HARNESSOPT: AGENT OPTIMIZATION THROUGH HARNESS KERNELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimizing a language-model agent requires coordinating its workflow, tools, and supporting components. Direct code editing is expressive, but patch size poorly captures the behavioral scope of an update. We introduce HARNESSOPT, a framework that optimizes a persistent *Harness Kernel*: a compact, structured specification of global workflow and local components. HarnessOpt extends SkillOpt-style refinement to complete harnesses by aggregating execution evidence, adopting bounded behavioral edits, and compiling each revised Kernel into coordinated runtime changes. Memory and patient selection guide exploration while retaining the best evaluated runtime. Model parameters remain fixed throughout. The same process supports black-box adaptation, white-box optimization, and from-scratch generation for individual task distributions and task mixtures. Evaluations with Codex, Claude Code, Pi, and standalone runtimes demonstrate strong task performance and transfer across models, hosts, and benchmarks. Paired artifact studies show higher task success with substantially less host-code growth under Kernel editing than direct runtime editing.

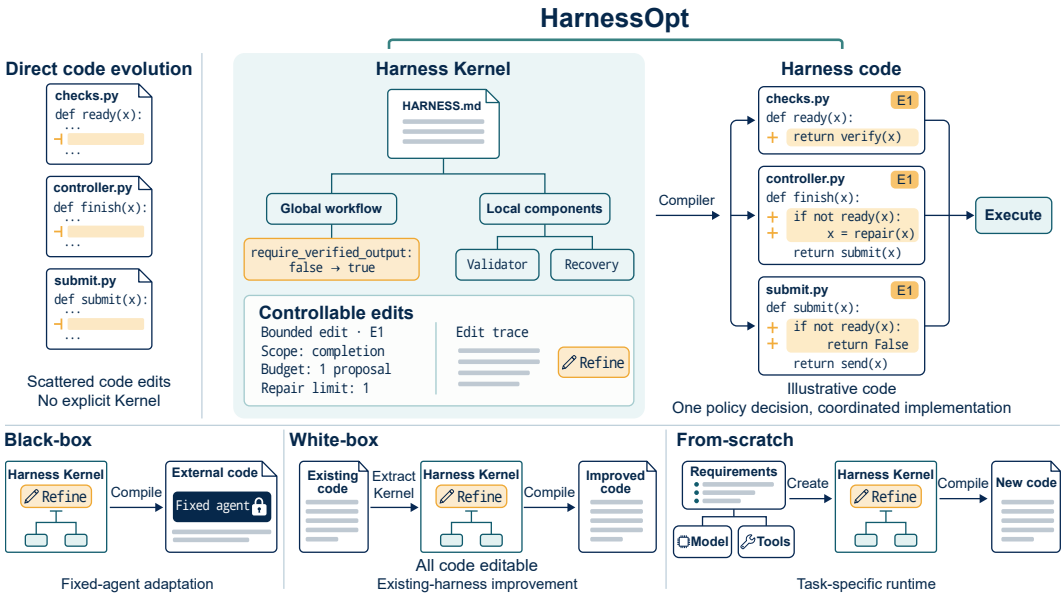

Figure 1: **One behavioral decision, coordinated implementation.** The Harness Kernel links global workflow and local components, allowing a bounded behavioral proposal to compile into coordinated changes across files.

1 INTRODUCTION

We study how to optimize a language-model agent's complete harness for a target task distribution while keeping model parameters fixed. The harness governs workflow, tools and assets, validation, memory and context, and recovery. These choices interact: a stronger validator helps only when

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.

the workflow invokes it appropriately and uses its findings to guide repair or completion. Harness optimization must therefore coordinate global workflow decisions with local component behavior.

Recent code-based approaches make this broad search space accessible. Meta-Harness uses a coding agent to iteratively revise a full harness, drawing on historical candidates, evaluation scores, and execution traces (Lee et al., 2026). Such editing is expressive, but patch size does not reliably measure an update's behavioral scope. One decision may require changes across many functions and files, while a one-line conditional can redirect the entire workflow. Patch-based budgets therefore offer limited control over behavioral scope.

SkillOpt offers a complementary discipline built around reflection, bounded edits, evaluation, and memory (Yang et al., 2026). Within a compact skill document, counting targeted additions, deletions, and modifications provides control analogous to a learning rate. Extending this discipline to a complete harness requires keeping each intended behavioral decision identifiable as its implementation spans code, tools, and other assets.

Our approach builds on an engineering observation: capable coding agents can expand concise, organized specifications into substantial programs. Engineers specify a harness's workflow, component responsibilities, configuration, and essential logic; the coding agent implements them across functions and files. Maintaining this specification as the persistent optimization object lets each revision express a behavioral change before expanding it into code. The specification preserves the conditions, state transitions, and critical details needed to realize that change.

We introduce HARNESSOPT, centered on the *Harness Kernel*, a persistent, structured behavioral design rooted at `HARNESS.md`. The entry describes the harness and its global workflow, then indexes subordinate assets organized in a directory tree. Text, configuration, and critical code specify local tools, checks, component contracts, and memory and context policies. This hierarchy supports progressive disclosure: the entry supplies a global view, while relevant branches expose implementation-sensitive details when needed. Global workflow decisions and local mechanisms thus remain jointly inspectable within one optimization object. In a single-module reconstruction study, the Kernel and interface notes occupy 56.05% fewer UTF-8 bytes than Pi's complete `agent-loop.js`. AppWorld-normal and DABStep performance remains nearly unchanged, suggesting that this module's smaller specification preserves the information needed for these tasks.

A coding agent such as Claude Code or Codex serves as a *compiler*, expanding a Kernel into a runtime or implementing a revision across files. Requiring verified output before submission, for example, affects validation, recovery, and submission logic (Figure 1). The Kernel specifies the check, bounded repair, and recheck; the compiler implements them, and execution supplies feedback. This preserves a clear behavioral scope even when the implementation is extensive. Table 6 further shows that Kernel editing produces fewer added functions and less host-code growth, with higher task success than direct runtime editing.

This representation scales SkillOpt-style optimization from individual skills to complete harnesses. Reflection aggregates development evidence into bounded behavioral proposals; compilation and evaluation turn them into tested revisions. History, mechanism memory, and an adaptive edit curriculum guide subsequent updates. Patient selection permits coupled changes to develop across steps while retaining the best evaluated artifact separately.

The same Kernel interface supports three access settings: *black-box adaptation* edits wrappers and assets around a fixed opaque agent; *white-box optimization* extracts and refines a Kernel from accessible harness code; and *from-scratch generation* initializes a Kernel from requirements and compiles a new runtime. Each setting retains its own access boundary and starting resources.

We evaluate all three access settings across six benchmark suites and seven task domains, using Codex, Claude Code, Pi-based harnesses, and standalone runtimes with GPT-6, GPT-5.5, Opus 5, and Sonnet 5. Comparisons include native agents, SkillOpt, LongHorizon, Meta-Harness, Life-Harness, MemoHarness, and Self-Harness. The reported results show substantial gains over native agents (Table 1): DABStep success rises from 70.61% to 97.12% with GPT-6/Codex, and WebShop success rises from 45.36% to 65.00% with Sonnet 5/Claude Code. These examples illustrate the practical benefit of optimizing the harness around a fixed model and agent interface.

Our contributions.

1. We introduce HARNESSOPT, to our knowledge the first framework to support and study all three harness access settings through one optimization process, for individual task distributions and mixtures of task families.
2. We propose the Harness Kernel as a compact, controllable representation of global workflow and local components, extending SkillOpt-style bounded editing through compilation and execution feedback.
3. Evaluations show strong task performance and transfer across models, hosts, and benchmarks; representation comparisons and execution studies show how the Kernel supports effective optimization and concise implementations.

2 RELATED WORK

Agent and harness optimization. Prior work optimizes instructions and demonstrations or accumulates reusable experience in memories and skills (Opsahl-Ong et al., 2024; Wang et al., 2024; Zhang et al., 2025; Yang et al., 2026). DSPy, TextGrad, and GEPA's optimize_anything extend feedback-driven optimization to compound programs and code (Khattab et al., 2024; Yuksekgonul et al., 2024; Agrawal et al., 2026). Life-Harness and MemoHarness adapt runtime interfaces and configurations (Xu et al., 2026; Huang et al., 2026); Meta-Harness, AHE, and Self-Harness revise accessible harness code (Lee et al., 2026; Lin et al., 2026; Zhang et al., 2026a). AutoSaddler structures patches, while HarnessCompass and ModularRSI restrict component-level search before integration (Park et al., 2026; Zhang et al., 2026b; Wu et al., 2026). SkillOpt supplies our starting discipline of bounded edits, reflection, evaluation, and memory.

Kernel-based controllable editing. Natural-Language Agent Harnesses express run-level policy in interpreted documents, and Ecdysis produces per-round modification specifications for a coding agent (Pan et al., 2026; Yue et al., 2026). HarnessOpt maintains a persistent Kernel spanning global workflow and local components. A budget limits adopted behavioral proposals, and a compiler realizes the integrated revision across code and assets; the budget therefore counts decisions rather than changed lines or files. Preserving the evolving design and its dependencies lets one proposal coordinate multiple implementation sites.

Unifying three access settings. For opaque agents, SkillOpt optimizes external skills and LongHorizon supplies an external control workflow (Yang et al., 2026; Ma et al., 2026). The source-editing methods above operate on accessible harnesses; ADAS also synthesizes new agents over a small execution substrate (Hu et al., 2024). Meta-Harness permits whole-program rewrites, although its reported searches start from supplied harnesses. HarnessOpt supports *black-box adaptation* through wrappers around fixed agents, *white-box optimization* through Kernels extracted from editable source, and *from-scratch generation* through Kernel initialization and runtime compilation. The same procedure combines a persistent whole-harness behavioral specification, a budget on adopted proposals, and separate runtime compilation across all three settings, while retaining their distinct starting resources and access boundaries.

3 HARNESS KERNEL OPTIMIZATION

HarnessOpt extends SkillOpt's bounded refinement to complete harnesses (Yang et al., 2026). A persistent *Harness Kernel* K specifies behavior, and a coding-agent compiler realizes it as a runtime R. Each iteration reflects on execution evidence, adopts a bounded Kernel revision, compiles it, and evaluates the runtime (Figure 2).

For task-family distributions $\{\mathcal{D}_j\}_{j=1}^{m}$, the objective is

$$J(R) = \sum_{j=1}^{m} w_j \mathop{\mathrm{E}}_{x\sim\mathcal{D}_j} \left[s_j(\mathrm{Run}(R, M, \mathcal{I}_j, x)) \right], \qquad \sum_j w_j = 1, \tag{1}$$

where M is the fixed agent model, $\mathcal{I}_j$ defines interfaces and execution constraints, s_j scores executions, and $w_j \geq 0$ sets the target mixture. Development executions estimate this objective and guide optimization. A single benchmark uses $m = 1$; joint optimization retains one shared Kernel/runtime with fixed environment adapters and declared family weights. Uniform weights average family-level utilities.

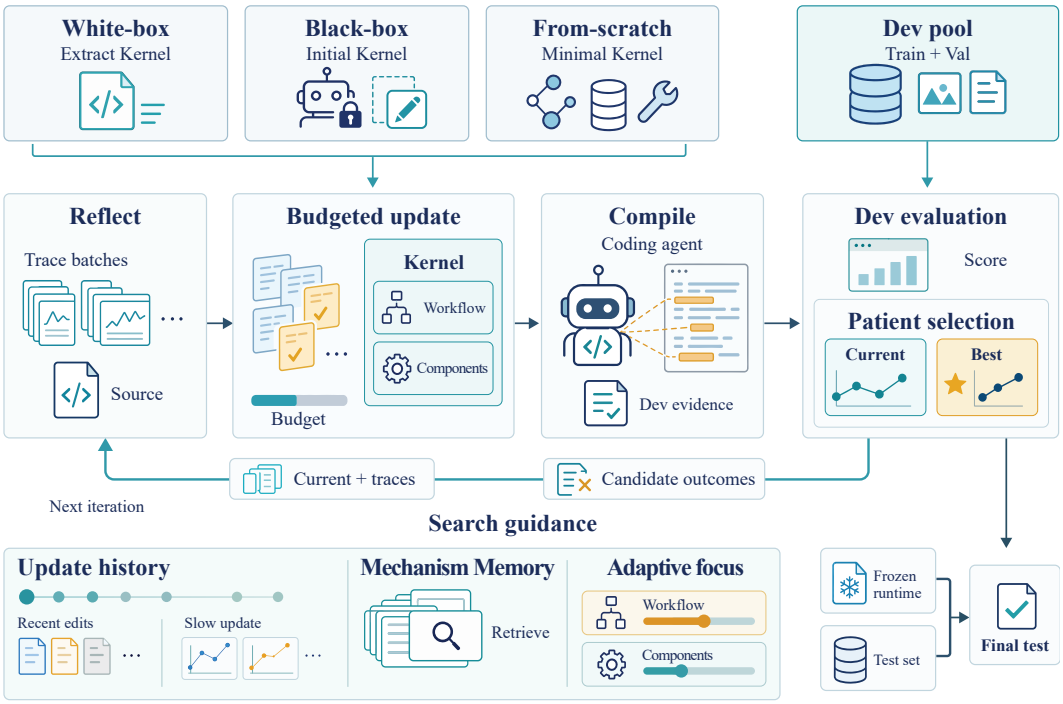

Figure 2: **HarnessOpt optimization loop.** Execution feedback drives bounded Kernel revisions and coordinated compilation. Patient selection explores candidates while retaining the best evaluated runtime for final testing.

3.1 The Harness Kernel and Three Access Settings

A Kernel specifies *global workflow*—stage ordering, routing, information flow, and completion—and *local components*: tools, validators, recovery routines, and memory or context policies. Its `HARNESS.md` entry indexes text, configuration, and critical code that define component contracts and workflow dependencies, allowing a behavioral revision to coordinate changes across files.

Three initializations. *White-box optimization* extracts a Kernel from supplied source, retained as an implementation reference. *Black-box adaptation* specifies a wrapper and assets around an opaque agent with fixed internals. *From-scratch generation* initializes a Kernel from requirements and model/tool primitives, then generates a runtime. Each setting evaluates (K_0, R_0) before entering the same refinement loop within its access boundary.

3.2 Evidence-Guided, Budgeted Behavioral Updates

At iteration t, reflection reads the Kernel K_t, minibatches of scored development trajectories from R_t, and permitted source feedback. Each *behavioral proposal* specifies a workflow or component change and its evidence. Evidence is aggregated across minibatches; overlapping proposals are consolidated, with dependencies and conflicts guiding selection of $\mathcal{P}_t$:

$$|\mathcal{P}_t| \le b_t, \qquad K'_t = \mathcal{U}(K_t, \mathcal{P}_t), \tag{2}$$

where b_t bounds adopted behavioral proposals, optionally by category, and $\mathcal{U}$ integrates them into one candidate Kernel. The budget measures behavioral decisions rather than implementation size: additional evidence can strengthen a revision while its scope stays bounded. Integration reconciles affected workflow rules and component contracts.

3.3 Coordinated Realization and Execution Feedback

A coding-agent compiler realizes the candidate under the available interfaces and editable boundary:

$$R'_t = \mathcal{C}(K'_t; K_t, R_t, \mathcal{I}, E_t^{\text{comp}}), \tag{3}$$

where $\mathcal{I}$ collects task interfaces and E_t^{comp} contains permitted development evidence. The reference pair supports incremental edits and can be omitted for initial generation. The compiler coordinates checks, control flow, and termination in accessible source, a wrapper and its assets, or a new program, according to the access setting.

Development execution links scores and trajectories to their runtime. *Runtime-source feedback* exposes permitted implementation details to subsequent reflection, helping diagnose gaps between design and execution. *Compiler evidence* guides implementation and evidence-supported plan adjustments during realization. The two channels thus inform behavioral revision and compilation at their respective stages.

3.4 Experience-Guided Search and Patient Selection

Accumulate experience and guide attention. *Update history* records revisions, outcomes, and selection decisions. *Mechanism Memory* retrieves evidence and reuse guidance from successful and rejected candidates. Optional *Slow Update* compares historical and current executions of the same tasks to identify recoveries and regressions. These records guide the optimizer. An *adaptive edit curriculum* shifts attention between workflow exploration and component refinement according to progress; a revision can update both layers under their proposal budgets.

Evaluate candidates while retaining the best realization. The shared-development protocol pools training and validation items for candidate scoring and samples saved current-runtime executions for reflection. A fixed-split protocol separates evidence and selection partitions. Candidate scores estimate Eq. (1) under the experiment's protocol and family weights.

Patient selection keeps a *current* pair for refinement and a separate *best* evaluated pair. Improvement beyond the configured threshold updates both. Otherwise, candidates within a score band relative to best may advance current for a bounded patience window; ties consume this allowance without replacing best. Exhausting the window, or leaving the band during pending exploration, restores best. Other out-of-band candidates are rejected. Reflection receives current with its matching traces, while candidate outcomes feed the configured experience mechanisms.

Final evaluation runs the retained, frozen R_{best} on independent test tasks, with no recompilation or test-driven updates. Appendix A.1 specifies the procedure; Section 4.2 examines its representation and search controls.

4 Experiments

Our experiments examine harness optimization under black-box, white-box, and from-scratch access across seven task domains. Tables 1–5 cover system performance, representation and design ablations, transfer, and multi-benchmark reuse. Appendices C and D provide the benchmark, model, and evaluation protocols.

4.1 System-level Results

Table 1 shows strong performance across the three access settings. With Codex/GPT-6, HarnessOpt reaches 97.12% on DABStep, compared with 70.61% for the native agent. With Claude Code/Sonnet 5, it raises WebShop success from 45.36% to 65.00%. White-box and from-scratch results extend these gains to editable and newly generated harnesses: with Opus 5, HarnessOpt reaches 83.33% and 78.57% on DeepPlanning Shopping, respectively. These results cover both adaptation of deployed agents and construction of task-specific runtimes; comparison protocols appear in Appendix D.3.

Table 1: System-level success (%) across three access settings. Bold and underline mark the best and second-best reported scores within each block.

Method / setting	AppWorld	DABStep	DP Shop	DP Travel	GAIA2 Mini	WebShop	TB 2.1
Black-box — fixed native hosts							
Codex / GPT-6 Astra							
Native	87.35	70.61	76.98	78.57	63.75	59.64	83.90
HarnessOpt	**92.14**	**97.12**	**78.17**	**84.52**	**65.00**	**62.14**	**84.32**
Codex / GPT-5.5							
Native	80.17	74.44	67.86	71.43	53.12	47.14	77.42
LongHorizon	90.09	70.93	55.95	66.67	16.88	55.36	77.05
HarnessOpt	**94.36**	**95.85**	**79.76**	**80.95**	**60.00**	**61.07**	**83.87**
Claude Code / Opus 5							
Native	93.16	59.42	65.48	80.95	58.75	55.71	78.69
LongHorizon	90.43	60.06	64.29	80.95	39.38	**63.57**	**85.00**
HarnessOpt	**93.50**	**94.89**	**80.95**	**94.05**	**65.62**	60.00	83.61
Claude Code / Sonnet 5							
Native	91.28	49.52	53.57	57.14	53.75	45.36	68.85
SkillOpt	91.62	89.14	55.95	55.95	41.25	48.93	**75.81**
HarnessOpt	**93.33**	**92.01**	**64.29**	**72.62**	**56.88**	**65.00**	75.41
Pi / GPT-5.5							
Native	83.59	78.91	66.67	79.76	51.88	48.21	80.49
HarnessOpt	**94.53**	**89.46**	**80.95**	**80.95**	**61.25**	**50.00**	**82.93**
Pi / Sonnet 5							
Native	89.06	45.37	54.76	51.19	45.00	47.86	78.05
HarnessOpt	**89.57**	**95.21**	**76.19**	**82.14**	**48.12**	**55.71**	**79.27**
White-box — editable harness code							
GPT-5.5							
Baseline	87.69	82.43	69.05	48.81	41.25	54.29	75.81
Meta-Harness (Pi variants)	91.97	88.82	72.62	78.57	58.13	**57.86**	77.42
MemoHarness (Pi extension)	90.43	85.62	64.29	71.43	34.55	52.48	80.65
HarnessOpt	**95.04**	**95.21**	**83.33**	**83.33**	**63.13**	55.36	**83.87**
Pi-based systems / Opus 5							
Baseline	93.85	58.15	64.29	67.86	68.13	61.79	79.03
HarnessOpt	**94.70**	**90.10**	**83.33**	**90.48**	**79.38**	**65.71**	**83.87**
From-scratch and standalone baselines							
GPT-5.5							
Baseline	88.21	61.02	66.67	29.76	41.25	50.00	72.58
Life-Harness	46.84	81.79	64.29	**69.05**	51.82	52.72	73.77
MemoHarness	88.55	70.61	67.86	50.00	23.64	54.14	**88.71**
HarnessOpt	**92.65**	**88.82**	**78.57**	64.29	**56.25**	**54.50**	75.81
From-scratch / Opus 5							
Baseline	93.50	59.42	63.10	70.24	35.63	60.00	62.90
HarnessOpt	**94.02**	**86.58**	**78.57**	**88.10**	**76.25**	**64.29**	**77.42**

4.2 Ablations

Kernel representation. Table 2 compares Kernel and direct-code optimization across two models, two access settings, and five domains. Kernel editing scores higher in 16 of 20 comparisons and ties in one. With GPT-5.5 in the white-box setting, it improves on direct code across all five domains, including 63.13% versus 57.50% on GAIA2 Mini.

Table 2: **Kernel representation ablation.** Success rates (%).

Variant	AppWorld	DABStep	DP Shop	DP Travel	GAIA2 Mini
White-box — GPT-5.5					
Baseline	87.69	82.43	69.05	48.81	41.25
Kernel	**95.04**	**95.21**	**83.33**	**83.33**	**63.13**
Direct code	91.26	92.33	79.76	78.57	57.50
White-box — Opus 5					
Baseline	93.85	58.15	64.29	67.86	68.13
Kernel	**94.70**	**90.10**	**83.33**	**90.48**	79.38
Direct code	91.97	88.82	76.19	**90.48**	**81.88**
From-scratch — GPT-5.5					
Baseline	88.21	61.02	66.67	29.76	41.25
Kernel	**92.65**	**88.82**	**78.57**	64.29	**56.25**
Direct code	91.97	76.04	71.43	**76.19**	37.50
From-scratch — Opus 5					
Baseline	93.50	59.42	63.10	70.24	35.63
Kernel	**94.02**	86.58	**78.57**	**88.10**	**76.25**
Direct code	93.85	**93.61**	76.19	86.90	71.25

Workflow and search design. The default configuration in Table 3 matches or exceeds the listed controls on all four domains. On DeepPlanning Travel, the default scores 80.95%, compared with 69.05% for component-only edits and 65.48% without external orchestration. The pattern supports refining global workflow together with local components. Appendix D.5 gives additional controls and their protocols.

Table 3: **Black-box configuration design.** GPT-5.5 success rates (%).

Variant	AppWorld noninteractive	DABStep	DP Shop	DP Travel
Default (reported reference)	**77.95**	**95.85**	**79.76**	**80.95**
Components only	58.29	94.57	78.57	69.05
No external orchestration	56.41	95.53	78.57	65.48
No mechanism memory	68.03	95.53	77.38	72.62
No selection/infra memory	73.50	**95.85**	77.38	70.24
Fixed train/validation split	75.21	91.05	78.57	72.62
Greedy selection	71.28	95.53	78.57	73.81
Fixed curriculum	76.58	88.18	77.38	69.05

4.3 TRANSFERABILITY

Table 4 highlights successful reuse across benchmarks, models, agents, and simultaneous model–agent changes. AppWorld transfer to GPT-5.4-mini/Pi raises success from 23.08% to 75.21%. A DABStep package transferred from Pi to Codex reaches 95.53%, compared with 79.87% for the target's native reference. These examples show that learned harness behavior can remain useful beyond its original configuration. Appendix D.10 reports the complete inventory and protocols.

Table 4: **Positive transfer results across four axes.** Selected reported gains relative to each target's native reference.

Task / transfer		Base (%)	Transfer (%)	Gain (pp)
(a) Cross-benchmark				
Fixed: GPT-5.5; Cx unless marked Pi.				
DP-Shop → WebShop (Pi)		46.25	55.25	**+9.00**
TB2.1 → DABStep		74.44	77.64	**+3.19**
TB2.1 → DP-Travel		71.43	79.76	**+8.33**
AppWorld → DP-Shop		67.86	72.62	**+4.76**
(b) Cross-model				
Source: GPT-5.5; exceptions below.				
AppWorld	mini/Pi	23.08	75.21	**+52.14**
AppWorld	nano/Pi	17.09	74.02	**+56.92**
DABStep	mini/Cx	76.68	94.25	**+17.57**
LiveSQL	mini/Cx	51.89	54.17	**+2.27**
WebShop	Opus 5/CC	55.71	65.00	**+9.29**
DP-Shop	27B/Cx	58.33	65.48	**+7.14**
DP-Travel	Max/Cx	22.62	83.33	**+60.71**

Task / transfer		Base (%)	Transfer (%)	Gain (pp)
(c) Cross-agent				
Fixed: GPT-5.5 and benchmark.				
DABStep	Cx→Pi	81.47	94.89	**+13.42**
DABStep	Pi→Cx	79.87	95.53	**+15.65**
DABStep	Pi→OC	81.47	95.53	**+14.06**
DP-Shop	Pi→OC	67.86	80.95	**+13.10**
DP-Travel	Pi→Cx	28.57	77.38	**+48.81**
(d) Cross-model and agent				
Source: GPT-5.5/Cx; exceptions below.				
WebShop	GPT-5.5/Cx	47.14	61.07	**+13.93**
DP-Shop	Astra/Cx	77.38	78.57	**+1.19**
DP-Shop	27B/Pi	52.38	63.10	**+10.71**
AW Normal	27B/CC	76.79	96.43	**+19.64**
AW Normal	Max/Pi	48.81	95.83	**+47.02**
TB2.1	Max/CC	72.58	80.65	**+8.06**
DP-Travel	Max/Pi	73.81	85.71	**+11.90**

Cx: Codex; CC: Claude Code; OC: OpenClaw. mini/nano: GPT-5.4-mini/nano; 27B/Max: Qwen3.8-27B/max; Astra: GPT-6 Astra.

4.4 Improving Pi Across Benchmarks

Table 5 evaluates Improved Pi-5.5, a shared white-box Pi harness compiled from expert Kernels for seven benchmarks. Mean success increases from the native reference's 65.62% to 76.02%, with gains on six of seven benchmarks. DABStep rises from 82.43% to 94.57%, and GAIA2 Mini from 41.25% to 72.50%. Appendix D.8 describes this shared-harness evaluation.

Table 5: Improved Pi-5.5 across seven benchmarks. Success rates (%).

Variant	AW	DAB	DP Shop	DP Travel	GAIA2 Mini	WS	TB	Avg.
Pi-5.5 — White-box								
Native Pi	87.69	82.43	69.05	48.81	41.25	**54.29**	75.81	65.62
HarnessOpt	**89.74**	**94.57**	**72.62**	**67.86**	**72.50**	53.21	**81.67**	**76.02**

4.5 Kernel Structure Study

Table 6 connects task performance to the structure of five paired Pi-agent runtimes. Kernel editing adds fewer host functions (F_{add}), logical lines (ΔL), and decision points (ΔD) in every pair, while achieving 2.9–5.6 percentage points higher test success. Across the five domains, it adds 10 functions and 209 logical lines, compared with 234 functions and 4,023 lines under direct runtime editing. Task-group analyses and execution cases are presented in Appendices E and F.

Table 6: **Host-code growth.** Structural entries are Kernel/direct runtime; Δ success is the paired difference in percentage points.

Benchmark	n_{test}	F_{add}	ΔL	ΔD	Δ success (pp)
Travel	84	0 / 66	62 / 974	0 / 311	+4.76
DABStep	313	0 / 40	6 / 509	0 / 119	+2.88
GAIA2	160	5 / 49	85 / 1,183	23 / 306	+5.63
AppWorld	585	4 / 30	37 / 464	16 / 120	+3.42
Shopping	84	1 / 49	19 / 893	1 / 382	+3.57
Total	–	**10 / 234**	**209 / 4,023**	**40 / 1,238**	–

Higher success with less host-code growth. The structural difference appears across task domains. On Travel, Kernel editing adds 62 logical lines and no decision points, compared with 974 lines and 311 decision points under direct runtime editing, while improving success by 4.76 percentage points. On AppWorld, the corresponding line counts are 37 and 464, with a 3.42-point success advantage. GAIA2 shows the largest success difference, 5.63 points, alongside 85 versus 1,183 added logical lines. Shopping adds one decision point versus 382, with a 3.57-point success advantage. DABStep gains 2.88 points with six added logical lines, compared with 509 under direct runtime editing. The pattern spans planning, application use, data analysis, and interactive assistance, linking the representation choice to how task-solving behavior is realized in each selected runtime.

Across the five pairs, the summed additions are 95.7% lower for functions, 94.8% lower for logical lines, and 96.8% lower for decision points under Kernel editing. These measurements concern added host code and complement the performance comparisons in Table 2. The ablation examines success across models and access settings; the structural study examines how selected Pi runtimes implement their behavior. Together, they support an optimization interface that keeps behavioral choices explicit while letting the compiler coordinate their realization. A useful behavioral revision can therefore be reflected in improved task outcomes without accumulating a comparably large set of additional host functions or control-flow branches.

5 Discussion

The Harness Kernel provides a persistent design interface between behavioral decisions and executable implementations. This interface gives reflection a coherent object to revise and gives the compiler a specification for coordinating workflow and component changes. The representation comparisons and artifact studies illustrate the value of this separation: strong task performance can accompany a more concise host implementation.

The same interface supports external adaptation, source refinement, and runtime generation, making harness optimization applicable across different access boundaries. Extending this approach to broader task mixtures and execution environments is a natural direction for future work. Appendix H discusses experimental scope, attribution, and reproducibility.

6 Conclusion

We introduced HarnessOpt, a framework for optimizing complete agent harnesses through a persistent behavioral Kernel. The Kernel connects global workflow and local components, allowing bounded design revisions to produce coordinated changes across code and supporting assets. Reflection, compilation, execution feedback, and experience-guided selection form one refinement process for black-box adaptation, white-box optimization, and from-scratch generation. Experiments demonstrate strong task performance and successful transfer, while paired artifact studies show higher success with less host-code growth. Harness Kernels offer a practical interface for systematic, whole-harness optimization.

AI USE STATEMENT

AI tools assisted with outlining, drafting, revising, and checking the manuscript; literature and source-record organization; and supporting code, configurations, and presentation scripts. These included Codex and Claude Code, with Claude Fable 5 used for technical and writing-consistency review. AI assistance also supported preliminary illustration development; the final artwork was professionally vectorized. The authors direct the research and presentation and remain responsible for the final claims, citations, results, and figures. Models used as experimental students, optimizers, compilers, reviewers, and judges are documented separately in the implementation and evaluation protocols.

REPRODUCIBILITY STATEMENT

Appendix A specifies the optimization algorithm, selection rule, and prompt interfaces. Appendices B and C document model roles, execution settings, benchmark splits, and scoring; Appendix D details baseline scope, artifact freezing, failure handling, and reporting. The supplementary artifacts include frozen prompt templates, ordered reference task IDs and hashes, protocol metadata, and a result-record index. The result index links reported values to cohort-specific protocols and available source records; the study artifacts identify evaluated runtimes and paired task outcomes. Appendix H discusses the reproducibility scope of historical and hosted-model evaluations.

REFERENCES

Lakshya A. Agrawal, Donghyun Lee, Shangyin Tan, Wenjie Ma, Karim Elmaaroufi, Rohit Sandadi, Sanjit A. Seshia, Koushik Sen, Dan Klein, Ion Stoica, Joseph E. Gonzalez, Omar Khattab, Alexandros G. Dimakis, and Matei Zaharia. optimize_anything: A universal API for optimizing any text parameter. *arXiv preprint arXiv:2605.19633*, 2026. URL https://arxiv.org/abs/2605.19633.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024. URL https://arxiv.org/abs/2408.08435.

Yue Huang, Wenjie Wang, Han Bao, Yuchen Ma, Xiaonan Luo, Yi Nian, Haomin Zhuang, Zheyuan Liu, Yue Zhao, and Xiangliang Zhang. MemoHarness: Agent harnesses that learn from experience. *arXiv preprint arXiv:2607.14159*, 2026. URL https://arxiv.org/abs/2607.14159.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.03714.

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. URL https://arxiv.org/abs/2603.28052.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026. URL https://arxiv.org/abs/2604.25850.

Ziyu Ma, Hailang Huang, Shun Zou, Yong Wang, Shidong Yang, Yiming Hu, Fei Wei, and XiangXiang Chu. LongHorizon-Harness: Advancing long-horizon agents for real-world tasks. *arXiv preprint arXiv:2608.01964*, 2026. URL https://arxiv.org/abs/2608.01964.

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. *arXiv preprint arXiv:2406.11695*, 2024. URL https://arxiv.org/abs/2406.11695.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026. URL https://arxiv.org/abs/2603.25723.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. AutoSaddler: Automatic harness optimization with durable updates from agent execution traces. *arXiv preprint arXiv:2608.23041*, 2026. URL https://arxiv.org/abs/2608.23041.

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. *arXiv preprint arXiv:2407.18901*, 2024. URL https://arxiv.org/abs/2407.18901.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. *arXiv preprint arXiv:2409.07429*, 2024. URL https://arxiv.org/abs/2409.07429.

Siwei Wu, Jincheng Ren, Yizhi Li, Haau-Sing Li, Chengran Yang, Yuxuan Zhang, Weicheng Gu, Jian Yang, Riza Batista-Navarro, Chuanyi Zhang, Xianglong Liu, Ming Zhou, Bryan Dai, and Chenghua Lin. ModularRSI: Modular and generalizable recursive harness self-improvement. *arXiv preprint arXiv:2609.14857*, 2026. URL https://arxiv.org/abs/2609.14857.

Tianshi Xu, Huifeng Wen, and Meng Li. Adapting the interface, not the model: Runtime harness adaptation for deterministic LLM agents. *arXiv preprint arXiv:2605.22166*, 2026. URL https://arxiv.org/abs/2605.22166.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026. URL https://arxiv.org/abs/2605.23904.

Ruiqing Yue, Yu Cui, Zhuoyu Sun, Sicheng Pan, Xianhong Xue, Tingyu Li, Ting Li, Wenzhuo Zhu, Yi Chen, Yifei Liu, Baohan Huang, Zhe Cui, Haibin Zhang, and Cong Zuo. Ecdysis: Efficient and effective training of runtime harnesses for LLM agents. *arXiv preprint arXiv:2609.11677*, 2026. URL https://arxiv.org/abs/2609.11677.

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. URL https://arxiv.org/abs/2406.07496.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026a. URL https://arxiv.org/abs/2606.09498.

Luan Zhang, Ruochen Zhou, Dandan Song, Zhengyu Chen, Yuhang Tian, Jun Yang, Huipeng Ma, Chenhao Li, Guangyuan Feng, Xudong Li, Yizhou Jin, and Yan Xu. HarnessCompass: Guiding automatic harness evolution toward generalizable and effective agent harnesses. *arXiv preprint arXiv:2608.01918*, 2026b. URL https://arxiv.org/abs/2608.01918.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025. URL https://arxiv.org/abs/2510.04618.

SUPPLEMENTARY MATERIAL: CONTENTS

A Optimization Procedure and Prompt Interfaces **14**

A.1 Reference Algorithm and Development Data Roles . . . 14

A.2 Typed Proposals and Kernel Integration . . . 15

A.3 Compiler and Review Contracts . . . 15

A.4 Key Prompt Excerpts . . . 16

B Experimental Implementation and Model Access **17**

B.1 Access Settings and Initialization . . . 17

B.2 Optimization Configuration . . . 18

B.3 Compilation and Execution Checks . . . 19

B.4 Model Identities, APIs, and Environment Versions . . . 19

C Benchmarks, Splits, and Scoring **19**

C.1 Benchmarks and Models . . . 19

C.2 Development and Held-out Populations . . . 20

C.3 Benchmark Interfaces and Metrics . . . 20

C.4 Dataset Revisions and Supplementary Populations . . . 21

D Evaluation, Baselines, and Reporting **21**

D.1 Baselines and Evaluation Protocol . . . 21

D.2 Selection and Frozen Evaluation . . . 22

D.3 Scope of the System Comparison . . . 22

D.4 Comparisons and Intervention Definitions . . . 22

D.5 Additional Ablation Results . . . 24

D.6 Earlier Campaign References for Feedback Ablations . . . 25

D.7 Metrics, Failures, and Uncertainty . . . 26

D.8 Multi-benchmark Shared-Harness Evaluation . . . 26

D.9 Transfer, Resources, and Reproducibility Artifacts . . . 26

D.10 Complete Transferability Results . . . 26

E Learned Workflows and Task Behaviors **30**

E.1 Which Tasks Benefit Most? . . . 30

E.2 Task-Group Evaluation Scope . . . 31

E.3 Learned Workflows . . . 31

E.4 Paired Task Executions . . . 33

F Runtime Structure and Execution Details **35**

F.1 Kernel Editing and Host-Code Measurements . . . 35

F.2 How Do Local Repairs Interact with Global Review? . . . 36

F.3 Review and Repair Control Flow . . . 37

F.4 Task-Level Comparisons . . . 38

F.5 Code Example: DABStep 38
F.6 Additional Code Example: Shopping 39

G Initialization and Workflow Evolution **40**
G.1 How Does Kernel Initialization Affect Optimization? 40
G.2 What Distinguishes Harnesses with Similar Workflows? 41
G.3 Can Optimization Introduce a New Repair Workflow? 41

H Limitations **42**

A Optimization Procedure and Prompt Interfaces

This section specifies the shared-development optimization loop, its selection rule, and the instructions connecting reflection, Kernel updating, compilation, and review. The pseudocode summarizes the executable implementation; the prompt excerpts reproduce text from the frozen source templates. The accompanying `supplement_assets/prompts/` directory contains the full template sources, their source revisions, and a SHA-256 manifest. The configuration of a particular run determines which backend and optional stages are active.

A.1 Reference Algorithm and Development Data Roles

An evaluated state $A = (K, R, E, q)$ stores the Kernel, its executable realization, the corresponding scored development executions, and the selection score. Keeping these objects together ensures that reflection reads traces from the runtime represented by the current state. In `full_dev_pool` mode, $D_{\text{dev}} = D_{\text{train}} \cup D_{\text{val}}$ supplies both reflection evidence and candidate scores. Each candidate is scored on the entire development pool; the reflection sample is a subset of its saved executions. In `fixed_split` mode, reflection instead uses training executions and selection uses the separate validation partition. Final held-out tasks have no role in either operation.

Listing 1: Shared-development HarnessOpt with patient selection. Functions summarize backend-specific executable stages.

```
Inputs: initial (K0, R0), development tasks D, held-out tasks T;
        search schedule; proposal quotas; patience P;
        score-drop allowance d; item allowance m; improvement e.
Evaluate(A) := execute A.R on all D; attach executions E and score q.
current = best = Evaluate((K0, R0)); pending = 0
history = []; memory = initialize_configured_memory()
for each optimization opportunity in the configured schedule:
    evidence = sample_saved_executions(current.E, configured_seed)
    context = configured_history_and_memory(history, memory)
    proposals = Reflect(current.K, evidence, context,
                        permitted_runtime_source=current.R)
    selected = DeduplicateAndSelect(proposals, proposal_quotas)
    if selected is empty: record_no_update(); continue
    Knew = Integrate(current.K, selected)
    Rnew = Realize(Knew, seed=current.R, reference=current.K,
                   permitted_development_evidence=evidence)
    qualify_and_review_using_configured_backend(Knew, Rnew)
    if artifact_is_invalid: record_rejection(); continue
    candidate = Evaluate((Knew, Rnew))
    band = max(d, m / len(D)) if score_is_hard else d
    if candidate.q > best.q + e:
        current = best = candidate; pending = 0
    else if candidate.q + 1e-12 >= best.q - band:
        if pending + 1 <= P:
            current = candidate; pending += 1
        else:
            current = best; pending = 0
    else if pending > 0:
        current = best; pending = 0
    else:
        retain current and best
    record_candidate_outcome_and_selection(history, candidate)
    update_configured_memory(memory, candidate, selection_outcome)
freeze best.K and the complete evaluated best.R
return execute_frozen(best.R, T)
```

Evidence sampling. The shared-development sampler excludes records classified as infrastructure failures from behavioral reflection. It shuffles the eligible success and failure groups using the configured seed, targets approximately 75% failures and 25% successes, and fills any unused quota from the remaining eligible records. Failure-only sampling is separately configurable. With repeated

evaluation, unstable outcomes and stable failures are interleaved before sampling. These choices affect reflection evidence; they do not replace the full development pool used to score a candidate.

Exact selection semantics. The score band is relative to *best*, rather than the preceding candidate. For a hard score over n development items, its width is $\Delta = \max(d, m/n)$; for a soft score it is d. A new best requires the strict inequality $q' > q_{\text{best}} + \epsilon$. A tie may advance current within the patient window but does not replace best, and it consumes one pending step. Exhausting the window restores the retained best state and its matching traces. An out-of-band candidate triggers the same rollback during pending exploration; otherwise it is rejected while the existing state is retained. Thus the final artifact can be the initial artifact if no eligible improvement is found. Immediate selection and a disabled gate are separate experimental controls.

Schedule and repeats. The implementation indexes optimization opportunities by epochs and steps, with the seed schedule recorded in the run configuration. In shared-development mode, an epoch groups these opportunities; it does not imply a new full pass through distinct reflection items. Selection repeats specify repeated task executions for a candidate score, rather than independent optimization seeds. The final evaluation executes the retained runtime tree, including its helpers and configuration, without recompiling its Kernel.

A.2 Typed Proposals and Kernel Integration

The CLI-based teacher writes `typed_proposals.json` after inspecting a Kernel snapshot and a development-evidence minibatch. Each proposal records its edit category, operation, evidence identifiers, affected paths, trace-supported problem, intended change, expected effect, complexity assessment, and regression risk. Operations are `create`, `update`, `compress`, `merge`, `split`, `replace`, and `delete`. The supported edit categories depend on the host interface: the external-wrapper path admits external orchestration, internal orchestration, and components; the Pi and standalone Python paths use internal orchestration and components. The selected host contract fixes the available execution capabilities.

The typed aggregation path merges proposals with the same category, operation, affected-path key, and normalized title (or proposal text fallback), accumulating their support and evidence identifiers. Within each category it orders proposals by decreasing support count, increasing declared risk, and title, then applies the configured category quotas. Component creation and maintenance have separate allowances; in this path the maintenance allowance is the configured component budget multiplied by the maximum component count. A nonpositive category limit means no truncation in the corresponding helper. The typed proposals are already selected and materialized before the trainer's generic selection stage, so they pass through that stage without a second ranking call.

The updater then integrates the selected proposals into a copied Kernel tree and writes a materialization report describing changed files, skipped proposals, and conflict resolutions. Its instructions distinguish an advisory component from a mandatory workflow gate: introducing the latter requires an orchestration proposal. This makes the adopted behavioral change the unit of control; a proposal may require edits in several files. Numerical proposal allowances are therefore not limits on changed lines, tokens, or semantic distance.

The alternative routed-chat path has separate architect, component, expansion, consistency-merge, and ranking prompts. These templates are included in the companion bundle and identified as a separate dispatch path. Their call graph and budget application are not substituted for the typed implementation.

A.3 Compiler and Review Contracts

Realization receives the current Kernel, the retained executable when available, the reference Kernel, an edit description, the active public interface, and any configured development-evidence bundle. In the evidence-revisable compiler policy, the Kernel supplies a default implementation plan. The compiler may follow, adapt, or omit a planned mechanism when the supplied evidence supports that choice, recording its rationale and runtime correspondence. The host contract specifies model, interface, access, and budget constraints. Development evidence is compiler-side input; the compiler is instructed to keep that evidence out of the solver's runtime assets and prompts.

When the candidate and reference Kernels have matching behavior hashes, the Pi identity path copies the seed runtime byte-for-byte without model-based compilation or review. Runtime loading and interface validation still apply. The structural study separately checks the seed artifacts; the identity path alone does not establish the provenance of historical initial-score reports.

Table 7: Backend-specific realization contracts. Access setting and execution backend are separate protocol fields; the two Pi interfaces expose different editable capabilities.

Realization path	Executable interface	Validation and review interface
External wrapper	Materialized `HARNESS.md` and assets; executable orchestration uses the host's declared entry point and node APIs.	Materialization checks and the configured materialization-review loop inspect the resulting tree before evaluation.
Pi extension	TypeScript default export registering handlers on the fixed Pi extension API.	Protected-input checks, runtime load validation, and the bounded compiler review apply to the compiled assets.
Pi agent program	`async run(runtime)` controls stages, agent/tool calls, and state under the supplied program API.	The program manifest and host inputs remain protected; review follows the complete-program contract.
Standalone Python	`runtime.py solve` owns control flow and uses `reflact_bridge` for configured model calls.	Kernel/evidence integrity, program structure, and Python syntax are checked; this compiler path does not invoke the Pi compiler-review task.

The Pi compiler reviewer is a separate task after compilation. It inspects source-to-runtime correspondence, host-interface use, conditions, retained state, effects, bounds, and fallbacks. It permits at most two local repair rounds, with an offline checkpoint after each changed asset snapshot. Rechecking an unchanged snapshot does not consume another repair round. Benchmark execution and live model calls inside this review task are prohibited; local tests using stubs and public interfaces are allowed. The reviewer records `passed`, `unresolved`, or `ambiguous`, along with inspected requirements, repairs, tests, and remaining issues. Unresolved semantic issues can proceed to development evaluation after the structural checks; a load check or review verdict is not a proof of behavioral equivalence. The compile-attempt limit is configurable; the review repair budget is fixed at two rounds in the inspected revision.

A.4 Key Prompt Excerpts

The following are contiguous excerpts from the frozen compiled-Kernel implementation, reproduced without rewriting. The reference Black-box package campaign uses an earlier typed-teacher and materialization-review revision, preserved separately in the bundle. The manifest identifies the exported source revisions; these templates are not a record of every instantiated historical request. The excerpts specify four consequential interfaces in the pipeline. CLI-based stages receive task instructions through their task files and invocation prompts; these are not necessarily API `system` messages. The routed-chat templates in the accompanying bundle are supplied through the system-message interface. Full source templates retain conditional backend instructions and run-time insertion points; actual task evidence and per-run instantiated requests are separate artifacts.

Reflection: inspect execution evidence. From `codex_typed.py`, `_teacher_task_text`:

```
Treat `evidence/minibatch.md` as an index, not a complete execution
    trace. Before making a non-empty proposal, first inspect the
    structured oracle summary and evaluator feedback for the samples
    that support it, then inspect their full raw traces. Always inspect
    the full raw traces selectively for the cited samples. Distinguish
    tests the student ran from the independent evaluator's result. Use
    both to verify what the student actually read, changed, executed,
    tested, and observed and why the evaluator accepted or rejected it;
    do not rely only on the student's final summary or a binary reward.
```

Kernel updating: reconcile proposals and report the update. From `codex_typed.py`, `_materializer_task_text`:

```
When several proposals conflict, choose the coherent change best
    supported by their evidence and record skipped proposals in the
    report. Finish by writing `materialization_report.json` at the
    workspace root with keys:
`created`, `updated`, `deleted`, `skipped`, `proposal_results`, `notes`.
```

Standalone compiler: implement the adopted plan. From `native_compiler.py`, `TASK_GUIDANCE`:

```
Implement the planned changes and evidence-supported implementation or
    strategy
corrections. Preserve successful behavior and reusable helpers. Treat
    the kernel
diff and edit intent as attention hints, not proof that the plan is
    correct.
Resolve contradictions using the supplied evidence or report uncertainty
    .
Record followed, adapted or omitted mechanisms, evidence, runtime
    locations and
public observable checks in out/COMPILE_NOTES.md. Keep kernel/ and
compiler_feedback/ read-only. Do not claim checks or score improvements
    that
were not observed.
```

Pi reviewer: inspect the implemented mechanisms. From `pi_compile_review.py`, `_task_text`:

```
Check executable triggers, conditions, retained state, actions, resets/
    bounds and
fallbacks against actual code. Check that advisory policies remain
    advisory.
Judge changes by the host contract, inspected evidence and intended
    effect.
Do not undo a supported adaptation merely because it differs from the
    kernel,
introduces an unstated strategy, or changes internal code organization.
    Check
that deviations have evidence and that successful behavior is preserved.
     Repair
concrete execution defects or unsupported changes; do not restore a
    contradicted
kernel rule simply to make the runtime match its text. Report source/
    runtime and
evidence locations for each issue. Do not begin an unrelated
    optimization search
or make stylistic changes. Report uncertainty when a conflict cannot be
    resolved
from the supplied evidence; do not claim a measured score improvement.
```

B Experimental Implementation and Model Access

B.1 Access Settings and Initialization

Table 8 defines the editable objects and inherited execution machinery for the three access settings. Main-table model names follow the reported evaluation configurations; optimizer, compiler, and evaluator are separate roles in linked run records.

Table 8: Execution boundaries. Fixed-host package adaptation, editable Pi programs, full-source edits, and standalone generation retain their respective scopes.

Setting	Editable object	Initialization and fixed mechanisms
Black-box	External harness package, wrapper, and supporting assets.	Native agent internals remain fixed. The reference campaign starts with an empty additional harness, retaining task adapters and public system inputs.
White-box	Existing harness code within an explicit editing boundary.	The documented Pi instantiation edits a program above a fixed core. Pi-source baselines edit a broader boundary and are identified separately.
From-scratch	Standalone runtime generated over model RPC and environment tools.	Reported runs start from a handwritten minimal Kernel; they do not inherit a complete Pi agent loop.

Kernel revisions can change both workflow rules and component assets. Black-box package materialization uses the native host interface and does not separately isolate a Kernel compiler. White-box representation comparisons require matching the editable boundary and initial executable.

B.2 Optimization Configuration

Table 9 gives the reference Black-box recipe; Appendix A.1 defines its development-data roles and selection rule.

Table 9: Documented reference configuration. The cohort variations below and result-specific test repeats take precedence over this recipe.

Parameter	Reference Black-box configuration
Search schedule / seed	2×6 update opportunities; seed 42
Development policy	Source train and validation partitions merged
Reflection size / minibatch	20 / 4
Reflection composition	Approximately 75% failures and 25% successes, with valid-sample fallback
Candidate selection	Entire development pool; current and best retained separately
Patient window	3
Allowed drop	Fraction 0.08; one-item allowance for hard metrics
Minimum improvement	0; strict improvement required
Proposal quotas	External orchestration 4; internal orchestration 8; component 8; expansion 8
Component scope	Up to four selected components; quota is per component
Global edit selection	Autonomous ranking without a global cap
Rollout / selection / test repeats	1 / 1 / 1 in the reference recipe

The reference package recipe enables rejected-candidate history and cross-step comparison. Update history, Mechanism Memory, and Slow Update have distinct inputs and separate configurations. Adaptive curriculum sets the editing focus; proposal quotas control the numerical allowances.

Cohort configurations. The compiled-Kernel feedback ablation uses seed 42, eight updates, and one final evaluation repeat. Mechanism Memory and Slow Update are disabled and the curriculum is fixed, so its history intervention tests recent update records. In the earlier audited Terminal-Bench snapshot, the four Kernel pairs freeze after 8/5/4/8 completed updates, selecting steps 8/4/3/2 for GPT-5.5 White-box, GPT-5.5 From-scratch, Opus White-box, and Opus From-scratch, respectively. These settings identify that development-selected snapshot; the September 26 summary does not provide new selection-step receipts for every updated aggregate. The archived Terminal-Bench recipe uses a 0.03 fractional drop allowance and reflection minibatches of six; the newer GPT evidence configuration uses all 27 development tasks and minibatches of four. The Kernel WebShop recipe uses reflection size 24, minibatch six, eight update opportunities, and a 100-action horizon. These configurations are separate from the twelve-update package recipe.

B.3 Compilation and Execution Checks

The Pi and standalone compilers default to two attempts and 900 seconds. Pi review uses the passed compiler model with a timeout of $\min(300\text{ s}, \text{compiler timeout})$; standalone compilation uses the configured teacher model. Backend-specific checks, repair limits, and review outcomes are specified in Appendix A.3. Runtime-source feedback informs the next Kernel update, while compiler development evidence informs realization; these are separate interventions in the feedback ablation.

B.4 Model Identities, APIs, and Environment Versions

Table 10 distinguishes runtime version records from configuration pins. Model roles are identified by requested model, provider route, API family/version, client, and reasoning settings; a compatible transport does not establish an immutable model snapshot.

Table 10: Version records. Configuration pins do not substitute for missing run receipts. Immutable model snapshots and complete client/image metadata are unavailable for some historical reports.

Component	Recorded version or configuration	Evidence scope
Codex native host	CLI 0.153.4	Reference runtime record
Pi source-editing baseline	Pi 0.84.3	Baseline report
GAIA execution environment	ARE 1.2.0	Reference runtime record
Azure Responses	`api-version=2025-04-01-preview`	Route templates and Kernel configuration
Native Anthropic client	`anthropic==0.125.0`	Source dependency pin
Student/optimizer identities	Model and effort specified per cohort; the reference package recipe uses the same logical student/teacher model with medium effort	Configuration policy
Compiler/reviewer identities	Pi review receives the compiler model; standalone compilation receives the configured teacher model	Backend implementation

Cohort metadata preserves requested aliases and provider routes, including enterprise Responses, Copilot compatibility, and route transitions. The accompanying API version notes record client fallback and version translation. Qwen transfer requests medium effort with transport-specific thinking parameters; nominal effort does not imply equal compute across models.

Reference adapter limits are 40 interactions for AppWorld, 100 actions for WebShop, and a 7,200-second outer timeout with a 2.0 timeout multiplier for Terminal-Bench. Individual system ports and transfer protocols can differ; the corresponding source, task-ID, prompt, and configuration records accompany their evidence.

C Benchmarks, Splits, and Scoring

C.1 Benchmarks and Models

The seven main-table domains are AppWorld (Trivedi et al., 2024), DABStep, DeepPlanning Shopping and Travel, GAIA2 Mini, WebShop, and Terminal-Bench 2.1. Their reference test populations contain 585, 313, 84, 84, 160, 280, and 62 tasks, respectively. AppWorld combines Normal and Challenge by task counts; DABStep uses local reconstructed-gold scoring. Historical and imported cohorts retain their own populations and protocols (Appendices C and D.3).

The six principal Black-box configurations pair Codex with GPT-6 Astra or GPT-5.5, Claude Code with Opus 5 or Sonnet 5, and Pi with GPT-5.5 or Sonnet 5. White-box and From-scratch results use GPT-5.5 and Opus 5, with their reported baselines. Pi/Opus 5 remains an explicitly named system reference. Model labels follow the reported evaluation configuration; teacher and compiler roles are recorded separately.

C.2 Development and Held-out Populations

Table 11 defines the main reference data protocol. The source train and validation partitions are merged into one development pool under `full_dev_pool`; reflection and checkpoint selection use that pool. The `fixed_split` alternative separates these roles and is identified when used. Test tasks are a distinct frozen population. Counts in this table describe the reference protocol; historical and study cohorts retain the populations listed in their result captions.

Table 11: Reference task populations. Dev is the unique merged train/validation population, not a fourth independent split. Exact ordered IDs accompany the supplementary artifacts.

Benchmark	Train	Val	Dev	Test	Split construction
AppWorld	90	57	147	585	Official partitions; 168 Normal + 417 Challenge
DABStep	89	45	134	313	Seed 42 over 447 locally scoreable tasks
DP Shopping	24	12	36	84	Level stratification; seed 42
DP Travel	24	12	36	84	English duration stratification; seed 42
GAIA2 Mini	47	23	70	160	Official Mini test; capability-balanced non-Mini dev
WebShop	80	40	120	280	Seed 42 within original goal partitions
Terminal-Bench 2.1	18	9	27	62	Category/difficulty balance; seed 42

Grouping and sampling. AppWorld retains official membership and scenario-family separation. Shopping levels 1 and 2 each contribute 10/5/35 train/validation/test cases, and level 3 contributes 4/2/14. Travel's six duration groups (2–7 days) each contribute 4/2/14. GAIA2 holds out all 160 Mini instances and samples 14 development instances per capability from non-Mini data, using universe-proportional allocation and seed-42 scenario-hash ordering; train/validation ordering uses seeds 43/44. WebShop reconstructs source goals with seed 233, then independently subsamples the original goal-index partitions with seed 42. GAIA universes can be shared across partitions; this protocol therefore holds out instances rather than entire universes.

C.3 Benchmark Interfaces and Metrics

AppWorld. The agent interacts with applications through the declared adapter, and private task checks determine hard success. Public tool results and execution errors are available during solving; private expected states and grader traces are excluded from the student's task inputs. Normal and Challenge evaluate one selected harness. Their combined score is $100(c_N + c_C)/(168 + 417)$, where c_N, c_C are success counts, rather than an unweighted mean of their percentages. The two partitions do not double the offline optimization cost.

DABStep. The local hard-task protocol uses reconstructed gold answers and a scorer with numeric tolerances, list normalization, and fuzzy textual comparisons. The score is local answer correctness, rather than an official hidden-test leaderboard result or simple exact string match. The frozen loader admits 447 scoreable instances from the 450-instance source; its task IDs and gold/scorer provenance define the reported population.

DeepPlanning Shopping. In official mode, success requires all expected product and coupon matches; extra predicted items are not automatically penalized. The implementation's `official_weighted_average_case_score` is the mean of per-case hard success. Item/coupon match rate is a separate quantity. The hardened mode additionally requires exact multiset matching and completion, and is kept separate. The current reference pair reports binary case counts; imported results retain their specified scorer mode.

DeepPlanning Travel. Hard success requires both commonsense and personalized constraint scores to equal one. The local adapter writes and scores the saved `travel_plan.json`; soft constraint scores are auxiliary diagnostics. The structured submission bypasses the public leaderboard's separate Markdown-to-JSON model conversion, while retaining the plan schema and constraint checks. Consequently, local and leaderboard scores have distinct submission protocols.

GAIA2 Mini. The ARE scenario validator supplies task success. JSON-action, native interaction, and unverified records remain separate in the result index. Notification, session, and judge configurations are part of this protocol. Judge identity is separate from the student: the reported LongHorizon

Claude route uses a GPT-5.5 judge, whereas the native SkillOpt Claude configuration uses its Claude model. These evaluator differences are retained when interpreting cross-system comparisons.

WebShop. Hard success requires a completed episode with reward at least $1 - 10^{-9}$; average reward is reported separately when used. The reference protocol retains the full product catalog and the source goal partitions. The reference population is distinct from WebShop-lite and the legacy 400/423-task evaluations. The linked GPT-5.5 From-scratch experiment uses 200 training, 100 validation, and 400 test tasks; its 280-task projection is a diagnostic of the same trained artifact, not retraining on the reference development split. The archived projection has disjoint task IDs but one shared development/test instruction hash. The current 54.50 result uses the full 400-task test.

An earlier WebShop-lite record uses compact rollouts and spaCy scoring, with 54.29 for both baseline and selected arms. The latest September 26 White-box GPT-5.5 report instead gives Baseline 54.29 and Kernel 55.36. We retain this revised aggregate without inferring its evaluation variant, task count, or artifact identity from the earlier record. Other unlinked Opus WebShop aggregates retain the same reporting qualification.

Terminal-Bench 2.1. Official verifier outcomes determine success on the selected tasks from the 89-task release. The earlier audited Kernel pairs retain all 62 task IDs, including failures and resource-exhaustion flags; those receipts bind their particular artifacts rather than every later report aggregate. Other cohorts use explicit 59- or 61-task populations, which are not interchangeable with the 62-task estimate. Repeat counts identify evaluation attempts separately from unique tasks.

C.4 Dataset Revisions and Supplementary Populations

Table 12: Revisions recorded by the reference split scripts and inspected grader. Scoring adapters also contain local compatibility checks.

Source	Revision
GAIA2 data	`78ea3bdbdeec2bdcd6afa5420915d8a22f23ed99`
WebShop data	`ce990fff5aee388db2706f07820c578ab68e0453`
WebShop source	`64fa2a5c15c7daa698b9ac93f5bb5437b634c9bd`

The compiled-Kernel feedback ablation uses 36 development and 84 test tasks for each DeepPlanning domain. Its DABStep and WebShop mini populations contain 36 development and 64 test tasks, drawn from the corresponding main partitions without cross-role task-ID overlap. Each mini development list contains 24 source-train and 12 source-validation members. This does not recover the original mini sampling order. AppWorld's 147/585 identities match the reference protocol. The extended study captions specify their historical populations, selected artifacts, and repeats independently.

D Evaluation, Baselines, and Reporting

D.1 Baselines and Evaluation Protocol

Native agents use fixed task/tool adapters. SkillOpt optimizes external skills, and LongHorizon supplies a workflow around its supported executor. In White-box comparisons, Meta-Harness edits accessible Pi source, MemoHarness supplies a Pi extension, and Self-Harness edits DeepAgents; the documented HarnessOpt runs edit programs above a fixed Pi core. From-scratch HarnessOpt generates a standalone runtime over model and tool primitives. Its comparison block also includes Life-Harness and ordinary MemoHarness as standalone baselines using their existing executors, plus the handwritten LongHorizon runtime reference.

Development outcomes guide optimization and checkpoint selection; the selected artifact is frozen for held-out evaluation. Source reports retain their recorded task populations, selection rules, and execution budgets. Some imported aggregates lack uniquely linked task manifests, and system budgets are not globally matched. Appendix D specifies these scopes and the available repeat and failure records.

D.2 Selection and Frozen Evaluation

As specified in Appendix A.1, development outcomes guide updates and checkpoint selection. The selected Kernel, executable, and configuration are frozen; final evaluation executes the retained runtime on separate test tasks without recompilation. Recompiling produces a new artifact.

A cohort binds an initial artifact, selected artifact, task manifest, scorer, model configuration, and execution budget. The supplementary result index records the material differences among imported cohorts: population sizes; GAIA interaction modes; historical or provenance-limited reports; and unchanged selected artifacts. Source reports for some imported Pi/GPT-5.5 and Pi/Sonnet 5 cells specify score-dependent batch-adoption rules. The affected cells are identified by the selection-qualification field in the supplementary result index. These observations carry cohort-selection uncertainty and are interpreted descriptively, separately from comparisons with documented selection freezes. Previously reported and negative pairs remain in the underlying result ledger.

Reported aggregates and linked runs. The Black-box matrix adopts the author-designated report percentages verbatim. A republished aggregate does not create a new evaluation repeat, optimization seed, or documented selection freeze. Some source reports select among earlier cohorts or repeated evaluations; those selection qualifications remain in the result index; previous snapshots remain in the source archive. Counts, repeats, and recovery status are retained only when the run linkage is explicit. In the detailed GAIA2 table accompanying this report, the GPT-6 R2 Native arm is 104/160 (65.00), whereas the R2 HarnessOpt arm is 98/160 (61.25). The summary instead labels 65.00 as HarnessOpt. We retain the designated summary value and leave its arm/count linkage unresolved; 104/160 is not relabelled as an optimized run.

D.3 Scope of the System Comparison

Table 1 combines reported system results with different populations and execution surfaces. Unlinked task counts remain unspecified; the benchmark reference population is not substituted for a run's denominator.

Reported TB rescalings. The Codex/GPT-5.5 entries 77.42 and 83.87 follow the latest report's round-to-62 conversion of historical 44/57 and 207/248 records. They do not represent newly observed 48/62 and 52/62 outcomes. The underlying repeat-four record labels 207/248 as baseline and 204/248 as its selected best; its integrity audit also records external same-task trajectory access in the baseline. We retain the designated report values as historical references with unresolved optimized-arm attribution, not as a measured optimization gain. Original counts, earlier displayed values, and qualifications remain in the result index.

D.4 Comparisons and Intervention Definitions

System comparisons require common task populations and scoring protocols. Attributing an effect to the optimizer or Kernel representation additionally requires matching hosts, editing permissions, initializations, and budgets apart from the intended intervention. Different judges, scorers, or populations provide protocol-specific context unless equivalence is established. Main-table system budgets are not globally equal.

Native agents use fixed task/tool adapters without an optimized additional harness. SkillOpt edits a skill on a fixed host. LongHorizon uses its stated adapter and at most 25 rounds per task. Life-Harness, MemoHarness, Meta-Harness, and Self-Harness retain their own executor and editing scope. In particular, MemoHarness is distinct from Meta-Harness; source-editing baselines have broader permissions than the fixed-core Pi-program Kernel runs. The MemoHarness controller can change reasoning effort; equal student labels therefore do not establish equal inference cost.

Package, compiled-Kernel feedback, and representation ablations retain their intervention scopes. The updated report's higher-value Reference rows are separate from the earlier own-Full controls in Appendix D.6. The feedback campaign's history intervention concerns recent update records; its other memory and curriculum settings are listed in Appendix B.

Additional execution surfaces. Table 15 reports three complete baseline tests whose execution surfaces differ from the existing main-table rows.

Table 13: Population and protocol differences in the system comparison.

Rows or benchmark	Scope retained in the comparison
GAIA protocols	Linked Native/HarnessOpt entries use native interaction; legacy or unverified entries remain separate. The Codex/GPT-6 Astra HarnessOpt value 65.00 has unresolved arm attribution.
GAIA populations	Linked standard runs use 160 tasks. Life-Harness, MemoHarness and MemoHarness/Pi use 110; Self-Harness evaluates 110 tasks twice. Kernel aggregate counts remain unspecified where no run receipt is linked.
WebShop populations	GPT-5.5 From-scratch Kernel uses 400 test tasks and 300 development tasks. Life-Harness, MemoHarness and MemoHarness/Pi use 423 test tasks. Most linked hosted-system results use 280. New unlinked aggregates, including the White-box GPT-5.5 value 55.36, are not assigned that denominator.
Codex/GPT-5.5 WebShop	HarnessOpt's 61.07 is 171/280 from a frozen Sonnet 5/Claude Code package transferred to GPT-5.5/Codex; it is not a new target-specific optimization run.
Terminal-Bench	Codex/GPT-6 Astra Native/HarnessOpt use 59 tasks with four repeats; Claude Code Native/HarnessOpt use 61. LongHorizon uses 61/60/60/61 for Codex GPT-5.5/GPT-6 Astra and Claude Opus/Sonnet, after authorized exclusions. Life-Harness uses 61; identified Kernel pairs use 62. Other unlinked aggregates retain unknown populations. Equal subset size does not establish common cross-method task IDs.
Repeated evaluation	Self-Harness repeats each task twice. Codex/GPT-6 Astra WebShop and Life-Harness native-interactive GAIA reuse their initial runtime; repeated-score differences are not learned changes.
Standalone baselines	Life-Harness adapts four hooks around an existing native executor. Ordinary MemoHarness uses an existing executor with frozen memory and task-time configuration adaptation; six Terminal-Bench tasks use higher reasoning effort. Both are standalone baselines, not generated runtimes. MemoHarness/Pi remains a White-box source extension.
Source variants	Meta-Harness/Pi combines benchmark-specific source variants, including a Pi-worker GAIA implementation. Kernel rows retain their author-reported scopes. Matching task counts alone does not match search or inference budgets.

Table 14: Ablation interpretation. Own-campaign controls and the report's cross-campaign Reference rows are identified separately; the latter provide descriptive comparisons.

Intervention	Meaning in its own campaign
Kernel / direct runtime	Changes the optimization representation; shared initial percentages alone do not prove identical artifacts or budgets.
Component-only / +workflow	Nested restrictions on ordinary package edits; components can include executable code.
No explicit workflow proposals	Restricts typed proposals; the compiler can still change control flow.
No runtime-source feedback	Removes additional source feedback from the updater, retaining other permitted evidence.
No compiler development evidence	Removes the designated compiler/reviewer evidence channel; the Kernel still evolves.
No update history	Removes recent edit/outcome records; learned behavior remains in the current Kernel/runtime.
Greedy selection	Changes the acceptance rule; best-checkpoint retention remains.
Fixed split	Separates reflection and selection data; it is a data-use intervention.

Table 15: Additional completed baseline results on distinct execution surfaces.

Method / model	Benchmark	Passed / n	Success (%)
SkillOpt / GPT-5.5	AppWorld	437/585	74.70
SkillOpt / GPT-5.5	DP-Shop	58/84	69.05
Meta-Harness / GPT-6 Astra	DP-Shop	59/84	70.24

SkillOpt uses a standalone direct-chat executor. Meta-Harness evaluates a generated runtime through Codex; its initial-runtime control is incomplete. Each test uses one evaluation per task.

Historical Pi variants and baseline recovery. The historical Meta-Harness/Pi AppWorld, DABStep, Shopping, and Travel results use source-editing campaigns with their recorded starts and search budgets. The GAIA2 result uses a direct Pi-worker variant with 160 test tasks and 20 search steps. These are system-level observations across distinct campaigns, not one matched optimization cohort. LongHorizon Terminal-Bench results evaluate authorized subsets of the original 62 tasks. These are complete subset reports; the excluded tasks and unmatched subset identities preclude treating them as common full-population comparisons.

D.5 Additional Ablation Results

Main-text design controls. Table 3 uses the reported default reference; its AppWorld column follows the historical noninteractive protocol. White-box feedback controls and partial AppWorld results are retained below.

The full five-domain representation comparison is consolidated in Table 2. The following controls retain their source campaigns and evaluation populations. The new From-scratch AppWorld outcomes are completion-selected partial observations and are reported separately from full-population representation results.

Table 16: Additional Black-box workflow and search controls.

Variant	DP Shop	DP Travel	DABStep	AW noninteractive	AW Pi	BIRD	TB2.1
Native	67.86	71.43	74.44	50.94	83.59	39.52	77.42
Report Reference	**79.76**	**80.95**	**95.85**	**77.95**	**94.53**	**42.86**	83.87
Components only (no IO or EO)	78.57	69.05	94.57	58.29	93.50	40.48	77.42
Components + IO (no EO)	78.57	65.48	95.53	56.41	**94.53**	40.00	74.19
w/o Mechanism Memory	77.38	72.62	95.53	68.03	**94.53**	40.48	74.19
w/o Selection/infra memory	77.38	70.24	**95.85**	73.50	92.48	39.05	**79.03**
Fixed train/validation split	78.57	72.62	91.05	75.21	93.85	38.10	**79.03**
w/o Patient gate (greedy)	78.57	73.81	95.53	71.28	**94.53**	41.43	**79.03**
Fixed curriculum	77.38	69.05	88.18	76.58	94.19	41.90	72.58

Success (%). DP uses 84 tasks per domain; DABStep 313, both AppWorld protocols 585 and BIRD 210. TB removal arms use 62 tasks; Native/Reference are the historical rescalings in Table 1, with unresolved optimized-arm attribution. References and removal arms come from distinct snapshots. IO/EO denote internal/external orchestration.

Table 17: Additional reported feedback and selection controls.

White-box feedback and selection — author report

Variant	DP Shop	DP Travel	DABStep mini arms	AppWorld 585	GAIA2 interactive
Report Reference	**85.71**	**83.33**	95.21	**95.04**	63.13
w/o Patient gate (greedy)	80.95	**83.33**	87.50	92.99	—
w/o Explicit update history	80.95	82.14	**93.75**	94.02	—
w/o Runtime source to updater	79.76	76.19	82.81	93.85	—
w/o Compiler development evidence	78.57	76.19	87.50	94.87	—

From-scratch feedback and selection — author report

Variant	DP Shop	DP Travel	DABStep mini arms	AppWorld partial, 195	GAIA2 interactive
Initial (partial AppWorld)	—	—	—	54.36	—
Report Reference	**78.57**	**64.29**	88.82	95.38	56.25
w/o Patient gate (greedy)	59.52	29.76	89.06	85.13	—
w/o Explicit update history	71.43	40.48	**90.62**	91.28	—
w/o Runtime source to updater	48.81	28.57	87.50	**95.90**	—
w/o Compiler development evidence	76.19	29.76	85.94	95.38	—

DP uses 84 tasks; DABStep references use 313 but removal arms use 64; White-box AppWorld uses 585 and GAIA2 references use 160. These reported references can select a higher Full across campaigns. The feedback campaign uses GPT-5.5 medium, one repeat, fixed curriculum and disabled Mechanism Memory/Slow Update.

From-scratch AppWorld is partial: each arm retains its own first 195 completed outcomes, comprising 117 previously completed tasks plus the earliest 78 new completions. Only 150 task IDs are common to all six conditions. Initial is 106/195 (54.36); the 95.38 reference is the partial Full under this same completion rule, not the historical 585-task reference. Unfinished tasks are neither zero-filled nor rerun. Greedy completes four training steps (selecting step 1) with test concurrency 128; the other four optimized arms complete eight steps with concurrency 64. These are descriptive partial observations, not paired or equal-budget ablations.

D.6 Earlier Campaign References for Feedback Ablations

Table 18 preserves the earlier frozen feedback campaign underlying the White-box feedback controls reported here. These own-Full references and 64-task WebShop results are separate from the report's higher-value references in Table 17. The earlier White-box Travel no-history replay scored 86.90; the updated report lists 82.14. Both records are retained with their snapshot identity rather than treated as one result.

Table 18: Earlier frozen feedback-ablation snapshot with each campaign's own Full. This archival panel is separate from the updated report references.

Earlier frozen campaign snapshot — own-Full controls

	White-box — Pi				From-scratch		
Variant	DP-Shop	DP-Travel	DAB mini	WebShop mini	DP-Shop	DAB mini	WebShop mini
Initial harness H_0	67.86	48.81	68.75	59.38	33.33	4.69	51.56
HarnessOpt (own Full)	**85.71**	79.76	84.38	59.38	66.67	79.69	45.31
w/o Runtime source to updater	79.76	76.19	82.81	50.00	48.81	87.50	51.56
w/o Compiler evidence	78.57	76.19	87.50	53.12	**76.19**	85.94	**53.12**
w/o Update history	80.95	**86.90**	**93.75**	54.69	71.43	**90.62**	51.56
w/o Patient selection	80.95	83.33	87.50	**65.62**	59.52	89.06	48.44

DP uses 84 tasks per domain; DABStep-mini and WebShop-mini use 64. One optimization seed (42), eight steps, one test repeat, and development-based selection. Provider-fault replays are recorded in the source; the White-box Shopping no-compiler-evidence result retains an unresolved fault.

D.7 METRICS, FAILURES, AND UNCERTAINTY

For binary success with n tasks and r evaluations of a fixed artifact, the mean score is $100(nr)^{-1}\sum_{i=1}^{n}\sum_{j=1}^{r} y_{ij}$. Unique-task count remains n; nr is the number of attempts. Best-of-r and average reward are different metrics. AppWorld Normal/Challenge are combined by counts; joint-domain averages weight domain scores as stated in the corresponding table.

Ordinary method failures, refusals with valid verifier outcomes, and budget exhaustion remain in the declared denominator. An execution flag can coexist with a scoreable submission, so scores and error flags are retained separately. Provider-request retries, recovery within a task, candidate repair, and repeated evaluation have different units. The reference runtime allows two request/stream retries and does not automatically resample tasks. Invalid infrastructure requests or grading failures require a documented qualification or recovery decision; incomplete and withheld results do not supply final scores.

Optimization seeds measure search variability; repeats of a fixed artifact measure conditional execution variability. Single-seed results do not establish multi-seed stability, and repeat-score differences for an unchanged runtime do not identify learned behavior. Case studies and directional comparisons are not significance tests. Reported task-level uncertainty preserves pairing but does not quantify search-seed or retrospective-selection uncertainty.

D.8 MULTI-BENCHMARK SHARED-HARNESS EVALUATION

Table 5 compares the author-designated native Pi references with a shared White-box harness compiled from expert Kernels. This composition is evaluated without further joint optimization updates. The baseline retains its per-benchmark source protocols and is not a matched shared-program initial test. White-box AppWorld is an author-reported external result, and Terminal-Bench uses 60 tasks. The mean weights the seven domains equally. Six benchmark scores improve; WebShop changes from 54.29% to 53.21%.

D.9 TRANSFER, RESOURCES, AND REPRODUCIBILITY ARTIFACTS

Frozen package transfer reuses the learned package on its target; it is distinct from recompiling a Kernel for a new host. Simultaneously changing model and host does not isolate either transfer axis. The completed Qwen matrix contains 24 paired target configurations, with 14 improvements, one tie, and nine declines. The Travel time-budget follow-up is a separate, test-informed adaptation and is not substituted for the original frozen transfer results. Joint optimization on seen domains is distinct from unseen-suite evaluation, and expert-informed initialization is prior information. Incomplete joint tests remain unfilled.

Offline costs include initialization, reflection, compilation, review, repair, rejected candidates, and development evaluation; deployment costs include solver/tool calls, tokens, latency, and failures. Equal steps do not match exposure or expenditure. Missing usage is not zero, and each trained artifact incurs one optimization cost across its evaluation partitions. The scores do not establish a uniform cost-efficiency advantage.

The supplementary artifact bundle includes ordered reference task IDs and hashes, protocol metadata, source prompt excerpts/templates, algorithm descriptions, a main-table record index, and a run-record schema. A complete run record links its source/configuration, role identities, scorer/environment, initial and selected artifact hashes, selection decision, task-level scores, repeats, and usage. Version and artifact coverage for historical results is stated at its available level; code defaults are not used to reconstruct missing run settings.

D.10 COMPLETE TRANSFERABILITY RESULTS

Main-table transfer configurations. AppWorld Normal uses 168 tasks. WebShop in panels (b,d) uses a Sonnet 5/Claude Code donor; DP-Shop to Astra/Codex uses Opus 5/Claude Code. DP-Travel Max/Codex and Max/Pi use the 1,800-second budget-adaptation follow-up, with different execution costs.

Table 4 presents 23 selected positive results across benchmarks, models, agent harnesses, and simultaneous model–agent changes. Each result retains its target's native reference. The two displayed Qwen-Max Travel results come from a separate 1,800-second budget-adaptation follow-up and do not establish equal-cost gains. Transfer is not uniformly beneficial: the original frozen Qwen matrix has 14 improvements, one tie, and nine declines across 24 configurations. Appendix D.10 retains all 56 rows, including unsuccessful transfers and the distinct budget-adaptation results. These package transfers do not establish Kernel recompilation across hosts.

Table 19 evaluates frozen harness packages under changes in model, execution harness, or benchmark. Panels (a–c) separate the three axes; panel (d) changes model and harness together, and panel (e) reports a distinct budget-adaptation follow-up. Historical rows retain their August or September cohort and denominator. The transferred object is a HarnessSkill/package, rather than a demonstrated Kernel-recompilation route. Each target retains its own original-harness reference, population and protocol.

What transfers, and what fails? The original frozen-transfer Qwen matrix contains 24 paired target configurations: 14 improve, one ties and nine decline. AppWorld Normal improves in all six targets, whereas DP-Travel declines in all six. These results show that useful procedures can transfer, but their orchestration budgets and interface assumptions need not remain suitable for a new target. Its execution audit covers all 24 pairs; the budget-adapted outcomes are reported separately below.

Budget adaptation is a separate experiment. The original Travel transfer uses a 900-second planning stage and a fresh repair session, versus a 1,800-second original-harness session. The follow-up raises planning and repair limits to 1,800 seconds and the total orchestration limit from 3,600 to 7,200 seconds. The updated report lists four completed 84-task arms, with transfer success of 83.33 for Qwen3.8-max/Codex, 70.24 for Qwen3.8-27B/Claude Code, 85.71 for Qwen3.8-max/Pi and 77.38 for Qwen3.8-max/Claude Code. Panel (e) displays these results alongside the original 900-second outcomes. The report presents only nonnegative transfers; the four listed arms therefore do not form a complete recovery matrix or an equal-cost comparison. All original negative and tied transfers remain visible.

Table 19: The complete 56-row transfer inventory, retaining unsuccessful transfers, historical cohorts and the separate budget-adaptation experiment.

Source → target	Benchmark / harness	n	Original	Transfer	Δ (pp)
(a) Cross-model — fixed harness and benchmark					
GPT-5.5 → GPT-5.4-mini	AppWorld / Pi	585	23.08	75.21	+52.14
GPT-5.5 → GPT-5.4-nano	AppWorld / Pi	585	17.09	74.02	+56.92
GPT-5.5 → GPT-5.4-mini	DP-Shop / Pi	84	48.81	64.29	+15.48
GPT-5.5 → GPT-5.4-nano	DP-Shop / Pi	84	45.24	71.43	+26.19
GPT-5.5 → GPT-5.4-mini	DP-Travel / Pi	84	11.90	32.14	+20.24
GPT-5.5 → GPT-5.4-nano	DP-Travel / Pi	84	10.71	16.67	+5.95
GPT-5.5 → GPT-5.4-mini	AppWorld / OpenClaw	294	38.10	66.33	+28.23
GPT-5.5 → GPT-5.4-nano	AppWorld / OpenClaw	294	23.81	48.30	+24.49
GPT-5.5 → GPT-5.4-mini	DABStep / Codex	313	76.68	94.25	+17.57
GPT-5.5 → GPT-5.4-mini	LiveSQLBench / Codex	264	51.89	54.17	+2.27
Sonnet 5 → Opus 5	WebShop / Claude Code	280	55.71	65.00	+9.29
GPT-5.5 → Qwen3.8-27B	DP-Shop / Codex	84	58.33	65.48	+7.14
GPT-5.5 → Qwen3.8-max	DP-Shop / Codex	84	57.14	66.67	+9.52
GPT-5.5 → Qwen3.8-27B	DP-Travel / Codex	84	27.38	4.76	-22.62
GPT-5.5 → Qwen3.8-max	DP-Travel / Codex	84	22.62	0.00	-22.62
GPT-5.5 → Qwen3.8-27B	AW Normal / Codex	168	67.86	92.86	+25.00
GPT-5.5 → Qwen3.8-max	AW Normal / Codex	168	54.76	95.24	+40.48
GPT-5.5 → Qwen3.8-27B	TB2.1 / Codex	62	56.45	58.06	+1.61
GPT-5.5 → Qwen3.8-max	TB2.1 / Codex	62	75.81	69.35	-6.45
(b) Cross-harness — fixed GPT-5.5 and benchmark					
Codex → Pi	DABStep	313	81.47	94.89	+13.42
Pi → Codex	DABStep	313	79.87	95.53	+15.65
Pi → OpenClaw	DABStep	313	81.47	95.53	+14.06
Pi → Codex	DP-Shop	84	69.05	79.76	+10.71
Pi → OpenClaw	DP-Shop	84	67.86	80.95	+13.10
Pi → Codex	DP-Travel	84	28.57	77.38	+48.81
Pi → OpenClaw	DP-Travel	84	63.10	82.14	+19.05
(c) Cross-benchmark — fixed GPT-5.5 and harness					
DP-Shop → WebShop	Pi	400	46.25	55.25	+9.00
WebShop → DP-Shop	Pi	84	65.48	59.52	-5.95
TB2.1 → DABStep	Codex	313	74.44	77.64	+3.19
TB2.1 → DP-Travel	Codex	84	71.43	79.76	+8.33
DP-Shop → AW procurement	Codex	120	22.50	18.33	-4.17
AppWorld → DP-Shop	Codex	84	67.86	72.62	+4.76

Success (%); deltas use unrounded counts where available. Historical cohorts retain their displayed populations. WebShop-to-DP-Shop includes target-test-informed repair. Report additions select nonnegative transfers; the original negative and tied results remain visible. Costs are not generally matched.

Table 19 (continued). Simultaneous model and harness changes.

(d1) Historical simultaneous model and harness changes

Donor	Target	Benchmark	n	Original	Transfer	Δ
GPT-5.5 / Codex	GPT-5.4-mini / chat_exec	DABStep	313	53.67	79.55	+25.88
GPT-5.5 / Codex	GPT-5.4-nano / chat_exec	DABStep	313	45.37	61.34	+15.97
Sonnet 5 / Claude Code	GPT-5.5 / Codex	WebShop	280	47.14	61.07	+13.93
Opus 5 / Claude Code	GPT-6 Astra / Codex	DP-Shop	84	77.38	78.57	+1.19

(d2) Simultaneous model and harness changes — GPT-5.5/Codex donor

Target model	Target harness	Benchmark	n	Original	Transfer	Δ
Qwen3.8-27B	Pi	DP-Shop	84	52.38	63.10	+10.71
Qwen3.8-27B	Claude Code	DP-Shop	84	64.29	64.29	0.00
Qwen3.8-max	Pi	DP-Shop	84	63.10	70.24	+7.14
Qwen3.8-max	Claude Code	DP-Shop	84	63.10	70.24	+7.14
Qwen3.8-27B	Pi	DP-Travel	84	16.67	3.57	-13.10
Qwen3.8-27B	Claude Code	DP-Travel	84	67.86	14.29	-53.57
Qwen3.8-max	Pi	DP-Travel	84	73.81	27.38	-46.43
Qwen3.8-max	Claude Code	DP-Travel	84	73.81	1.19	-72.62
Qwen3.8-27B	Pi	AW Normal	168	60.12	92.26	+32.14
Qwen3.8-27B	Claude Code	AW Normal	168	76.79	96.43	+19.64
Qwen3.8-max	Pi	AW Normal	168	48.81	95.83	+47.02
Qwen3.8-max	Claude Code	AW Normal	168	72.02	92.26	+20.24
Qwen3.8-27B	Pi	TB2.1	62	59.68	58.06	-1.61
Qwen3.8-27B	Claude Code	TB2.1	62	67.74	51.61	-16.13
Qwen3.8-max	Pi	TB2.1	62	70.97	79.03	+8.06
Qwen3.8-max	Claude Code	TB2.1	62	72.58	80.65	+8.06

The 24 original Qwen pairs retain all ordinary failures: 14 improve, one ties, and nine decline. Historical simultaneous model/harness changes are separate cohorts.

Table 19 (continued). Travel transfer after budget adaptation.

(e) DP-Travel — GPT-5.5/Codex donor — separate 1,800-second follow-up

Target model	Target harness	n	Original	Transfer 900 s	Transfer 1,800 s	Δ vs. Original
Qwen3.8-max	Codex	84	22.62	0.00	83.33	+60.71
Qwen3.8-27B	Claude Code	84	67.86	14.29	70.24	+2.38
Qwen3.8-max	Pi	84	73.81	27.38	85.71	+11.90
Qwen3.8-max	Claude Code	84	73.81	1.19	77.38	+3.57

This target-test-informed follow-up raises planning/repair limits from 900 to 1,800 seconds and total orchestration from 3,600 to 7,200 seconds. Only four nonnegative arms are reported; these are neither a complete recovery matrix nor equal-cost repeats.

E LEARNED WORKFLOWS AND TASK BEHAVIORS

We first document the task-group evaluation in Section E.1. The remaining analyses archive historical HarnessSkill experiments that optimize instructions and orchestration assets without an independent Kernel compiler. They do not provide task-level explanations for the September Codex/GPT-5.5 comparison in Figure 3. The selected WebShop case uses a separate 400-task Pi evaluation, and the GAIA2 case comes from the first repeat of an earlier three-repeat Pi study.

E.1 WHICH TASKS BENEFIT MOST?

We compare Native and HarnessOpt-BB across five benchmarks using Codex with GPT-5.5 at medium reasoning effort (Figure 3). Each pair is evaluated once on the same held-out tasks within each benchmark. Appendix E.2 records the frozen pairs and their evaluation protocols.

Difficulty and explicit requirements. DABStep's gains are concentrated in the hard split, while easy-task performance declines slightly. WebShop gives a more concrete view of what makes a task demanding: gains increase with the number of required product options and appear across product categories. Each option adds a condition that must be satisfied at checkout.

Ambiguity and adaptation. GAIA2's largest gains occur in ambiguity, followed by adaptability, while execution is unchanged and search declines slightly. This pattern suggests that learned guidance is especially useful for interpreting unclear requests and adjusting decisions as circumstances change.

Constraint type and task size. In Travel, the largest gains occur for tasks involving trains or attractions. Shopping's largest basket group also benefits most, but the gains do not increase steadily with item count. Products and coupons contribute to the same count while imposing different requirements on the final cart.

These single-evaluation comparisons describe task-group differences; they do not identify which learned workflow caused them.

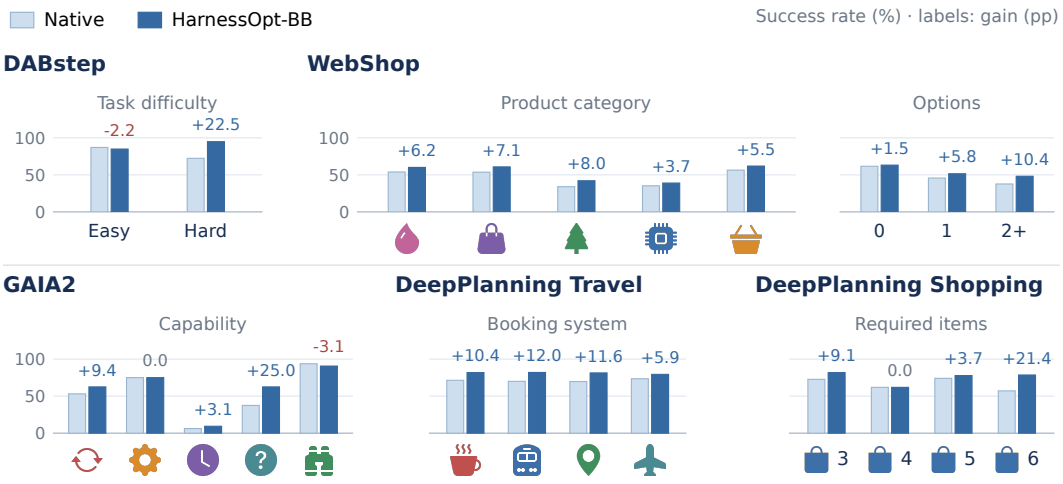

Figure 3: **Held-out success by task group with Codex/GPT-5.5 medium.** Bars compare Native and HarnessOpt-BB success rates (%); labels give signed gains in percentage points. Sample sizes are 313 (DABStep), 280 (WebShop), 160 (GAIA2), and 84 each (Travel and Shopping). Icons denote WebShop categories: beauty, fashion, garden, electronics, grocery; GAIA2 capabilities: adaptability, execution, time, ambiguity, search; and Travel booking systems: restaurants, trains, attractions, flights. DABStep uses official difficulty labels. WebShop options count required product-page selections; its two partitions reuse the same 280 tasks. Shopping's groups count required products and coupons. Travel groups overlap; hotels apply to all Travel tasks and are omitted.

E.2 Task-Group Evaluation Scope

The task-group analysis in Section E.1 uses five frozen Codex/GPT-5.5 medium Native/HarnessOpt-BB pairs. The respective Native and optimized success counts are 233/313 and 292/313 for DABStep, 132/280 and 149/280 for WebShop, 85/160 and 96/160 for GAIA2, 60/84 and 68/84 for Travel, and 57/84 and 63/84 for Shopping. Development-selected steps are 9, 1, 9, 12, and 7, respectively. DABStep uses the September same-batch pair rather than the historical 300/313 selected result. WebShop uses the runtime optimized with GPT-5.5, rather than the Sonnet-harness transfer result (171/280). GAIA2 uses the native Codex CLI R1 retest, with 69 development tasks; it is distinct from JSON-action evaluations. Travel uses submission layout v2 in both arms and the original development-selected step 12.

The 921 task IDs are paired within benchmark. The 23 displayed groups comprise two DABStep difficulty groups, five WebShop product categories, three WebShop option-count groups, five GAIA2 capabilities, four Travel booking-system groups, and four Shopping item-count groups. Each partition is exhaustive and mutually exclusive except for the overlapping Travel constraints; the two WebShop partitions reuse the same evaluation. Shopping counts expected products and coupons from the grader. No selected result row is marked as an infrastructure error. One WebShop Native model-output failure remains a zero score in the fixed denominator. The released frozen records retain paired binary outcomes, group labels, source hashes, and configurations; group success rates, fixes, and regressions can be recomputed without raw task text or gold answers. One evaluation per task and arm describes these selected artifacts and does not estimate variation across optimization runs.

E.3 Learned Workflows

Learned harnesses change both task guidance and execution steps (Figure 4). Shopping adds constraint checks, coupon trials, and a cart audit. Travel and DABStep combine programmatic checks with independent review; AppWorld reviews code and execution results. Findings trigger repairs followed by rechecking or re-execution.

Repairs within a run. Figure 5 shows coupon and train corrections.

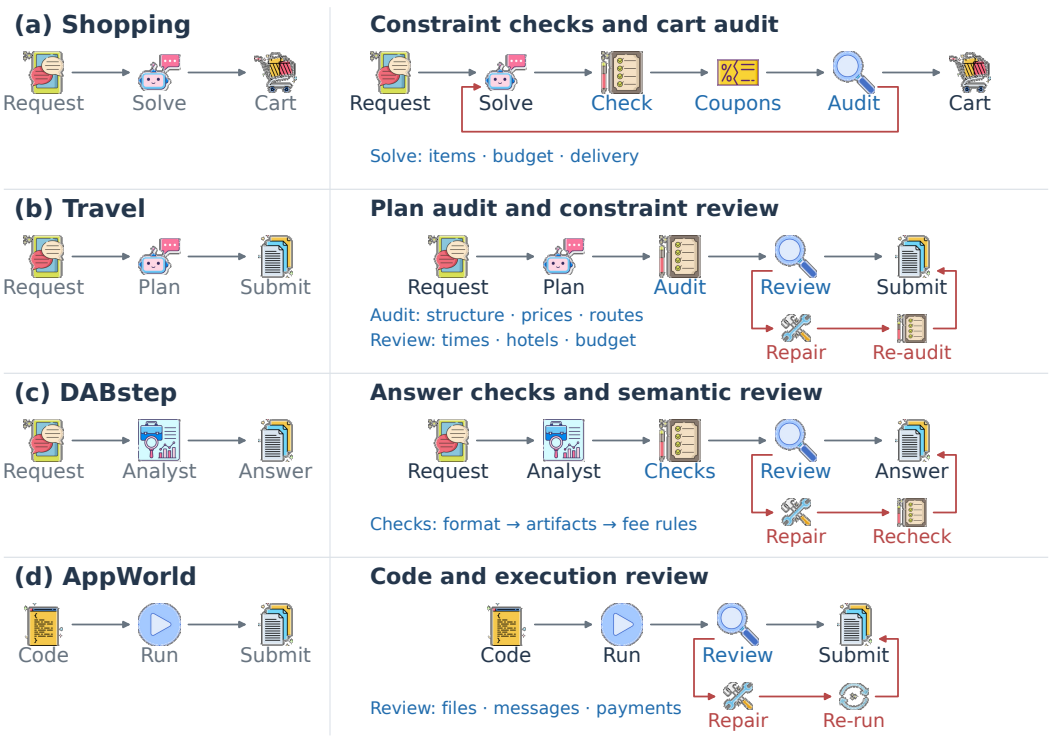

Figure 4: **Initial and learned harness workflows.** Left: initial; right: learned. Blue marks additions and red marks conditional repairs. Travel repairs follow audit or review findings. DABStep's answer stage includes formatting and answer selection.

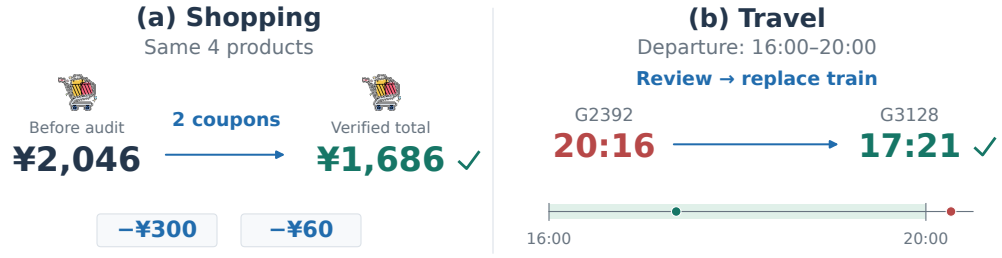

Figure 5: **Repairs in two learned-harness runs.** (a) `shopping_l3_case_8`: two coupons reduce the tool-verified total for the same four products. (b) `travel_en_114`: the reviewer requests a train replacement.

E.4 Paired Task Executions

These eight cases compare runs with the initial and learned harnesses on the same tasks across seven benchmarks. In each pair, the initial run fails the benchmark's task-success criterion and the learned run passes. Task requirements appear above the initial (left) and learned (right) runs. Red marks unmet requirements and green marks satisfied requirements.

Historical GAIA2 evaluations. The earlier three-repeat Pi study reports $56.04\% \rightarrow 60.21\%$; a separate OpenClaw evaluation reports $54.79\% \rightarrow 60.21\%$ after infrastructure failures were reevaluated. These are distinct archived experiments. Figure 3 uses the September Codex evaluation, $53.13\% \rightarrow 60.00\%$, with one evaluation per task and arm.

E.4.1 Recipients and Payment Funding

Email recipients. The task requests a zip archive of Bali photographs in one email to three roommates. The initial run merges roommate contacts with unfiltered search results, sending the email to 26 recipients. The learned run filters contacts by the roommate relationship and sends the archive to the three intended recipients. It then checks the recipients and attachment in the sent message.

Payment funding. Both runs compute the dinner reimbursement as $30.50 from a $122 bill shared by four people. The initial run attempts to pay with a card that has insufficient funds. The learned run checks the Venmo balance before choosing a funding source, completes the payment, and checks the recorded transaction.

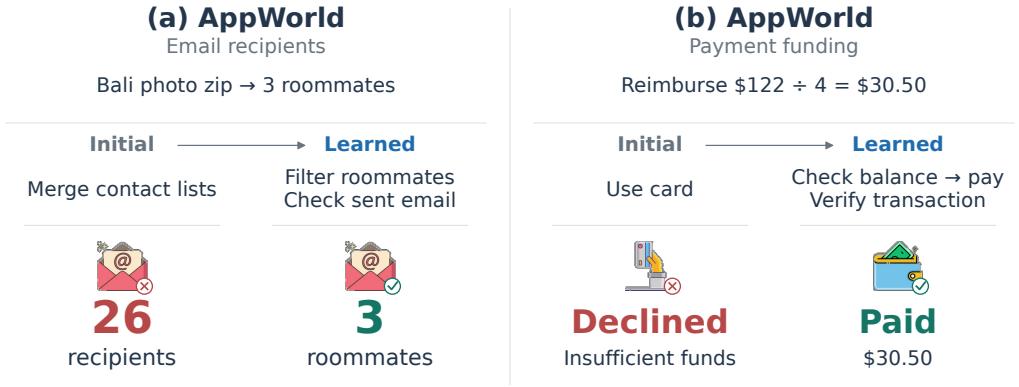

Figure 6: **AppWorld: recipients and payment funding.** (a) `baeb104_1`: emailing vacation photos to roommates. (b) `2d9f728_2`: reimbursing a roommate for dinner.

E.4.2 Following Up and Keeping Services Available

Following up on replies. The task asks the agent to invite three friends to Game Night on Friday, October 18 and, if necessary, move it to the first date all three can attend. Conflicting events must be canceled and the new date confirmed. The initial run creates the event and sends invitations, then stops. The learned run reads the replies: two friends cannot attend on Friday, and all three are free on Saturday. It checks the calendar, cancels the original event and a conflicting event, and sends all three friends confirmations for Saturday, October 19, 19:00–23:00.

SSH availability. The task requires an Alpine virtual machine reachable over SSH on port 2222. The initial run starts QEMU in a console session and logs in successfully during setup, but the later benchmark check returns `Connection refused`. The learned run uses `-daemonize` to keep QEMU running independently of the setup command. A separate reviewer agent opens fresh SSH sessions and tests an interactive shell. The learned run passes the benchmark check.

E.4.3 Applying Filters and Rules at the Right Scope

SQL filter scope. The SQL task asks for mean and median signal quality by weather condition, together with the number of analyzable signals, defined by $\mathrm{SNQI} > 0$. The initial query puts this

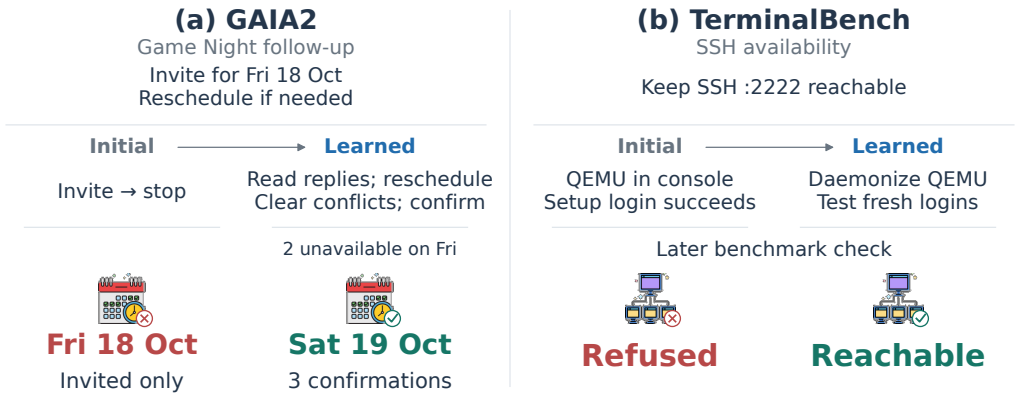

Figure 7: **Follow-up and SSH availability.** (a) GAIA2, Game Night: rescheduling after replies. (b) TerminalBench, `qemu-alpine-ssh`: SSH access at the benchmark check.

condition in a `WHERE` clause, restricting all three aggregates. The learned query retains all signals for the mean and median and uses `COUNT(*) FILTER (WHERE SNQI > 0)` for the count. Both queries execute, but only the latter passes.

Fee-rule matching. DABstep asks for the average applicable fee for a EUR 1,234 restaurant payment under account type H and scheme SwiftCharge. The initial answer averages only three strict matches, excluding records with null or empty restrictions, and returns EUR 5.681933. The learned harness's fee procedure treats a null or empty restriction as imposing no constraint on that field. The learned run therefore averages all 60 applicable records and returns EUR 7.180793.



Figure 8: **Calculation scope.** (a) LiveSQLBench-Lite, `alien_1`: SNQI > 0 applies only to the count. (b) DABstep, task `1331`: null or empty rule fields impose no restriction. The reported fees are the submitted answers; 3 and 60 count the records used to compute them.

E.4.4 Selecting Options and Ranking Candidates

Product options. The WebShop request specifies a high-speed, red, three-foot USB charging cable under $50. The initial run buys a mixed-length cable pack for $9.99 without selecting any options and receives partial reward 0.5. The learned run chooses a different product for $11.99, selects `3 ft` and `red`, and confirms both options on the page before checkout. This purchase satisfies the request and receives reward 1.0.

Flight duration. The Travel task requests the shortest direct flight from Nanning to Chongqing on November 12, 2025. The initial run chooses CZ6797, which takes 105 minutes. Following the learned harness's transport guide, the learned run keeps only direct flights on the requested route and

ranks them by duration. It selects CA3527 at 100 minutes, the shortest direct option in the query results.

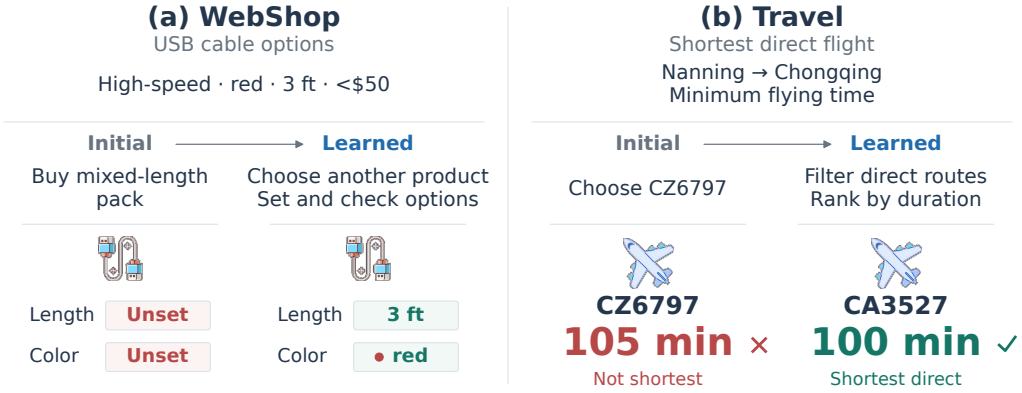

Figure 9: **Product options and flight selection.** (a) WebShop, `webshop_00024`. (b) DeepPlanning Travel, `travel_en_34`.

F Runtime Structure and Execution Details

These measurements describe selected host programs and do not establish lower total computation or an equal-cost causal effect.

F.1 Kernel Editing and Host-Code Measurements

The routes differ in what the optimizer edits. Kernel editing revises a natural-language specification of policies and workflow; a compiler turns it into a runtime and can revise the Kernel plan using development evidence. Direct runtime optimization edits the executable harness and supporting assets directly. We compare Pi-agent runs from the same seed runtime; both routes execute TypeScript.

Kernel editing adds less host code. Across all five benchmark pairs in Table 6, Kernel editing adds fewer functions, logical lines and decision points than direct runtime optimization. The Kernel route also achieves 2.9–5.6 percentage points higher test success. Appendix F gives the counting convention, evaluation scope, and unchanged WebShop pair.

Table 6 definitions. F_{add} counts added named functions; ΔL and ΔD measure net growth in logical lines and decision points. Structural entries are kernel / direct runtime, measured on the selected runtimes evaluated on the n_{test} paired tasks. Δ success is kernel minus direct-runtime test success, in percentage points (pp). Travel and Shopping are the DeepPlanning domains. Each pair uses one development-selected runtime per route.

Behavior can reside in policies as well as host code. The learned policies specify concrete responsibilities and action constraints. In DABStep, the Kernel-compiled runtime instructs the agent to select an answer from its computed table; the directly edited runtime also adds host-code rules that can override the answer. In Shopping, the policy in the Kernel-compiled runtime requires a state check after each cart mutation, whereas the directly edited runtime includes host code that executes an approved product ledger. These examples show how policies in Kernel-compiled runtimes govern local decisions alongside executable workflow control. The corresponding excerpts appear in Appendices F.5 and F.6.

We measure host-code growth for the five Pi-agent benchmark pairs in Table 6. Each pair uses **one development-selected runtime per route**. All 10 seed runtimes are byte-identical: one named function, four logical lines and no decision points. Growth is measured from that shared seed. Table 20 lists the selected steps and paired test populations, including the unchanged WebShop pair.

Table 20: **Selected checkpoints in the structural study.** Step 0 is the seed runtime; subsequent steps are optimization iterations. Test IDs match within each benchmark pair.

Benchmark	Test tasks	Kernel step	Runtime step	Selected artifacts
Travel	84	6	8	Modified
DABStep	313	8	6	Modified
GAIA2	160	5	5	Modified
AppWorld	585	6	6	Modified
Shopping	84	8	7	Modified
WebShop	400	0	0	Identical seed

Counting convention. Counts cover host TypeScript in each selected runtime, excluding policy prose, task-time scripts and source copies stored as compilation evidence. Added functions are new named callables relative to the seed. Logical lines are nonempty physical lines after masking comments and string/template text; net growth is the final count minus the seed count. Decision points are a static syntax-tree count of branches, loops, conditional expressions, catches, and logical operators. Neither route adds any classes. Travel and DABStep Kernels modify the existing `run` and its policy without introducing new named functions.

Evaluation setup. Each selected runtime is evaluated once on the same held-out tasks within each benchmark. Both routes allow eight optimization steps; AppWorld stops after six on perfect development performance. Success rates use the full test set. The total sums structural counts only.

Scope. A common step limit does not match development interactions or compiler expenditure. Failures remain in the denominator, including recorded infrastructure errors: 0/5 for Travel and 0/2 for DABStep (Kernel/direct runtime). WebShop retains the identical seed in both routes, with test scores 216/400 and 221/400. It therefore illustrates execution variation without an artifact change and is excluded from the modified-artifact growth totals. Host-code counts exclude policy prose and task-time scripts, so they do not measure total computation or establish that smaller programs cause higher success.

F.2 How Do Local Repairs Interact with Global Review?

A local change can trigger work throughout the review-and-repair loop. On Travel task 116, the directly optimized runtime checks the repaired plan with four reviewers, then repeats all four after only a ledger pointer changes (Figure 10). The Kernel-compiled runtime adds a missing seven-minute walk, re-runs its reviewer, and returns after four agent calls; the directly optimized runtime makes fifteen calls.

	Kernel-compiled runtime	**Directly optimized runtime**
Plan and review	Planner + reviewer. Finds a missing walking segment.	Planner + four reviewers. Finds defects in the plan.
Repair and re-review	Adds the walk; reviewer passes. Returns the plan.	Planner + four reviewers. Finds an incorrect ledger reference.
Final repair	Already finished.	Fixes only the reference; **all four reviewers run again**.
Total	**4 agent calls; task succeeds**	**15 agent calls; task fails**

Figure 10: **A local repair under two global review policies.** On task 116, each runtime repairs a different defect found in review. The final direct-runtime edit changes a ledger reference from a walking activity to a meal, without changing the itinerary, and all four reviewers run again. Totals include every observed agent call. Final hard success is 1/0 (Kernel/direct runtime).

This case makes the connection between the Kernel's two levels explicit: local rules specify what to check or repair, while the global workflow determines which checks run next and when execution ends. Representing both levels provides an optimization target for revising the complete mechanism.

The selected traces repair different defects and come from an earlier cohort than Table 6; they illustrate this design dependency rather than establish an average efficiency gain or a causal effect of Kernel editing. Appendix F gives the complete observed agent-call sequences and additional cases, including a failed review.

F.3 Review and Repair Control Flow

Figure 11 expands the paths summarized in Figure 10. They come from an earlier Travel cohort, separate from the structural measurements, with selected steps 6/8. The final direct-runtime repair changes a constraint-ledger reference from a walking activity to a meal. No itinerary entity, time, route, or cost changes, but all four reviewers run again. The figure includes every observed agent call; deterministic checks and unexecuted branches are omitted.

Kernel-compiled runtime 4 agent calls	**Directly optimized runtime** 15 agent calls
Stage 1: plan, then review	

`await runtime.agent({`
`  name: "main", ... });`
`await runtime.agent({`
`  name: "travel_verifier", ... });`
Review: a 7-minute walk to a restaurant is missing.
	`await runtime.agent({`
	`  name: "travel_planner", ... });`
	`await runtime.agent({`
	`  name: "travel_verifier", ... });`
	`await runAttractionScheduleAudit(...);`
	`await runRestaurantConstraintCheck(...);`
	`await runMealProvenanceAudit(...);`
	Reviews: defects to repair.

Stage 2: repair, then review again	

`await runtime.agent({`
`  name: "main", ... });`
`await runtime.agent({`
`  name: "travel_verifier", ... });`
Planner adds the walk.
Review passes; plan returned.
	`await runtime.agent({`
	`  name: "travel_planner", ... });`
	`await runAttractionScheduleAudit(...);`
	`await runRestaurantConstraintCheck(...);`
	`await runMealProvenanceAudit(...);`
	`await runtime.agent({`
	`  name: "post_repair_verifier", ... });`
	Review: one bookkeeping entry points to the wrong activity.

Stage 3: fix one field, then review again	

(finished after Stage 2)	Planner edits one field of the constraint ledger, the list that records which itinerary activity satisfies each task constraint:
	`constraint_ledger[4].artifact_path`
	`- "daily_plans[3].activities[8]" // walk`
	`+ "daily_plans[3].activities[9]" // meal`
	Itinerary unchanged, yet **all four reviewers run again**:
	`await runtime.agent({`
	`  name: "travel_planner", ... });`
	`await runAttractionScheduleAudit(...);`
	`await runRestaurantConstraintCheck(...);`
	`await runMealProvenanceAudit(...);`
	`await runtime.agent({`
	`  name: "post_repair_verifier", ... });`

Figure 11: **Travel: how each runtime re-reviews after a repair.** On task 116, each runtime repairs a different defect found in review. The left runtime re-runs its single reviewer and stops. The right runtime re-runs all four reviewers after every repair, including a final one that changes only a ledger pointer and leaves the itinerary unchanged. The calls highlighted in orange on the right add reviews of attraction schedules, restaurant constraints, and meal provenance; each helper invokes one agent. Final task success is 1/0 (Kernel/direct runtime).

F.4 Task-Level Comparisons

Figure 12 shows three cases in which the Kernel-compiled runtime succeeds and the directly optimized runtime fails. These selected cases use earlier snapshots than the structural measurements in Table 6. Initial Travel plans were not separately graded.

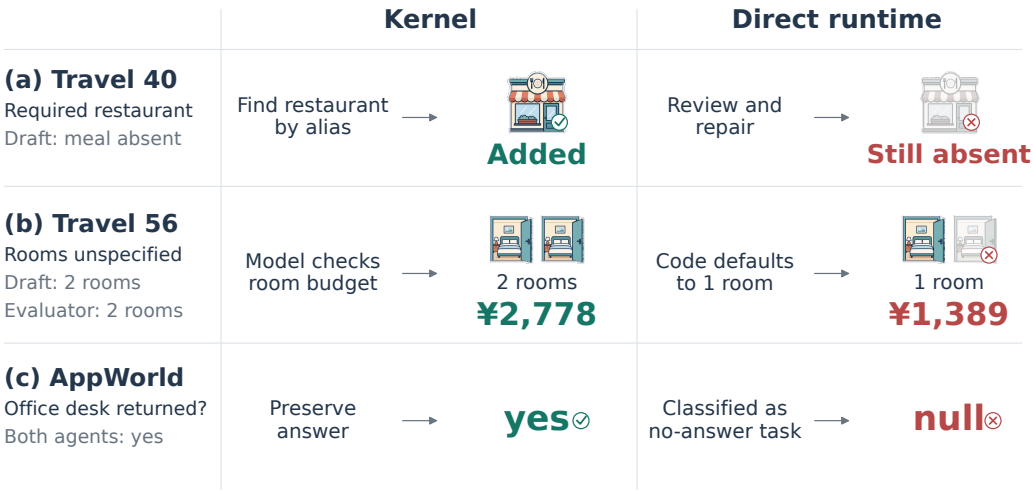

Figure 12: **Recovering a missing requirement and preserving correct results.** Each row shows the shared task facts, each route's action, and the final result. Green and red indicate passing and failing final runs. Travel 56's request does not specify room count; two rooms match the evaluator.

Recovering an omission. Resolving a restaurant's returned name lets the Kernel route add a required meal. The direct-runtime route also attempts repair but leaves this requirement unsatisfied.

Preserving an existing result. Travel and AppWorld illustrate how host-code assumptions can change a result that already matches the evaluator. In Travel, the direct runtime's one-room default enters the review and causes the agent to halve the lodging budget. In AppWorld, its question-intent rules misclassify an explicit yes-or-no request as requiring no answer; completion and output handling then replace `yes` with `null`. This correct-answer-to-`null` pattern occurs on 16 of 585 paired AppWorld tasks.

A failed review. A Travel counterexample shows that an explicit Kernel rule can still be executed incorrectly. The policy requires excluding connecting flights before ranking departure times. The audit displays only the first segment, omitting the segment count and airports. On that basis, the agent replaces a direct flight with a connecting itinerary. A second audit accepts the replacement, while the direct-runtime route retains the direct flight. The failure concerns how the check is executed, despite the relevant rule being present.

F.5 Code Example: DABStep

Figure 13 compares answer selection in the DABStep runtimes selected in Table 6. The kernel-compiled runtime leaves answer selection to the instructed agent; the directly optimized runtime adds host-code rules that can override its answer using the printed fraud-rate table.

Kernel-compiled runtime	Directly optimized runtime
The main agent selects from its computed candidate table. Its policy requires: *"The final answer must be exactly the top value or documented tie set from that table."* `const result = await runtime.agent({ name: "main", prompt: runtime.task, systemPrompt: DABSTEP_SYSTEM_PROMPT, }); return result.text;`	After the main agent returns, host code can replace its answer using the printed fraud-rate table. `const fraud = fraudRateDiagnostic( task, transcript); if (fraud && !sameInner(inner, fraud) && shouldOverrideFraudRate( task, inner)) { // logging omitted return '<answer>${fraud}</answer>'; }`

Figure 13: **DABStep: who selects the final answer?** Left: kernel editing yields a runtime that passes the instructed agent's answer through. Right: direct runtime optimization adds a rule that can override it. Excerpts are from Table 6's artifacts; identifiers and expressions are unchanged, with whitespace reformatted and logging omitted where marked.

F.6 Additional Code Example: Shopping

Figure 14 compares cart execution in the Shopping runtimes evaluated in Table 6. The kernel-compiled runtime instructs the agent to perform cart mutations, while the directly optimized runtime includes host code that executes an approved product ledger.

Kernel-compiled runtime	Directly optimized runtime
Shopping: adding products to the cart	
The main agent issues cart calls. Its workflow policy requires: *"Issue cart mutations one public tool call at a time; after each product or coupon add/delete, inspect the returned state or call `get_cart_info` before the next mutation."* The runtime includes that policy in `prompt`: `const result = await runtime.agent({ name: "main", prompt}); return result.text;`	After ledger approval, host code adds missing units from the selected products: `for (const product of intent.products) { const have = observed.get( product.product_id) || 0; for (let i = 0; i < Math.max(0, product.quantity - have); i += 1) { await dpCall(runtime, "add_product_to_cart", { product_id: product.product_id, quantity: 1 }, 20); } }`

Figure 14: **Shopping: who executes the cart operations?** Left: the policy and agent call. Right: host code adds missing product units from the approved ledger. Identifiers and expressions are unchanged; whitespace is reformatted.

The direct runtime falls back to the agent when ledger execution produces no final state. The kernel-compiled runtime leaves cart operations to the instructed main agent, which can still execute task-time scripts.

G Initialization and Workflow Evolution

We examine how initial Kernel evidence affects optimization and how learned workflows use review and repair. The development trajectories and selected examples below characterize observed behavior; they do not establish convergence to a common optimum or isolate individual design choices.

G.1 How Does Kernel Initialization Affect Optimization?

We construct five initial Kernels for AppWorld and DABStep (Table 21). For s_0–s_3, one model solves training tasks and a second summarizes the resulting trajectories and supporting evidence into a Kernel. The two roles use all four combinations of GPT-5.4 and GPT-5.5. The fifth startpoint, s_4, has no task-specific evidence. All five retain an existing agent workflow.

Table 21: **Initial Kernel construction.** These model roles construct the startpoint; they do not identify the student used for final evaluation. The s_4 startpoint contains no task-specific evidence.

Startpoint	Task-solving evidence	Evidence summarizer
s_0	GPT-5.4	GPT-5.4
s_1	GPT-5.4	GPT-5.5
s_2	GPT-5.5	GPT-5.4
s_3	GPT-5.5	GPT-5.5
s_4	–	–

Different initializations produce different development trajectories. Figure 15 shows that accuracy fluctuates during optimization and then stabilizes at different levels on both benchmarks. The resulting differences motivate examining the learned workflows and their task-solving behavior.

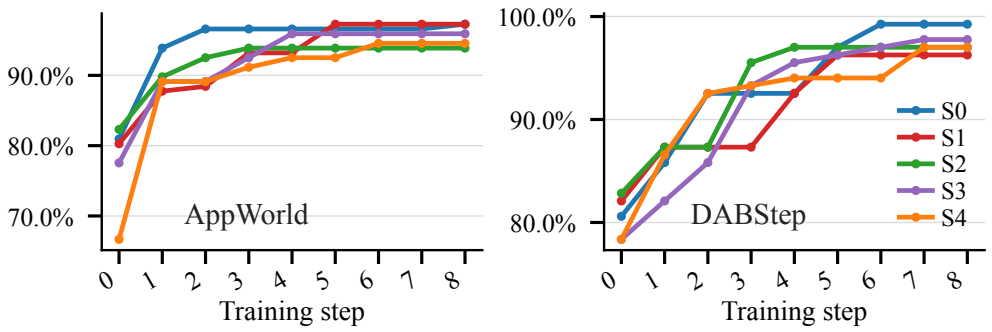

Figure 15: **Development accuracy from five initial Kernels.** AppWorld and DABStep accuracy across optimization steps. The model pairs used to construct each startpoint are listed in Table 21.

G.2 What Distinguishes Harnesses with Similar Workflows?

The AppWorld workflow separates generation, review, audit, and repair. Figure 16 shows a solver that produces a solution, a reviewer that inspects it, and an auditor that checks visible execution logs. Detected issues trigger correction and another pass through review and execution; otherwise, the solution proceeds to final evaluation. Skill inputs provide instructions to each of the four agent roles.

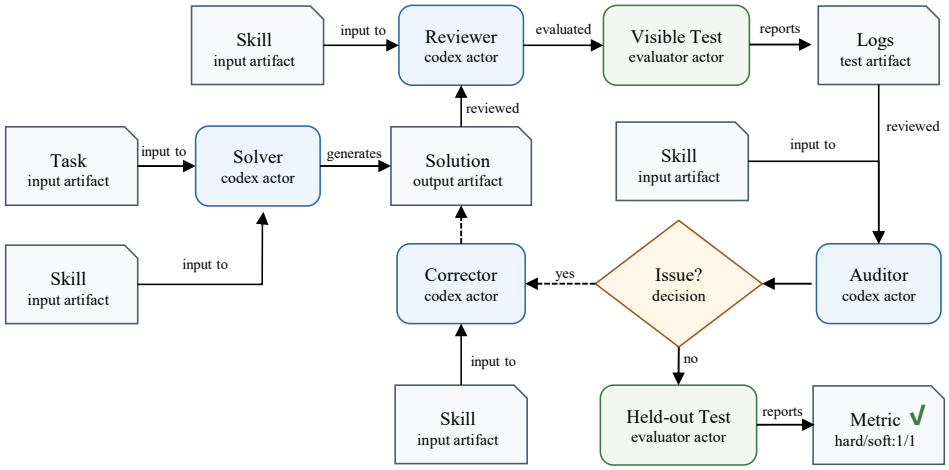

Figure 16: **A learned AppWorld workflow.** Generation, review, execution auditing, and correction are connected through shared solution and log artifacts. Each agent role receives a Skill input. The diagram shows the successful evaluation branch; it does not summarize performance across initializations.

Similar workflows can use different source artifacts. An earlier paired AppWorld study illustrates a complementary distinction: two learned harnesses with similar workflows use different information sources for the same reminder-email task. One searches the email subject and body for dates and misses the schedule attachment. The other follows a Kernel instruction to preserve explicit source entities before applying heuristics: it retrieves the schedule and recipient attachments, parses their contents, and checks the reminder drafts (Figure 17).

Task: Schedule reminder emails to company members at 09:00 on each day in the manager's talk schedule.

Message-text heuristics (s_4)	**Explicit source grounding** (s_0)
Search the email subject and body for dates; miss the schedule attachment.	Read the schedule and recipient attachments; check reminder drafts against them.
The task fails.	The task passes.

Kernel instruction: "Preserve explicit source entities before applying heuristics: names, amounts, titles, dates, paths, and requested output fields."

Figure 17: **Source grounding within a learned workflow.** The two harnesses differ in which source artifacts they retrieve and verify for the same reminder task.

G.3 Can Optimization Introduce a New Repair Workflow?

Evolution adds a feedback path for correcting generated solutions. An initial AppWorld harness produces a solution that assumes ISO-formatted dates. It fails when the calendar instead contains "Saturday, May 20, 2023." The evolved harness executes the solution, reviews the visible feedback, and revises the parser to handle both formats (Figure 18). The final task assertions then pass. This example connects a change in execution structure—adding a review-and-repair path—to a concrete recovery from a parsing error.

Task state: A calendar date is given as "Saturday, May 20, 2023."

Initial: direct solution	**Evolved: execution-driven repair**
Generate a solution with an ISO-only parser.	Generate and execute a solution, then audit visible feedback.
The parser expects `%Y-%m-%d`; execution raises a date-format error.	Repair the parser to accept ISO and natural-language dates, then execute again.
Final task assertions do not all pass.	Final task assertions pass.

Repaired parsing behavior (illustrative pseudocode):

```
try:
    date = parse(date_text, "%Y-%m-%d")
except DateFormatError:
    date = parse(date_text, "%A, %B %d, %Y")
```

Figure 18: **Initial and evolved harnesses on a date-parsing task.** The evolved workflow uses visible execution feedback for correction; held-out assertions measure the final outcome.

H LIMITATIONS

Attribution and initialization. The experiments span different inherited runtimes, editable boundaries, initial designs, and model roles. System-level improvements therefore do not uniformly isolate the Kernel representation. A minimal handwritten Kernel differs from empty initialization, and fixed-core program editing differs from full-source optimization. The reported paired artifact studies and feedback ablations address these questions under their stated conditions. Additional controls jointly matching initialization, editing permissions, and total optimization expenditure remain future work for separating the representation and compiler contributions.

Data reuse and statistical scope. Repeated search on a small development pool can overfit it. The shared-development protocol reuses tasks for reflection and selection; the source-selection qualifications and unlinked report aggregates in Appendix D add uncertainty to the affected imported cohorts. Instance separation does not imply separation of templates, repositories, or environments, and local split checks cannot exclude foundation-model pretraining contamination. Several results use one search seed or limited repeats; they establish observations under those conditions, not universal or statistically significant gains for every component.

Realization and reproducibility. Compiler and reviewer checks are finite, backend-dependent, and model-based. They improve inspectability but do not prove semantic equivalence between Kernel and runtime. Hosted-model aliases and execution environments can change, and some historical reports lack immutable model identities or complete runtime metadata. Exact replication is correspondingly limited for those records.

Cost and generalization. Additional reasoning, validation, and repair can improve success while increasing resource use; complete cost/latency comparisons are unavailable for all cohorts. Negative package transfer shows dependence on target models, interfaces, and budgets. Kernel recompilation across hosts and complete multi-benchmark joint evaluation remain open empirical questions. Evaluation on additional task families and execution environments would extend the scope of the current observations.