicting edits. The treatment applies the active component's change to this control. Improvement can start from the component's previous best checkpoint while the control remains fixed. Unrelated branches are excluded. Implementation review checks whether the code follows the specification for the active component.

For a merged component u representing decisions S, equation (6) applies with $i = u$ and the shared intervention Δ_u. The planner must supply an acyclic revised graph. For example, merging z_a and z_c in $z_a \to z_b \to z_c$ while retaining both incident dependencies gives $u \to z_b \to u$ and is rejected. The planner then revises the decomposition. Figure 4 shows single-decision and joint comparisons.

A.2 Numerical Criteria and Checkpoint Qualification

$\mathsf{Fin}(v_1, \ldots, v_k)$ is true when every argument is a finite numerical value and none is Boolean. For a performance test, the predicate specified before execution chooses $y_i^{\text{test}} = \delta_i$ or m_i^+, and the

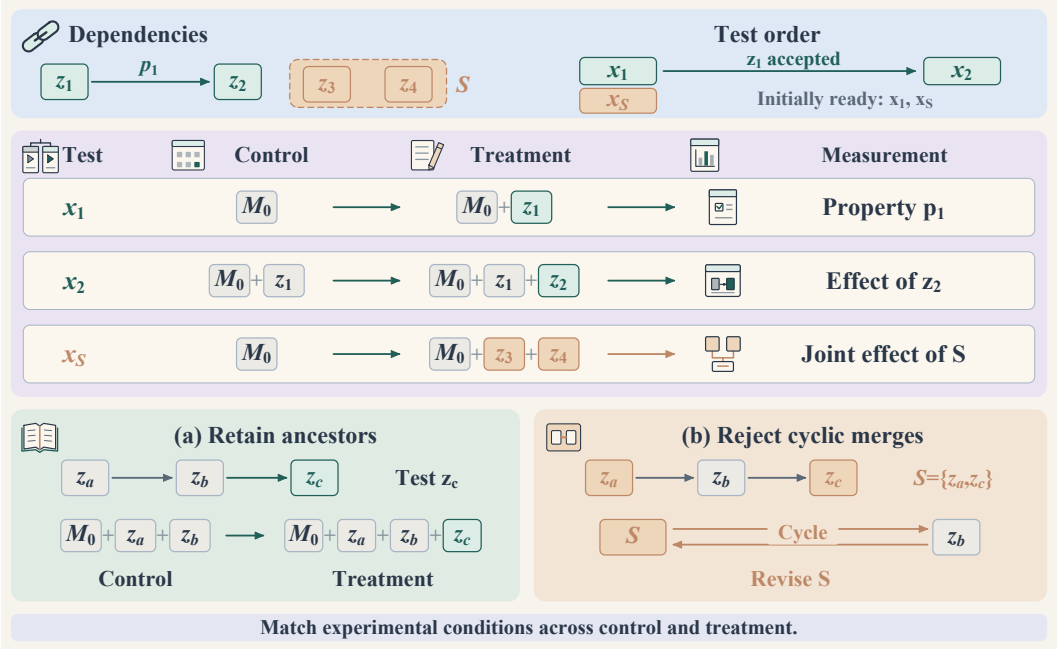

Figure 4: Comparisons that preserve required dependencies. The upstream test checks p_1; the downstream test includes z_1 in its control. Decisions in $S = \{z_3, z_4\}$ form one merged component with a shared test and result. A cyclic proposed merge is rejected during graph construction.

numerical acceptance criterion is

$$g_i^{\mathrm{dir}} = \mathbf{1}[\mathsf{Fin}(m_i^-, m_i^+)]\,\mathbf{1}[\delta_i > 0]\,\phi_i(y_i^{\mathrm{test}})\,\mathbf{1}[\delta_i > \epsilon_i]. \tag{12}$$

Here ϵ_i is the minimum improvement specified by the measurement protocol, with $\epsilon_i = 0$ for a test requiring positive improvement. Thus a positive observed gain must also satisfy the declared predicate. For a property check, let a_i^{obs} be its reported intermediate observable and $b_i = 1$ indicate exactly one report matching the specified test and linked to its execution record. Its acceptance criterion is

$$g_i^{\mathrm{ena}} = b_i\,\mathbf{1}[\mathsf{Fin}(a_i^{\mathrm{obs}})]\,\phi_i(a_i^{\mathrm{obs}}). \tag{13}$$

An aggregate observable, such as the number of failed property checks, can summarize several assertions. The execution script computes that aggregate; the criterion compares it with the threshold specified before execution.

Numerical acceptance and checkpoint qualification are separate. For attempt t, let e_{it} indicate successful execution, h_{it} an existing checkpoint, r_{it} a requested plan revision, and c_{it} a review of task constraints that has not blocked the attempt. Method implementation review (MIR), denoted by MIR_{it}, assesses whether the code implements the specified method. Let g_{it} be g_i^{dir} or g_i^{ena} according to the test type. Qualification is

$$q_{it} = e_{it}h_{it}(1 - r_{it})c_{it}\,\mathbf{1}[\mathrm{MIR}_{it} \in \{\mathrm{faithful}, \mathrm{partial}\}]\,g_{it}, \tag{14}$$

A partial implementation supports only the implemented and tested scope. Failed attempts remain in the history, while an earlier qualified checkpoint can be retained.

A.3 Memory Records and Feedback Scheduling

The component histories record the specification reference, ancestor set, checkpoints, code difference, observations, validation results, and execution artifacts. We write an attempt record as

$$F_{it} = (z_i, \nu_i, e_i, D_i, P_i, O_{it}^-, O_{it}^+, g_{it}, q_{it}, \mathcal{A}_{it}), \tag{15}$$

where ν_i identifies the specification version and $\mathcal{A}_{it}$ references supporting artifacts.

Let $\sigma_t(i) \in \{\text{pending}, \text{active}, \text{accepted}, \text{exhausted}\}$ denote search not yet started, search in progress, search ended with a qualified checkpoint, and attempts exhausted without one, respectively. With topological order π, define

$$\begin{aligned} \mathcal{U}_t &= \{i : \sigma_t(i) = \text{active}\}, \\ \mathcal{R}_t &= \{i : \sigma_t(i) = \text{pending},\ \forall j \in d_i,\ \sigma_t(j) = \text{accepted}\}, \\ i_t &= \begin{cases} \min_\pi \mathcal{U}_t, & \mathcal{U}_t \neq \varnothing, \\ \min_\pi \mathcal{R}_t, & \mathcal{U}_t = \varnothing,\ \mathcal{R}_t \neq \varnothing, \\ \bot, & \text{otherwise.} \end{cases} \end{aligned} \tag{16}$$

$\min_\pi$ denotes the first component in the stored order; $\bot$ denotes no available component or checkpoint. Thus an active component is processed before a newly ready one.

Let n_i count started attempts, u_i infrastructure attempts that did not start, and ℓ_i the last outcome. The next implementation action is

$$\text{Op}(i) = \begin{cases} \textsc{Draft}, & n_i = u_i = 0, \\ \textsc{Debug}, & \ell_i \in \mathcal{F}_{\text{impl}}, \\ \textsc{Check-Expectation}, & \ell_i = \text{contract_failed}, \\ \textsc{Improve}, & \text{otherwise}, \end{cases} \tag{17}$$

Here `contract_failed` denotes a valid execution whose result fails the declared test criterion. $\mathcal{F}_{\text{impl}}$ contains execution failure, implementation failure, a missing checkpoint, and code changes that depart from the specification. `Check-Expectation` reassesses the expected effect when a valid test fails its criterion; `Debug` repairs execution or implementation errors. The agent uses the selected action and research memory to generate the next implementation. A failed design or test can prompt replanning within the revision budget, including a revised decomposition or a group of decisions tested together.

Component search retains the best qualified checkpoint. Performance tests rank checkpoints by the task metric. Property checks use the specified comparison rule, with task performance breaking ties. Search continues until its attempt limit is reached; equation (18) gives the resulting state.

A.4 Full Research Loop

Algorithm 3 combines the graph, numerical predicates, research memory, and feedback policy. Its construction step applies the planner in Section 3.1 followed by Algorithm 2. Component attempts, execution launches, command durations, and outer revisions have separate limits. A failed execution counts as an attempt; an infrastructure failure before execution has a separate counter. Appendix C specifies state updates and evidence reuse.

Algorithm 3 Experiments that preserve prerequisites and method revision

Input: Task $\mathcal{T}$, baseline M_0, candidate M, configured resource limits

$(G, \pi, K, s) \leftarrow \mathrm{Construct}(M)$ using Algorithm 2
$\mathcal{L} \leftarrow \varnothing$; $\widehat{M} \leftarrow M_0$; $\sigma(i) \leftarrow \mathrm{pending}$
while the execution and revision limits permit another step **do**
  $i \leftarrow i_t$ from equation (16)
  if $i \neq \perp$ **then**
    $\mathcal{H} \leftarrow \mathcal{H}_t(i)$; $o \leftarrow \mathrm{Op}(i)$ by equations (9) and (17)
    $P_i \leftarrow \mathrm{Build}(M_0, K_i)$; implement $T_i = P_i \oplus \Delta_i$ using $(o, \mathcal{H})$
    Obtain O_i^-, O_i^+ and the available measurement records by equation (7)
    $(g_{it}, q_{it}) \leftarrow$ equations (12) to (14)
    $\mathcal{L}(i) \leftarrow \mathcal{L}(i) \parallel F_{it}$; update n_i, u_i and retained checkpoint
    Update $\sigma(i)$ by equation (18); select the next action using equation (17)
  else
    if $\forall j \in V,\ \sigma(j) = \mathrm{accepted}$ **then**
      $M^+ \leftarrow \mathrm{Assemble}(\{\widehat{T}_j\}_{j \in V})$; evaluate M^+
      Update $\widehat{M}$ by equation (10); record the complete result
    else
      Report the blocking or exhausted components
    end if
  end if
  if feedback requests a permitted component or method revision **then**
    $M \leftarrow \mathrm{Revise}(M, \mathcal{L})$; reconstruct (G, π, K, s)
    Initialize its component histories and reuse checkpoints satisfying equation (19)
  else
    Stop if component search is blocked or complete-candidate search has finished
  end if
end while
return $\widehat{M}, \mathcal{L}$

B Plan and Experiment Records

For each component, DAGLoop stores its specification and the history of implementation and evaluation attempts. The following fields describe these records.

The design fields specify the component identifier, purpose, mechanism, inputs, outputs, and dependencies. The experiment fields specify the expected effect, test type, and acceptance criterion. The implementation check describes how to assess whether the code follows the design; the optimization space specifies which changes are allowed; the downstream requirements state the properties required by later components. A performance test compares task performance, and a property check checks an intermediate property. Joint tests also record their original decisions. The tuple s_i groups the design and experiment fields for each component.

An attempt records the implementation action, ancestor set, checkpoints, code difference, measurements, implementation review, review of task constraints, and artifact references. Commands and execution records link observations to the corresponding attempt. The record F_{it} in equation (15) summarizes these fields. The research memory in equation (9) preserves detailed current-component and ancestor histories and summarizes other branches.

C State Transitions and Evidence Reuse

Graph and semantic checks. The identifier map N trims and lowercases strings, replaces runs of non-alphanumeric characters with a hyphen, and assigns an index-based identifier when the result is empty. The constructor computes ancestor sets, removes redundant edges, and validates acceptance prerequisites according to equations (5) and (11). Inputs and outputs are checked for structure and nonemptiness; the critic evaluates their semantic compatibility and the adequacy of the proposed tests. A merged component passes the ordinary DAG checks after replanning.

Control measurement. A root component has no prerequisites and uses the baseline as its control. For a single direct parent, the component history can provide that parent's finite metric. Components with multiple direct prerequisites require a measurement of the combined control implementation. A reported control attempt can supply this value after matching the control implementation, source state, and evaluation settings. Control reuse checks the saved measurement evidence, protocol, and restorable checkpoint. A control value that fails these checks is cleared. Protocol identity is defined by the recorded evaluation configuration.

Numerical predicates and repetitions. The criterion for performance tests accepts one comparison of signed metric improvement or candidate metric with a finite numerical constant. The criterion for property checks accepts one scalar comparison, including a reported aggregate such as a failed-check count. Unsupported expressions return a plan-format error. Boolean, missing, and nonfinite values fail the numerical check. Equality is exact unless tolerance is expressed in the declared observable or predicate. The measurement protocol specifies the reported statistic. For repeated measurements, the execution system checks the sample count, finite control and treatment samples, and the specified mean or median against the reported aggregate. These criteria implement threshold decisions, not significance tests for the measured effects.

Qualification and retained checkpoints. In equation (14), the task-constraint indicator excludes `violation` and `insufficient_evidence`. Qualification additionally requires successful execution, an existing checkpoint, a faithful or partial implementation review, a passed component test, and no pending plan revision. The latest attempt's test result and the retained checkpoint's test result are stored separately.

Let $\widehat{T}_i^t$ be the retained qualified checkpoint after attempt t and N_i the component attempt limit. Search continues to this limit, preserving the best qualified checkpoint. After a started attempt, the state is updated as follows:

$$\sigma_{t+1}(i) = \begin{cases} \text{accepted}, & n_i \geq N_i,\ \widehat{T}_i^t \neq \bot, \\ \text{exhausted}, & n_i \geq N_i,\ \widehat{T}_i^t = \bot, \\ \text{active}, & \text{otherwise.} \end{cases} \tag{18}$$

A non-started attempt instead increments u_i and follows the infrastructure limit. On exhaustion, an existing qualified checkpoint is preserved; otherwise the workflow reports infrastructure exhaustion. For a property check, checkpoints are ranked using the specified comparison direction. Equality tests use distance from the target value, and task performance breaks ties.

Checkpoint reuse. Write c_j for a component's recorded specification and acceptance requirements, h_j for its retained checkpoint identity, and $\mathbf{h}_{d_i}$ for the ordered direct-parent checkpoint identities. An old accepted checkpoint can enter a new component history only if

$$\begin{aligned}\mathrm{Inherit}(i) = \ & \mathrm{ExecMatch} \wedge \mathrm{Restorable}(h_i) \\ & \wedge \ \left(\forall j \in K_i \cup \{i\},\ c_j^{\mathrm{new}} = c_j^{\mathrm{old}}\right) \\ & \wedge \ \left(\forall j \in K_i,\ \sigma^{\mathrm{new}}(j) = \mathrm{accepted},\ h_j^{\mathrm{new}} = h_j^{\mathrm{old}}\right) \\ & \wedge \ (\mathbf{h}_{d_i}^{\mathrm{new}} = \mathbf{h}_{d_i}^{\mathrm{old}}).\end{aligned} \tag{19}$$

ExecMatch compares execution identifiers derived from baseline records. A specification reference identifies the tested version of the design, a protocol identifier denotes the recorded evaluation configuration, and a checkpoint identifier locates the saved implementation. These checks associate reuse with the recorded experiment. Old records remain in history when a checkpoint cannot be reused in the revised context.

Resource limits. Resource limits apply to component implementation attempts, experiment launches, individual commands, and revisions of the candidate method. The reference configuration uses six component attempts, six experiment launches per sandbox, 7,200 seconds per command, and three outer revisions. A control measurement counts as an experiment launch. An executed failure consumes a component attempt; a failure before execution increments the separate infrastructure count. Component progress does not consume an outer method-revision round. Cost is recorded, and component selection follows equation (16).

D PROOFS

D.1 BASELINE AND COMPLETE-CANDIDATE EVALUATIONS

For the metric f on feasible configurations and the marginal effect in equation (4), we have:

Proposition 1 (Insufficiency of baseline and complete-candidate evaluations)**.** *For an arbitrary deterministic metric f, the baseline value $f(\varnothing)$ and complete-candidate value $f(\{1,2\})$ do not determine the sign of $\delta_{2|1}$.*

Proof. Consider two metrics on the three feasible configurations:

	$f(\varnothing)$	$f(\{1\})$	$f(\{1,2\})$	$\delta_{2\|1}$
f_A	0	0	1/2	1/2
f_B	0	1	1/2	−1/2

Both metrics lie in $[0,1]$ and have identical baseline and complete-candidate values. Any rule that receives only those two values has identical inputs in the two cases, although the marginal effects have opposite signs. It therefore cannot determine the sign for all metrics. □

This argument concerns what the baseline and complete-candidate values determine without an additional structural assumption on f. A targeted experiment supplies the missing control observation; repeated measurements are a separate requirement when observations are noisy.

D.2 MINIMAL PREREQUISITE CONTEXT

A set $C \subseteq V$ is *closed under prerequisites* if $j \in C$ implies $d_j^0 \subseteq C$.

Proposition 2 (Minimal prerequisite context)**.** *For a DAG G_0, K_i is the unique smallest set containing d_i^0 that is closed under prerequisites. Also, $i \notin K_i$, $K_i \cup \{i\}$ is closed under prerequisites, and the edge reduction in equation (11) preserves every ancestor set.*

Proof. Every member of K_i lies on a directed path to i. If $j \in K_i$ and $k \in d_j^0$, appending the path from j to i shows $k \in K_i$, so K_i is closed under prerequisites. It contains d_i^0. Conversely, any set containing d_i^0 that is closed under prerequisites must contain every predecessor along each path ending at i, hence all of K_i. This proves uniqueness and minimality. Acyclicity gives $i \notin K_i$, and adding i preserves closure because $d_i^0 \subseteq K_i$.

For the reduction, process vertices in topological order. A removed parent $j \in d_i^0$ is an ancestor of another direct parent k. Following this relation among direct parents must terminate at a retained parent k^*, since the graph is finite and acyclic. By induction, the reduced graph on the earlier vertices preserves the path from j to k^*. The retained edge (k^*, i) therefore preserves reachability from j to i. Retained edges are original edges, so the reduction creates no new reachability. Induction over π establishes equality of all ancestor sets. □

Minimality is relative to the declared prerequisites. The proposition establishes which components enter a test context; implementation compatibility and experimental validity are checked separately.

E Definitions of Representative Tasks

This appendix explains representative task objectives and evaluator interfaces for the benchmark results. The task interfaces, measurement boundaries, and reward formulas below describe the previously audited AutoLab checkout at commit `4127da3dde8449be61a1cf9859473b9fbbd51751`; the public repository supplies the task descriptions (AutoLab Authors, 2026). The score comparison uses the reward records in Appendix F for all reported tasks.

E.1 BM25 Retrieval in Go

The AutoLab task uses BM25 retrieval with term frequency, inverse document frequency (IDF), and document-length normalization (AutoLab Authors, 2026). For query u and document d, the task computes the following retrieval score:

$$\begin{aligned} \mathrm{BM25}(u,d) &= \sum_{w \in \mathrm{uniq}(u)} \mathrm{IDF}(w)\, \frac{\mathrm{tf}(w,d)(k_1+1)}{\mathrm{tf}(w,d) + k_1(1-b+b|d|/\overline{L})}, \\ \mathrm{IDF}(w) &= \log\left(1 + \frac{N - \mathrm{df}(w) + 0.5}{\mathrm{df}(w) + 0.5}\right), \quad k_1 = 1.2, \quad b = 0.75. \end{aligned} \tag{20}$$

Here N is the corpus size, $\mathrm{df}(w)$ counts documents containing w, $\mathrm{tf}(w,d)$ counts its occurrences in d, and $\overline{L}$ is mean document length. $\mathrm{uniq}(u)$ denotes unique query terms, so repeated tokens are counted once; $|d|$ is document length. The output contains at most ten positive-score documents, ordered by decreasing score and increasing document identifier on ties. An empty query returns no results for the retrieval task.

The correctness check compares the batch query count, hit count, and checksum with the reference. Scores are rounded after multiplication by 10^6 before checksum aggregation. The timing workload uses 4,000 documents and 40 queries, with the median over five query seeds:

$$m_{\mathrm{BM25}}(M) = \mathrm{median}_{r=1}^{5}\, t\big(\mathrm{NewEngine}_M(\mathcal{D}), \mathrm{RunQueries}_M(\mathcal{U}_r)\big). \tag{21}$$

The measured interval includes index construction, query processing, ranking, and batch checksum computation. It excludes compilation, corpus and query generation, the preceding correctness check, and printing. "Go" in `bm25_search_go` identifies the implementation language.

For measured seconds $t > 0$, the official task reward is

$$R_{\mathrm{BM25}}(t) = \begin{cases} 0, & 2.1/t \le 1, \\ \min\{1, \tfrac{1}{2}\log(2.1/t)/\log(2.1/0.03)\}, & 2.1/t > 1. \end{cases} \tag{22}$$

The scoring anchors 2.1 and 0.03 are benchmark constants; local relative improvement uses the locally reproduced baseline instead.

E.2 TRANSFORMS, JOINS, STORAGE, AND MODEL TRAINING

Fourier transform (`fft_rust`). For a real signal $x \in \mathbb{R}^n$ with $n = 32768$, the task computes the complex discrete Fourier transform

$$X_k = \sum_{j=0}^{n-1} x_j \exp(-2\pi \mathrm{i} jk/n), \qquad 0 \le k < n. \tag{23}$$

The metric is the median duration of the transform call. Parseval and spectral-checksum checks use the task's 10^{-6} relative tolerance. The implementation is single-threaded Rust using the standard library for the transform.

Hash join (`hash_join`). Given indexed integer key–payload tables $R = (r_a)_{a=1}^{n_R}$ and $S = (s_b)_{b=1}^{n_S}$, the task computes the matching row pairs of an inner equijoin

$$\mathcal{J} = \{(a, b) : 1 \le a \le n_R,\ 1 \le b \le n_S,\ r_a.\mathrm{key} = s_b.\mathrm{key}\}. \tag{24}$$

Every matching row pair contributes, including duplicate keys. The output is a match count and a 64-bit checksum. The benchmark uses 20,000 rows in R and five million in S, measuring one join call after warm-up under a single-thread constraint.

Number-theoretic transform (`ntt_butterfly_cuda`). For $a \in \mathbb{F}_p^n$, the forward transform has the form

$$A_k = \sum_{j=0}^{n-1} a_j \omega^{jk} \pmod p, \qquad p = 2^{64} - 2^{32} + 1, \tag{25}$$

where ω is the task's forward nth root of unity. The CUDA implementation returns canonical field elements in natural order, in place. Correctness is bit-exact, and latency is the median CUDA-event time after warm-up, using the prescribed workspace.

Multi-scalar multiplication (`msm_pippenger_bls12_381_cuda`). The task maps BLS12-381 G_1 points P_i and 256-bit scalars s_i to

$$Q = \sum_{i=1}^{n} [s_i] P_i. \tag{26}$$

The evaluator checks the affine output in the required Montgomery representation and measures median CUDA-event latency. The released starter uses naive double-and-add.

SSTable compaction (`sstable_compaction_rs`). Sorted string tables store ordered key–value entries on disk. This task merges several prefix-compressed tables, retains the visible latest value for each key, and removes shadowed or deleted entries. It verifies live-entry count, checksum, and output size. The median of five compaction calls includes decoding, merging, and output construction, while input generation occurs outside the measured interval.

Visual reasoning (`grpo_multisource`). Group relative policy optimization (GRPO) fine-tunes Qwen2.5-VL-7B using Geometry3K, MathVision, and ChartQA, producing a low-rank adaptation (LoRA) adapter. The final native metric is accuracy on 100 held-out MathVista examples. Let a be this accuracy and r the general visual question answering (VQA) accuracy retention on 50 examples. The task reward is

$$R_{\mathrm{GRPO}}(a, r) = \mathbf{1}[r \ge 0.9]\ \mathrm{clip}_{[0,1]}\left(\frac{a - 0.20}{0.65 - 0.20}\right). \tag{27}$$

Retention is the adapted-to-baseline VQA accuracy ratio, defined as one when baseline VQA accuracy is zero. The smaller local evaluation uses 20 MathVista and 10 VQA examples; the final reward uses the 100-plus-50 evaluation.

F FULL AUTOLAB RESULTS

Table 3: CPU native metrics and task scores. T2 = Terminus-2; both use glm-5.2. T2 reports mean task scores only. Differences compare task scores.

		Native metric		Task score		
Task	Metric	Local baseline	**DAGLoop** metric	**DAGLoop** score	T2 mean	Difference
adaptive_compression	bpb ↓	5.019 bpb	3.9054	0.450200	0.496800	−0.046600
adversarial_splay	Rotations ↑	48656 rotations	67008	0.500000	0.584100	−0.084100
aes128_ctr	Time (s) ↓	7.683878 s	0.046295	0.613200	0.579350	+0.033850
agent_tool_routing	Time (s) ↓	9.528870 s	0.655875	0.390800	0.240300	+0.150500
bm25_search_go	Time (s) ↓	12.479437 s	0.000339	1.000000	1.000000	+0.000000
bvh_raytracer	Time (s) ↓	16.455072 s	0.216169	0.296100	0.455050	−0.158950
concurrent_kv_wal	Time (s) ↓	62.550840 s	1.21775	0.476400	0.603950	−0.127550
discover_sorting	Comparators ↓	80 comparators	60	1.000000	0.850000	+0.150000
fft_rust	Time (s) ↓	20.984767 s	0.001791	0.468400	0.526600	−0.058200
flash_attention	Time (s) ↓	2.278774 s	0.043171	0.708400	0.469450	+0.238950
fredkin_sort_network	Gates ↓	128 gates	78	0.666700	0.866700	−0.200000
gaussian_blur	Time (s) ↓	35.827048 s	0.227539	0.512200	0.557350	−0.045150
hash_join	Time (s) ↓	97.795046 s	0.03883	0.502400	0.640200	−0.137800
levenshtein_distance	Time (s) ↓	3.440261 s	0.489731	0.380500	0.387400	−0.006900
radix_sort	Time (s) ↓	9.120008 s	0.2187	0.592100	0.410750	+0.181350
regex_engine	Time (s) ↓	3.305667 s	0.135549	0.858700	0.880300	−0.021600
resnet_bit_flip	Bits ↓	95 bits	6	0.937500	0.881250	+0.056250
safety_router	Params ↓	16641 params	1301	0.921820	0.992128	−0.070308
sha256_throughput	Time (s) ↓	2.547909 s	0.502071	0.285300	0.179850	+0.105450
smallest_game_player	Params ↓	17988 params	913	0.949100	0.473950	+0.475150
sstable_compaction_rs	Time (s) ↓	0.197735 s	0.087564	0.069600	0.347900	−0.278300
stack_machine_golf	Instructions ↓	5132 instructions	1024	1.000000	1.000000	+0.000000
toy_isa_opt	Cycles ↓	9220 cycles	1544	1.000000	1.000000	+0.000000
vliw_scheduler	Cycles ↓	4080 cycles	1283	0.971200	1.000000	−0.028800
z_order_range_scan	Time (s) ↓	5.580838 s	0.116466	0.308700	0.195300	+0.113400

Table 4: GPU native metrics and task scores. T2 = Terminus-2; both use glm-5.2. Native results are recorded observations. T2 reports mean task scores only. Differences compare task scores.

		Native metric		Task score		
Task	Metric	Local baseline	**DAGLoop** metric	**DAGLoop** score	T2 mean	Difference
`data_select_ifeval`	IFEval ↑	0.373383	0.417745	0.377449	0.858050	−0.480601
`flux2_klein_lora`	Quality ↑	0.4935	0.6305	0.354500	0.000000	+0.354500
`grpo_multisource`	Accuracy ↑	0.8667 accuracy	0.58	0.844400	0.877750	−0.033350
`huffman_canonical_decode_cuda`	Time (ms) ↓	56.515778 ms	9.2824	0.310900	0.379750	−0.068850
`icp_correspondence_step_cuda`	Time (ms) ↓	66.515228 ms	0.221952	0.521300	0.499400	+0.021900
`llm_online_serving`	Serving ↑	1.4044 serving_score	1.5654	1.000000	0.000550	+0.999450
`moving_mnist_world_model`	PSNR (dB) ↑	14.558327 dB	15.275893	0.212649	0.257882	−0.045233
`msm_pippenger_bls12_381_cuda`	Time (ms) ↓	531.247986 ms	27.59408	0.672500	0.382400	+0.290100
`multilingual_ocr`	CER ↓	0.7257 CER	0.3200	1.000000	1.000000	+0.000000
`ntt_butterfly_cuda`	Time (ms) ↓	107.004387 ms	1.136992	0.513300	0.512450	+0.000850
`scaling_law`	PPL ↓	151.635016 PPL	22.618779	0.340174	0.605381	−0.265207

PSNR = peak signal-to-noise ratio; CER = character error rate; PPL = perplexity. IFEval measures instruction following. Quality and Serving denote the task-defined image-quality and serving scores.

G Additional FML-Bench Results

We additionally ran DAGLoop on 18 FML-Bench tasks with the task and repository specified (Zou et al., 2026). The results cover 13 GPU tasks and five CPU tasks. Table 5 reports their native metrics separately from AutoLab. For each task, we report the native metric from one DAGLoop run alongside its corresponding local baseline.

Table 5: FML-Bench native results from one DAGLoop run per task, compared with local baselines.

Task	Metric	Baseline	**DAGLoop**
GPU tasks (13)			
Opacus	Acc. ↑	0.5869	**0.6010**
DomainBed-CM	In-acc. ↑	0.2613	**0.7010**
ART	Defense ↑	0.3131	**0.8719**
OpenOOD	AUROC ↑	0.8693	**0.8997**
PyCIL	Inc.-acc. ↑	0.5957	**0.6016**
USB	Acc. ↑	0.0783	**0.0961**
EasyFSL	Acc. ↑	0.6534	**0.7034**
PFLlib	Acc. ↑	0.4524	**0.4914**
DomainBed-OH	Avg.-acc. ↑	0.5962	**0.6280**
PrivacyMeter	AUC gap ↓	0.3177	**0.0252**
Lightly	Acc. ↑	0.7537	**0.7674**
Solo-learn	Linear acc. ↑	0.4667	**0.4879**
OpenUnlearning	KS p ↑	≈ 0	**0.3890**
CPU tasks (5)			
Fairlearn	$\lvert\text{DP}\rvert$ ↓	0.1866	**0.0029**
gCastle	SHD ↓	72	**24**
AIF360	$\lvert\text{AOD}\rvert$ ↓	0.2399	**0.0209**
CausalML	MAE ↓	1.2963	**1.2680**
Cont.-Learn.	Avg.-acc. ↑	0.2841	**0.5935**

CM and OH denote ColoredMNIST and OfficeHome, respectively. **Metrics:** DP = demographic-parity difference; AOD = average odds difference; SHD = structural Hamming distance. Acc. = accuracy; In-acc. = mean accuracy on the splits designated "in" by DomainBed; Inc.-acc. = average incremental accuracy; Linear acc. = linear evaluation accuracy; Avg.-acc. = average accuracy. AUROC = area under the receiver operating characteristic curve; AUC gap = the task-defined area-under-curve gap; MAE = mean absolute error; KS p = Kolmogorov–Smirnov test p-value. Defense denotes the task-defined defense score. Accuracies remain fractions.