Agent-Native Research Artifacts

Jiachen Liu*
University of Michigan
Ann Arbor, MI, USA
amberljc@umich.edu

Jiaxin Pei
Chenglei Si
Alex Pentland
Stanford University
Stanford, CA, USA

Zexue He
Zhenyu Zhang
Stanford University
Stanford, CA, USA

Ao Qu
MIT
Cambridge, MA
USA

Haizhong Zheng
Beidi Chen
CMU
Pittsburgh, PA, USA

Xiangru Tang
Yale University
New Haven, CT, USA

Ruihao Zhu
Cornell University
Ithaca, NY, USA

Xiaoyan Bai
University of Chicago
Chicago, IL, USA

Mingyuan Wu
Fan Lai
UIUC
Urbana, IL, USA

Zijian Jin
New York University
New York, NY, USA

Zhiyang Chen
University of Toronto
Toronto, ON, Canada

Jintao Huang
Ohio State University
Columbus, OH, USA

Yujuan Fu
Meta
Menlo Park, CA, USA

Haojie Ye
NVIDIA
Santa Clara, CA, USA

Jiachen Sun
LinkedIn
Sunnyvale, CA, USA

Yuan Yuan
Boston College
Chestnut Hill, MA
USA

Baoyu Zhou
Arizona State University
Tempe, AZ, USA

Yao Li
Portland State University
Portland, OR, USA

Chenyu You
SBU
Stony Brook, NY
USA

Junyuan Hong
National University of Singapore
Singapore, Singapore

Shangquan Sun
Nanyang Technological University
Singapore, Singapore

Shijian Lu
Nanyang Technological University
Singapore, Singapore

Yiming Qiu
University of Hong Kong
Hong Kong, Hong Kong

Dianzhuo Wang
Qian-ze Zhu
Harvard University
Cambridge, MA
USA

Lichang Chen
Meta Superintelligence Labs
Menlo Park, CA, USA

Runyu Lu
Ang Chen
University of Michigan
Ann Arbor, MI, USA

Mosharaf Chowdhury
University of Michigan
Ann Arbor, MI, USA

Zechen Zhang
Harvard University
Cambridge, MA
USA

Abstract

Scientific publication compresses a branching, iterative research process into a linear narrative, discarding the majority of what was discovered along the way. This compilation imposes two structural costs: a **Storytelling Tax**, where failed experiments, rejected hypotheses, and the branching exploration process are discarded to fit a linear narrative; and an **Engineering Tax**, where the gap between reviewer-sufficient prose and agent-sufficient specification leaves critical implementation details unwritten. Tolerable for human readers, these costs become critical when AI agents must understand, reproduce, and extend published work. We introduce the **Agent-Native Research Artifact (Ara)**, a protocol that replaces the narrative paper with an agent-executable research package structured around four layers: scientific logic, executable code with full specifications, an exploration graph that preserves the failures compilation discards, and evidence grounding every claim in raw outputs. We complement the protocol with the **Ara ecosystem**, a coordinated set of agent skills—the *Live Research Manager* (LRM) that captures decisions and dead ends during ordinary development, the *Ara Compiler* that translates legacy PDFs and repos into Aras, and the *Ara Seal*, a three-level review pipeline (analogous to a grammar checker for prose)—so artifacts are produced, imported, and verified automatically while human reviewers focus on significance, novelty, and taste. On PaperBench and RE-Bench, Ara raises question-answering accuracy from 72.4% to 93.7% and reproduction success from 57.4% to 64.4%. On RE-Bench's five open-ended extension tasks, preserved failure traces in Ara accelerate progress, but can also constrain a capable agent from stepping outside the prior-run box depending on the agent's capabilities.

Keywords: agent-native research artifacts, agent skills, AI agents, reproducibility, knowledge representation

*Corresponding author.

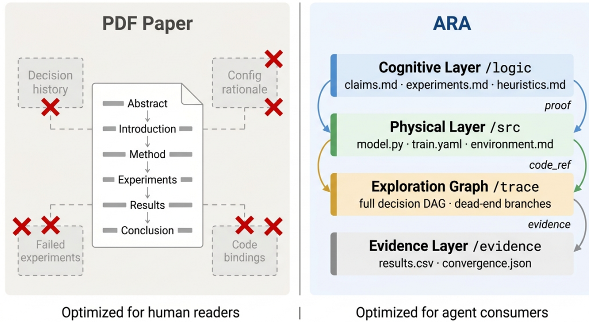


Figure 1. PDFs compile a rich research object into a lossy narrative; Ara preserves it as an agent-executable package.

1 Introduction

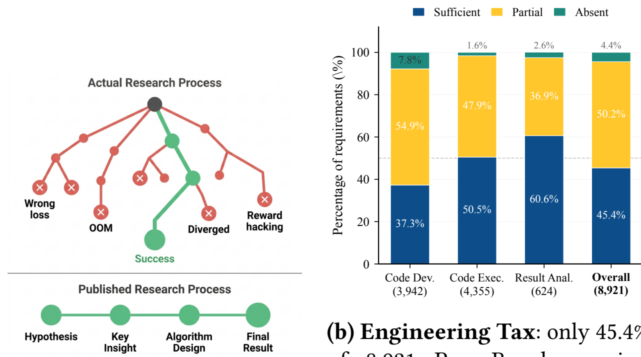
Research produces a rich, branching knowledge object: months of hypotheses tested and rejected, implementation tricks discovered through trial and error, design alternatives weighed against each other, and the full exploration trajectory that explains why the final approach was chosen. Publishing compiles this object into a linear narrative [12, 40], discarding failed experiments, tacit engineering knowledge, and the branching process to satisfy the conventions of human-readable storytelling [17, 49]. This compilation cost, a consequence of the documentation convention rather than any particular file format, was tolerable when every consumer of a paper was human. It is not when AI agents routinely read papers to understand a field, reproduce experiments to validate findings, and extend published methods to new settings [32, 36]: each task requires precisely the knowledge that compilation discards (Figure 1). More specifically, the compilation incurs two structural costs.

The first is the **Storytelling Tax**: the systematic erasure of research process knowledge imposed by compilation into narrative (Figure 2a). Research does not proceed linearly—it branches, backtracks, and accumulates hard-won failure knowledge before converging on a publishable result [28, 40]. Narrative compilation flattens this process into a polished linear story, discarding every failed experiment, rejected hypothesis, and abandoned approach. This emphasis on success leaves failures undocumented; although modern platforms archive final artifacts, the branching research process remains unrecorded, causing independent rediscovery of the same dead ends across groups [17, 49]. Our own analysis of the METR eval-analysis-public dataset [61], covering 24,008 agent runs across 21 frontier models on RE-Bench, quantifies the cost (per-task breakdown in Appendix F.2): failed runs account for **90.2%** of total dollar cost (and 59.2% of tokens), with a median failed-to-success token ratio of **113**×, agents without access to prior failure records must independently rediscover every dead end. Equally lost is the record of human *judgment* along the trajectory: every rejection, revision, and endorsement is a preference signal over what constitutes good research, the scarce resource that binds once agents shoulder the grunt work. Narrative compilation discards this signal; a preserved trajectory renders it as structured supervision that compounds across projects.

The second is the **Engineering Tax**: the gap between *reviewer-sufficient* and *agent-sufficient* documentation (Figure 2b). The paper communicates its contribution at the level of detail needed to convince a human reviewer; the codebase provides an implementation but not the operational specification needed to execute it. Between the two lies tacit knowledge [44]—algorithmic tricks, implementation decisions, and configuration choices—knowledge that exists in no written document and is transmitted only through direct lab contact or painstaking reverse-engineering. We quantify this void by classifying each of PaperBench's 8,921 expert-annotated reproduction requirements across 23 ICML 2024 papers [54] against its source PDF (per-category breakdown and gap-type taxonomy in Appendix A.2): despite widespread artifact sharing, only **45.4%** are fully specified. Code development is the most underspecified category (37.3% sufficient), and missing hyperparameters alone account for 26.2% of all gaps (full breakdown in Appendix A.2): a fundamental mismatch between the precision at which papers are written (sufficient to produce belief) and the precision at which agents must operate (sufficient to produce correct execution) [5, 56].

Three trends suggest the shape of an artifact that pays neither tax. **First**, AI agents now co-author code, run experiments, and iterate on hypotheses alongside researchers [36], with LLM adoption associated with paper-production increases of 23.7%–89.3% across scientific fields [29]; the full research trajectory is already captured as machine-readable text in researcher-agent coding sessions, yet no protocol preserves it as a first-class output. **Second**, humans skim [48] while agents benefit from exhaustive detail, so a single artifact optimized for human narrative can no longer serve both audiences. **Third**, research is scaling into a massively parallel enterprise in which agents fork, extend, and merge each other's work at machine speed; narrative PDFs cannot be forked, diffed, or merged, but a structured, lossless artifact can, enabling research to compound like software.

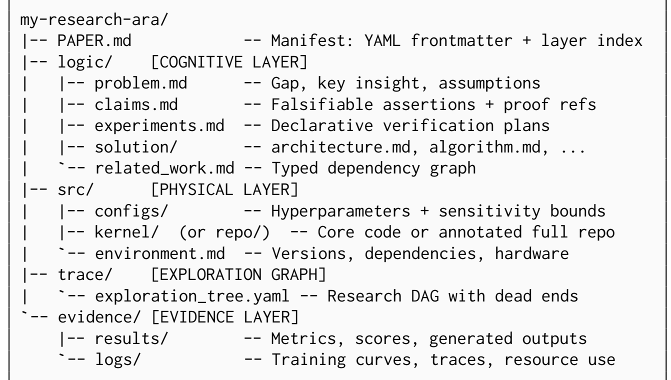
Existing efforts address fragments of this problem. The FAIR principles [62] mandate findable, accessible data but say nothing about the structure of research *arguments*. RO-Crate [53] packages research artifacts as archival bundles, not executable objects. Nanopublications [20] formalize atomic claims but lack the execution layer needed for reproduction. The emerging `AGENTS.md` standard [41] provides agent-oriented documentation for code repositories but does not address the epistemic structure of research itself. None of these jointly structure scientific logic, executable code, and exploration history into an operable object (§K; a thread-by-thread comparison is in App. J).



(a) Storytelling Tax: branching exploration with dead ends collapses into a linear story.

(b) Engineering Tax: only 45.4% of 8,921 PaperBench requirements are PDF-sufficient. Gap-type distribution in App. A.2 (Fig. 7).

Figure 2. The two structural costs of PDF-based research.



```
my-research-ara/
|-- PAPER.md             -- Manifest: YAML frontmatter + layer index
|-- logic/    [COGNITIVE LAYER]
|   |-- problem.md       -- Gap, key insight, assumptions
|   |-- claims.md        -- Falsifiable assertions + proof refs
|   |-- experiments.md   -- Declarative verification plans
|   |-- solution/        -- architecture.md, algorithm.md, ...
|   `-- related_work.md -- Typed dependency graph
|-- src/      [PHYSICAL LAYER]
|   |-- configs/         -- Hyperparameters + sensitivity bounds
|   |-- kernel/  (or repo/)  -- Core code or annotated full repo
|   `-- environment.md   -- Versions, dependencies, hardware
|-- trace/    [EXPLORATION GRAPH]
|   `-- exploration_tree.yaml -- Research DAG with dead ends
`-- evidence/ [EVIDENCE LAYER]
    |-- results/         -- Metrics, scores, generated outputs
    `-- logs/            -- Training curves, traces, resource use
```

Figure 3. The Ara directory structure. Full schema in Appendix A.4; design rationale in Appendix A.

We propose the **Agent-Native Research Artifact (Ara)**, a protocol that recasts the primary research object from narrative document to agent-executable knowledge package, with papers serving as compiled views of the underlying artifact (Figure 1). Ara organizes research into four interlocking layers: *structured scientific logic* that distills the paper's conceptual abstractions into queryable claims and dependency graphs; *executable code* with full operational specifications; an *exploration graph* preserving the branching research process—failed experiments, rejected hypotheses, and design pivots—that narrative compilation discards; and *grounded evidence* binding every claim to its raw empirical outputs. Instead of parsing prose, reverse-engineering repositories, and rediscovering dead ends, an agent operating on an Ara artifact can query structured claims, execute declarative specifications, and build on the full decision history directly—a research object designed not to be read, but to be *operated.*

A protocol is only useful with the surrounding tooling that produces and validates conforming artifacts. We complement Ara with an **ecosystem** (§3) of three coordinated agent skills: a *Live Research Manager* (LRM) that captures decisions and dead ends silently during AI–human research, an *Ara Compiler* that translates legacy PDFs and repositories into Ara format for backward compatibility with the existing literature, and an *Ara Seal* review pipeline whose three escalating levels automate structural verification, argumentative-rigor auditing, and budget-aware reproduction (analogous to a grammar checker for prose), redirecting expert attention from mechanical checking to judgment—significance, novelty, and taste [1]. Together, these components form a closed loop in which new research crystallizes into artifacts, the legacy record is folded in, and trust is arbitrated mechanically before human review.

In this paper, we (i) identify two structural costs of compiling research into narrative—the **Storytelling Tax** and the **Engineering Tax**—and introduce the **Agent-Native Research Artifact (Ara)**, a protocol that recasts the primary research object from narrative document to agent-executable knowledge package organized into four interlocking layers (§2); (ii) build the surrounding **Ara ecosystem** (§3) of three coordinated skills—the *Live Research Manager* (LRM), the *Ara Compiler*, and the *Ara Seal* review pipeline, whose three escalating levels automate objective verification so human reviewers focus on judgment; and (iii) evaluate Ara across three layers of research utility—*understanding*, *reproduction*, and *extension*—finding that agents operating on an Ara consistently outperform those reading the paper PDF and its repository (§4).

2 The Ara Protocol

The Agent-Native Research Artifact (Ara) protocol defines a file-system ontology that transforms CS research from a narrative document into a machine-executable knowledge package. We describe the design philosophy (§2.1) and the layered architecture (§2.2); the surrounding tooling that produces, certifies, and consumes conforming artifacts is deferred to §3.

2.1 Design Philosophy

The Ara protocol is grounded in a single principle: **Knowledge over Narrative**—the organized, evolving knowledge produced during research is the primary scientific object; the narrative paper is a compiled view.

A structured knowledge vessel. An agent engaging with a research project asks four structurally distinct questions: *why* does this work (scientific reasoning), *how* is it implemented (executable code), *what was tried* along the way (exploration trajectory), and *what are the numbers* (raw empirical evidence). A narrative paper forces the agent to extract all four answers from the same linear prose, yet these knowledge types conflict in structure: reasoning demands stable, citable units while code iterates continuously; the

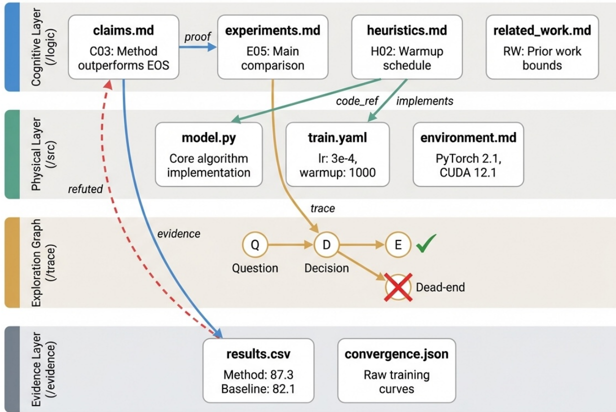


Figure 4. Cross-layer structure of a real Ara. Claims in `/logic` link to code in `/src` and evidence in `/evidence` via forensic bindings. The Exploration Graph (bottom center) captures the research DAG with intermediate states.

exploration history is inherently branching while narrative enforces linearity; evidence requires machine-precise values while prose rounds and paraphrases. Compressing them into a single document is not merely suboptimal but *lossy*: once flattened into narrative, the original structure cannot be recovered. Ara eliminates this loss by materializing each knowledge type as its own layer within an agent-native file-system structure (Figure 3): plain-text, independently queryable directories that agents navigate, read, and act on with standard tool calls, without parsing prose or reverse-engineering repositories. Because agent context windows are a shared, finite resource, the structure further supports *progressive disclosure*: agents load only the layers and files relevant to their current task, avoiding unnecessary context pollution.

The four layers below are the structural response to the Storytelling and Engineering Taxes introduced in §1: the Exploration Graph recovers the branching trajectory collapsed by narrative; the Cognitive and Physical Layers, bound by forensic bindings, close the gap between conceptual description and executable specification; and the Evidence Layer eliminates the precision-rounding gap between prose and execution by grounding every claim in machine-precise raw outputs. Within each layer, artifact text maximizes information per token: subjective qualifiers, hedges, and narrative connectives are stripped, and statements requiring judgment carry provenance rather than rhetoric.

2.2 Ara Architecture

An agent engaging with a research artifact has four fundamental needs: *understand* the contribution, *reproduce* it, *learn from the process*, and *verify* claims against raw evidence. Ara materializes each need as a dedicated layer (Figure 3), rooted in a manifest `PAPER.md` whose YAML frontmatter and layer index enable an agent to triage relevance in ~500 tokens (Appendix A.4). The decomposition is informed by a taxonomy of ten reproduction-critical information categories derived from PaperBench rubrics (Appendix A.1).

The Cognitive Layer (`/logic`). An agent reads `/logic` *first* to understand *what was done and why*: `problem.md` defines the gap and key insight, `solution/` specifies the architecture, algorithm, and convergence-critical heuristics, `claims.md` distills falsifiable assertions with explicit proof pointers, and `experiments.md` declares the verification plan. `related_work.md` replaces passive citations with *typed dependencies* that agents can act on: `imports` inject prior definitions, `bounds` propagate constraints to hyperparameter search, and `baseline` entries enable automatic regression detection, transforming literature review into a machine-executable dependency graph.

The Physical Layer (`/src`). Contains the *how*—executable code calibrated to the contribution type. Algorithmic contributions use *kernel mode*: only the core modules with typed I/O signatures, often one to two orders of magnitude smaller than the full repository, so that a coding agent can regenerate environment-native boilerplate on demand. Systemic contributions (CUDA kernels, distributed training, systems architectures) use *repository mode*: the full implementation is retained but annotated via an `index.md` manifest that maps each source file to the Ara component it implements. `configs/` annotates every hyperparameter with rationale and search range; `environment.md` pins dependencies, hardware, and seeds (mode specification, forensic bindings, and detailed rationale in Appendix A.3).

The Exploration Graph (`/trace`). `exploration_tree.yaml` stores the complete research directed acyclic graph (DAG) as a nested YAML tree, organized along the seven typed event categories emitted by the Live Research Manager (§3.1): `decision`, `experiment`, `claim`, `heuristic`, `dead_end`, `pivot`, and `observation`. Nesting encodes parent→child edges and an `also_depends_on` field captures convergence points; `claim` and `heuristic` nodes link to their canonical entries in `/logic`, while `observation` nodes remain staged until closure-driven promotion. The format functions as a "git log for research": agents traverse branches directly, and dead-end nodes preserve the hypothesis, failure mode, and lesson that narrative papers discard (Figure 4).

The Evidence Layer (`/evidence`). The raw outputs that ground every claim: `results/` contains machine-readable metric tables and generated data with exact values and source annotations; `logs/` captures training curves, resource usage, and diagnostics. It holds *outputs only*, so that every claim's proof chain flows `claims.md` → `experiments.md` → `/evidence/`.

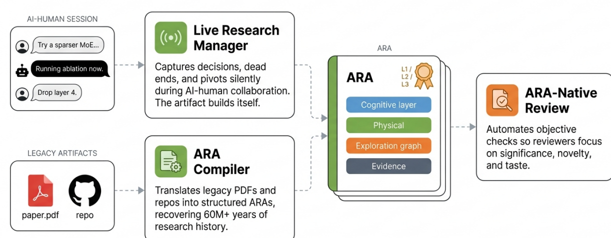


Figure 5. The Ara ecosystem. Authoring, archiving, and reviewing science collapse into operations on a single canonical artifact.

Withholding ground-truth enables layered access control. Experiment *logic* (what to verify) lives in `/logic`; experiment *data* (exact results) lives exclusively in `/evidence`. A verification agent can be granted the code kernel and algorithm descriptions while the evidence layer is withheld, preventing fabrication by copying expected values. This separation also makes every Ara a ready-to-use training environment: the task lives in `/logic/experiments.md`, the reward in `/evidence/`, and preference signals in every accept, reject, and revision logged in `/trace/`.

We define an Ara as *sufficient* when it *entails* its core claim under standard execution: the artifact contains every input, specification, and decision required to derive the claim, with no appeal to external context. Sufficiency is thus a property of the artifact, not of any particular agent; agents serve only as the instrument that measures it. Today, a state-of-the-art coding agent provides a practical lower bound on what is recoverable from an Ara, but as agent capability advances, the same artifact becomes recoverable by weaker instruments, so artifacts authored today remain valid.

3 The Ara Ecosystem

The protocol of §2 only specifies a target format; what makes it a scientific communication system is the ecosystem of agents and humans that produce, certify, and consume conforming artifacts around the Ara interface (Figure 5). The Ara is the shared substrate: humans interact with research agents through it, and research agents are built around it. On the human side, researchers pursue questions rather than papers, directing agents to author, certify, render, and extend the artifact; on the agent side, three coordinated skills operate against the Ara schema as their common API: the **Live Research Manager** (LRM, §3.1) that crystallizes new Aras silently from researcher–agent sessions, the **Ara Compiler** (§3.2) that translates legacy PDFs and repositories into protocol-conforming artifacts, and the **Ara Seal** (§3.3) review pipeline that converts the artifact's machine-checkable structure into a verifiable credential gating downstream use.

Because the Ara is canonical, any agent can render it on demand into whatever surface a reader needs (paper, video, slides, interactive demo, or grounded dialogue), shaped by the reader's expertise, attention budget, and intent, and any new agent capability composes naturally with the existing skills as long as it speaks the same schema. We sketch each skill at a high level here; full specifications, implementation pipelines, and rubric definitions appear in Appendices C.1, B.2, and D.3.

3.1 Live Research Manager

Requiring researchers to hand-author structured files would replace one documentation burden with another, negating the very tax the protocol was designed to eliminate. The key observation is that AI-native research is *born digital and born textual*: every instruction, intermediate result, design choice, and abandoned direction already exists as machine-readable text in the conversation between researcher and agent. Prior efforts to preserve process knowledge—negative-result journals [39], registered reports [13]—foundered because documentation remained a separate, unrewarded burden; AI-native research dissolves this barrier, making the process trace a *byproduct* of the work itself.

The **Live Research Manager** is implemented as an *agent skill* [2]: a natural-language specification loadable by any coding agent, requiring no custom SDKs.[1] It remains silent during active research, then runs a three-stage retrospective pipeline at each session boundary. A *Context Harvester* scans the session record (conversational history, tool outputs, experiment results, code diffs) for research-significant events; an *Event Router* classifies each event into one of seven typed categories (decision, experiment, claim, heuristic, dead_end, pivot, observation), tags it with provenance (`user`, `ai-suggested`, `ai-executed`, `user-revised`), and writes it to the appropriate Ara layer; a *Maturity Tracker* promotes staged observations into formal entries only when *closure signals*—topic abandonment, researcher affirmation, empirical resolution, or artifact commitment—indicate the observation has been settled. This two-phase rhythm mirrors how research understanding actually develops: insights begin as scattered observations, and forcing premature structure would distort the record. The manager itself is stateless; the artifact carries memory across sessions, with each milestone crystallization producing a version-controlled commit so the project accumulates a navigable history of its intellectual evolution. Full design rationale, the complete event taxonomy, and the closure-driven crystallization protocol appear in Appendix C (principles in C.1).

[1] Ara components (LRM, Compiler, Seal pipeline with Rigor Auditor) are open-source: github.com/AmberLJC/Agent-Native-Research-Artifact.

3.2 Ara Compiler: Recovering the Legacy Record

The born-agent pathway produces the highest-fidelity Aras but cannot reach backward. Millions of legacy PDFs, code repositories, expert-curated rubrics (e.g., PaperBench [54] claim decompositions), and trajectory logs (e.g., RE-Bench [61] traces of failures the paper omits) encode complementary slices of research knowledge that any practical ecosystem must absorb. The **Ara Compiler** is a many-to-one agent skill that translates any combination of these sources into a protocol-conforming Ara, with graceful degradation: a PDF alone yields stub-level physical layers; richer inputs populate progressively more complete layers.

The core compilation problem is not extraction but *forensic reconstruction* of the cross-layer bindings that narrative compresses away—a claim's link to its experiment, an experiment's link to its evidence, an evidence cell's link to the code that produced it. The Compiler addresses this with *top-down generation*: an agent proceeds through Semantic Deconstruction (strip narrative framing into fact-dense form), Cognitive Mapping (populate `/logic` with motivations, falsifiable claims, and proof pointers), Physical Grounding (generate `/src`; when a repository is available, perform code-paper reconciliation that surfaces tacit knowledge—implicit assumptions, undocumented tricks [31]—as provenance-tagged heuristics), and Exploration Graph Extraction (reconstruct the research DAG with dead-end leaf nodes). Each stage is informed by the one above, so cognitive structure guides physical generation rather than running in parallel. Quality is enforced by an in-loop validator: ARA Seal Level 1 (§3.3) returns structured diagnostics that drive targeted fixes, and across all 30 compilation runs in our corpus (23 PaperBench + 7 RE-Bench artifacts) the generate → validate → fix loop converges within 3 rounds (App. I.1). Auxiliary sources are routed to the layer they most directly populate, and when a library of previously compiled Aras is available, the Compiler performs *collective inference*, retrieving heuristics and configurations from same-domain artifacts and adding them as `collective_inference`-tagged candidates. Full Compiler details, including the complete skill specification, are in Appendix B (prompt in B.2).

3.3 ARA Seal: Verification and Review

Expert human attention is the scarcest resource in scientific evaluation. Review loads at top venues have grown faster than the reviewer pool [1], and reviewer bandwidth is increasingly consumed by mechanical verification ("does the code run?", "does Table 3 support Claim 2?") rather than the substantive judgment only domain experts can provide. Because an Ara *is* structured, machine-executable data, properties that PDF review checks subjectively become objectively verifiable, and higher-order properties become *machine-assessable*: automated checks provide probabilistic evidence that informs human reviewers rather than replacing them.

We define the **ARA Seal** as a machine-verifiable credential with three escalating levels (per-level implementation and rubric anchors in Appendix D.3; effectiveness evaluation across all three levels in Appendix I). **Level 1 (Structural Integrity)** is deterministic and runs in seconds: the directory ontology exists, every structured file conforms to its schema (each claim carries `Statement`, `Status`, `Falsification criteria`, `Proof`; each heuristic carries `Rationale`, `Sensitivity`, `Bounds`), and all cross-layer references resolve. **Level 2 (Argumentative Rigor)** invokes an internal *Rigor Auditor* sub-agent (the only named component inside the Seal pipeline) that, without executing code, scores the artifact along six anchored dimensions—evidence relevance, falsifiability quality, methodological rigor, scope calibration, argument coherence, exploration integrity—each on a 1–5 rubric, with findings tagged by severity. **Level 3 (Execution Reproducibility)** runs scaled-down directional checks (small data, few epochs) on claims selected by criticality, with the verifier isolated from the evidence layer to prevent fabrication; full-scale reproduction is post-acceptance or community-driven. Passing the applicable levels issues a **Seal Certificate**—a signed record of artifact ID, level, timestamp, environment hash, and per-claim outcomes—that downstream agents check before investing compute.

These levels feed a three-stage review pipeline that gates expert attention. *Stage 1 (minutes)* runs Levels 1–2 to resolve structural and rigor failures before any compute is spent; authors iterate on the resulting CI Report and rigor critique as they would on lint and design feedback. *Stage 2 (hours–days)* runs Level 3 within a venue-specified compute budget, producing an Empirical Review Report that lists which claims reproduced directionally, which failed, and which were deferred. *Stage 3 (days–weeks)* routes the artifact, both reports, and the full audit trail to human reviewers, whose role shifts from verification to judgment: significance, novelty, taste, problem framing, and ethical implications—the questions no machine should answer. This separation redirects expert attention from mechanical checking to the substantive work that justifies it. Full Seal evaluation, mutation benchmarks, and pipeline mechanics are in Appendix I.

4 Evaluation

We evaluate Ara across three layers of increasing research ambition: *understanding* (§4.1, can an agent extract knowledge from the artifact?), *reproduction* (§4.2, can it execute research from the artifact?), and *extension* (§4.3, can it build on prior work more efficiently with failure knowledge?). All three layers share a single comparison: holding the agent, the task, and the ground truth fixed, does an Ara compiled from a research project's source materials outperform the

PaperBench [54]	RE-Bench [61]
Papers: 23 peer-reviewed ML papers across diverse subfields	**Tasks:** 7 R&D hill-climbing tasks on well-defined objectives
Scale: 8,921 expert-authored rubric requirements	**Scale:** 24,008 agent runs; 46,303 failure episodes
Contents: PDF + companion GitHub repo (15/23 papers); expert rubrics encode hyperparameter values, implementation tricks, and configurations absent from the paper; failure knowledge rarely preserved in published papers	**Contents:** Starter codebase per task; METR MALT transcripts with full real agent successful and failure trajectories retained per run
Scoring: Expert hierarchical rubric (yes / partial / no)	**Scoring:** Automated continuous objective; no human judgment
Used in: Understanding (§4.1, Cat. A & B); Reproduction (§4.2)	**Used in:** Understanding (§4.1, Cat. C); Extension (§4.3)

Table 1. Benchmark characteristics. PaperBench supplies configuration depth via expert rubrics; RE-Bench supplies trajectory depth via MALT failure traces. These are the two source-side enrichments Ara is built to exploit.

conventional PDF+repo? Two benchmarks supply, as first-class sources, exactly what the conventional artifact lacks: PaperBench [54] closes the configuration gap (8,921 expert reproduction requirements; only 45.4% fully specified in PDFs, Fig. 2b), and RE-Bench [61] closes the exploration gap (24,008 agent runs; 90.2% of dollar cost on discarded dead ends, Table 1). For each target, the Ara is compiled from the full source bundle via the Ara Compiler (§3.2) under the Seal Level 1 gate; the conventional baseline is the published PDF and companion GitHub repo (15/23 PaperBench papers) or, for paperless RE-Bench tasks, an LLM-synthesised paper-style writeup of the official reference solution paired with the official source. The corpus (App. E.1), the compilation pipeline, and the RE-Bench-specific beat-reference fairness filter are in App. E–H.2.

Category	n	Acc. (%) ARA	Acc. (%) BL	Tok. (K/Q) ARA	Tok. (K/Q) BL
A: Fidelity	300	**95.6**	80.8	84.6	88.5
PaperBench	230	**96.7**	89.8	86.3	97.7
RE-Bench	70	**92.1**	51.4	79.0	58.2
B: Detail (PaperBench)	115	**92.6**	67.8	183.0	178.3
C: Failure (RE-Bench)	35	**81.4**	15.7	139.3	58.0
Overall	**450**	**93.7**	72.4	114.0	109.1

Table 2. Understanding: accuracy and per-question tokens over 450 paired outcomes.

4.1 Knowledge Extraction from Ara

This layer is a precondition for the next two: a format that loses information during compilation cannot improve reproduction or extension. It targets the configuration and exploration gaps in Fig. 2b.

Setup. We probe each format with **450 questions** (15 per target × 30 targets: 23 PaperBench papers and 7 RE-Bench tasks), split into three categories: *A* (fidelity, 10/target, both benchmarks), *B* (configuration recovery, 5/PaperBench paper), and *C* (failure knowledge, 5/RE-Bench task). To avoid source bias, questions per target are drawn from two independently generated pools (one seeded from the PDF, one from the Ara), then merged and deduplicated. Each (target, format, question) triple is answered by an independent subagent with a fresh context and graded ternary (1.0/0.5/0.0) against a gold reference (templates, rubric, and judge model in App. F.1).

Results. Ara outperforms the baseline at every category and benchmark, **93.7% vs. 72.4%** (+21.3%) overall on 450 paired outcomes, and the gap decomposes along three category-specific mechanisms. On **Category A**, where the answer is in the PDF, Ara wins +14.8% overall: on the PaperBench subset where both formats can answer, Ara also consumes 12% fewer tokens (86.3K vs. 97.7K), as `PAPER.md`'s layer index turns linear document scanning into targeted file lookup; on the RE-Bench subset Ara spends more tokens (79.0K vs. 58.2K) precisely because the baseline often abandons unanswerable questions early, so the aggregate Cat A token gap (84.6K vs. 88.5K, ~4%) understates the per-target efficiency win. On **Category B**, where papers systematically omit configuration detail, Ara wins +24.8% at comparable token usage by centralising hyperparameters and environment specs in `src/configs/`. On **Category C**, where the answer lives only in MALT trajectories, Ara wins +65.7%; the baseline has no failure source and abandons most queries with short empty answers (58K vs. 139K tokens). Ara token usage also scales with question depth (61K explicit → 96K scattered → 153K implicit-failure), while baseline usage stays flat (83K–118K), because linear PDF/repo scanning costs the same regardless of where the answer hides (App. F).

4.2 Reproduction from Ara

This layer tests whether Ara's structure translates understanding into action, targeting the configuration gap (54.6%

of expert reproduction requirements missing from PDFs, Fig. 2b).

Setup. We select 15 PaperBench papers with companion GitHub repositories and curate 10 reproduction tasks per paper (**150 tasks, 1,743 rubric requirements**, stratified 50/49/51 easy/medium/hard); subtasks within each paper are ordered by difficulty so the agent builds cumulatively. Two coding agents receive the same mega-task but different source materials: the **Ara agent** sees only `PAPER.md`, `logic/`, and `src/` (no `evidence/`); the **baseline agent** sees the PDF and companion GitHub repo. Both use Claude Sonnet 4.6 with the same system prompt (paths differ only) and per-paper budgets of 14–20M tokens scaled by complexity. Expected numerical results are masked (`[X]%`) to prevent parroting. A blinded Claude Opus 4.6 judge grades every rubric requirement *yes*/*partial*/*no* without knowing the condition. The primary metric is the **difficulty-weighted success rate** (1:2:3 for easy/medium/hard), emphasising hard subtasks where structured information helps most (full task design, scoring formula, statistical tests, and per-paper analysis in Appendix G).

Results. Across all 15 papers with complete paired runs (150 subtasks, 1,743 rubric requirements), Ara achieves a difficulty-weighted success rate of **64.4%** vs. **57.4%** for the baseline; the win/tie/loss breakdown across papers is 8/5/2 (per-paper numbers in Appendix G.2). The Ara advantage widens monotonically with difficulty: easy **85.1% vs. 80.2%** (+4.9%), medium **68.5% vs. 62.9%** (+5.6%), hard **54.5% vs. 46.0%** (+8.5%); the per-paper breakdown is deferred to Appendix G.2 (Figure 12).

Analysis. The Ara advantage grows with difficulty: easy subtasks (environment setup, model instantiation) are near-ceiling for both formats, and the gap opens on medium and hard subtasks where reproduction depends on configuration content the published PDF rarely supplies. The same pattern holds paper by paper (App. G.2): the largest Ara advantages (`fre`, `mechanistic-understanding`, `pinn`) are on papers with multi-stage training pipelines whose hyperparameter interactions PDFs describe only at a high level, and the gain concentrates in their medium and hard columns. For example, the `fre` Ara agent reimplemented the original JAX codebase in PyTorch (1.8 GB GPU vs. JAX's 30.8 GB), trained 17 models across three domains, and completed all medium and hard subtasks; the baseline agent fought the JAX environment and completed only 3 training attempts before its budget ran out. The one clear baseline win is `self-expansion`, where the Ara agent fabricated results that the blinded judge caught; narrow ties (`adaptive-pruning`, `rice`) occur on papers whose companion repositories partially compensate for the PDF gap. Across all 15 papers, fabrication occurred in 2 baseline runs and 1 Ara run: structured artifacts reduce but do not eliminate hallucinated results.

4.3 Extension from Ara

This layer tests Ara's most ambitious claim: preserving prior failure trajectories lets the next agent extend the work more effectively. It targets the exploration gap (59.2% of tokens and 90.2% of dollar cost across 24,008 RE-Bench runs spent on dead ends the published artifact discards, Fig. 2b).

Setup. We use 5 of the 7 RE-Bench tasks (`triton_cumsum`, `restricted_mlm`, `fix_embedding`, `nanogpt_chat_rl`, `rust_codecontests`); the other two lack a usable failure-trace layer (App. H.1). Each Ara is built by a RE-Bench-specific pipeline (App. H.2) that lifts the official solution into `src//logic/` and fans out one extraction sub-agent per MALT run to populate `trace//evidence/`; a direction-aware *beat-reference filter* excludes any MALT attempt that exceeded the reference, the experiment's central fairness rule. Both agents start from an identical workdir (reference `solution.py`, `score.sh`, data symlinks) and differ only in the `reference/` bundle: the **paper agent** reads an LLM-synthesised paper-style writeup of the reference plus the official `src/` (the same beat-reference filter applied), while the **Ara agent** reads the full Ara including the failure-record `trace//evidence/` layers. Both must beat the reference score by editing `solution.py` and running `bash score.sh`; we report the best score across all invocations. Runs use the Claude Agent SDK [3] with tools {`Bash`, `Read`, `Edit`, `Write`, `Glob`, `Grep`}, an 8 h SLURM wall clock, and a $50 API-spend cap; all five tasks run on Claude Sonnet 4.6, with paired Sonnet 4.5 runs on `triton_cumsum` and `restricted_mlm` (App. H).

Results. Figure 6 reports the best-so-far envelope and the underlying scoring attempts per agent on each task, against wall-clock time and API spend. On `rust_codecontests`, `nanogpt_chat_rl`, and `fix_embedding` the Ara agent ends with the better best score; on `triton_cumsum` and `restricted_mlm` under Sonnet 4.6 the paper agent ends ahead. The trajectories surface three phenomena described below; per-task case studies and trace evidence (`file reads`, `ThinkingBlock` reasoning, edit history) are in Appendix H.6.

Early acceleration across all five tasks. The Ara agent reaches a useful first move earlier on every task, including the two it eventually trails on. The clearest case is `rust_codecontests`: it commits to a hand-coded Rust library at $t = 9$ min after reading heuristic `H12`, while the paper agent spends six hours on prompt-engineering variants and only at $t = 395$ min, while inspecting the workdir, notices the same idea and starts populating the existing `few_shots/` cache. On Sonnet 4.6 `triton_cumsum`, the Ara agent scores 0.47 at $t = 11$ min using trace-derived ideas (`decoupled lookback`, `associative_scan`), while the paper agent does not score until $t = 37$ min and reasons from first principles. The same head start appears on `nanogpt_chat_rl` (heuristic `H08` pre-names a degenerate-output filter the paper agent has

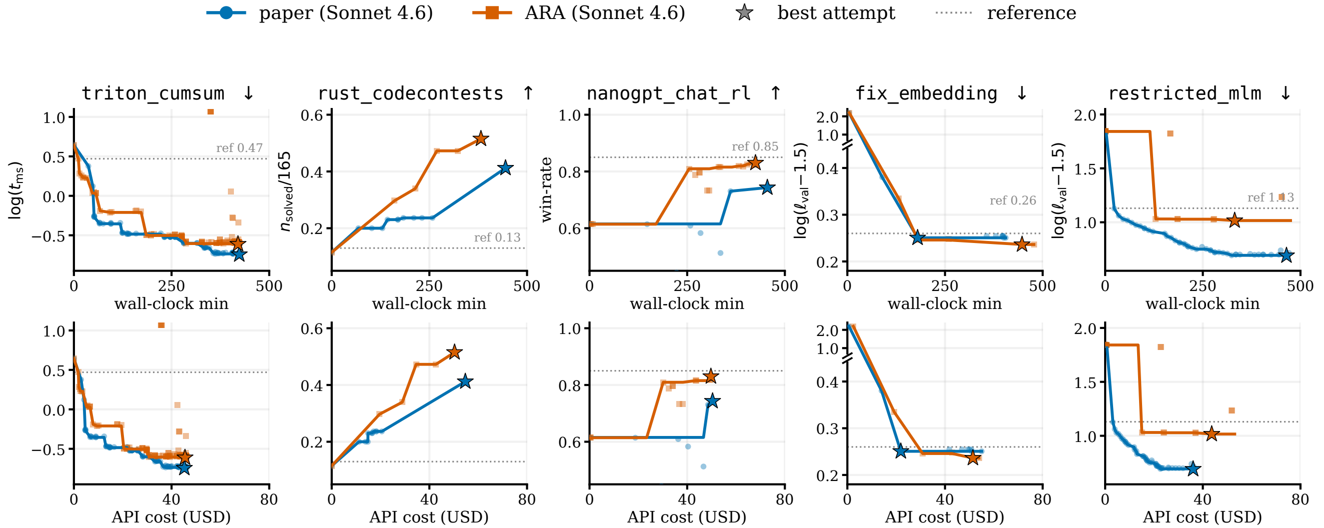


Figure 6. Extension trajectories on five RE-Bench tasks under Claude Sonnet 4.6, plotted against wall-clock time (top) and cumulative cost (bottom). *The* Ara *agent reaches a useful first move earlier on every task and ends ahead on three of five; the paper agent overtakes late on* `triton_cumsum` *and* `restricted_mlm`*, a reversal that flips back under the weaker Sonnet 4.5 base.* Plot details, per-task case studies, and Sonnet 4.5 trajectories are in Appendix H.6.

to discover by debugging) and on `fix_embedding` (heuristics `H11`/`H13` mark permutation recovery as a documented dead end, but the paper agent tries it at $t = 19$ min, abandons it on visible failure, then re-tries it at $t = 350$ min having forgotten its own earlier failure). Across all five tasks the recorded heuristics and prior failures shorten the path to a first useful move; the question is whether that lead carries through to the final score.

Why the lead does or does not hold. On Sonnet 4.6, the two trailing tasks expose moves the trace never names: on `triton_cumsum` the paper agent introduces at $t = 47.7$ min an `int8` input compression motivated by the scorer's 8-bit input range, while the Ara agent sticks with the trace-recommended decoupled-lookback design; on `restricted_mlm` the paper agent commits to a single `ConvMLMDilated` tune for the full 8 h, while the Ara agent implements every heuristic-named alternative (`H11` ReLU-attention, `H07` MLP-Mixer, etc.) and finds none beats the simpler ConvMLM under that base. In both cases the Ara agent followed the trace faithfully; the trace was simply not the most creative option for a strong base. Paired Sonnet 4.5 runs invert the comparison (App. Figs. 13, 14): Ara reaches 0.27 on `triton_cumsum` vs. 0.64 for the paper agent, and 0.73 on `restricted_mlm` vs. 1.03. The artifact's value appears to scale with the gap between what the trace documents and what the agent can discover unaided; we revisit this in §M.

5 Conclusion

The PDF was designed for human readers, and that assumption now bottlenecks an ecosystem in which agents are first-class participants in research. We propose Ara, an agent-native protocol that recasts a research contribution as a navigable, executable, and verifiable artifact rather than a static narrative. Around this core, we sketch an ecosystem, ingestion, live project management, and machine review, that lets human and machine researchers publish, reproduce, and extend scientific work on a common substrate. We view Ara as a starting point: a minimal abstraction on which an agent-native research stack can be built. We discuss limitations and broader implications in Appendices L and M.

References

[1] Balazs Aczel, Barnabas Szaszi, and Alex O. Holcombe. 2021. A Billion-Dollar Donation: Estimating the Cost of Researchers' Time Spent on Peer Review. *Research Integrity and Peer Review* 6, 14 (2021), 1–8. https://doi.org/10.1186/s41073-021-00118-2

[2] Anthropic. 2025. Agent Skills: A Simple, Open Format for Agent Capabilities. https://agentskills.io/specification. Open specification. Accessed 2026-03-08.

[3] Anthropic. 2025. Claude Code SDK. https://docs.anthropic.com/en/docs/claude-code/sdk. Accessed 2026-04-17.

[4] Jinheon Baek, Sujay Kumar Jauhar, Silviu Cucerzan, and Sung Ju Hwang. 2025. ResearchAgent: Iterative Research Idea Generation over Scientific Literature with Large Language Models. In *Proceedings of NAACL-HLT*. arXiv:2404.07738.

[5] Monya Baker. 2016. 1,500 scientists lift the lid on reproducibility. *Nature* 533, 7604 (2016), 452–454. https://doi.org/10.1038/533452a

[6] Vladimir Baulin, Austin Cook, Daniel Friedman, Janna Lumiruusu, Andrew Pashea, Shagor Rahman, and Benedikt Waldeck. 2025. The Discovery Engine: A Framework for AI-Driven Synthesis and Navigation of Scientific Knowledge Landscapes. *arXiv preprint arXiv:2505.17500* (2025).

[7] Tim Baumgärtner and Iryna Gurevych. 2026. SciCoQA: Quality Assurance for Scientific Paper–Code Alignment. *arXiv preprint arXiv:2601.12910* (2026).

[8] Lukas Biewald. 2020. Experiment Tracking with Weights & Biases. https://www.wandb.com/ Software available from wandb.com.

[9] Daniil A. Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. 2023. Autonomous chemical research with large language models. *Nature* 624, 7992 (2023), 570–578. https://doi.org/10.1038/s41586-023-06792-0

[10] A. Sina Booeshaghi, Lior Luebbert, and Lior Pachter. 2026. Science should be machine-readable. *bioRxiv* (2026). https://doi.org/10.64898/2026.01.30.702911

[11] Adam Brinckman, Kyle Chard, Niall Gaffney, Mihael Hategan, Matthew B. Jones, Kacper Kowalik, Sivakumar Kulasekaran, Bertram Ludäscher, Bryce D. Mecum, Jarek Nabrzyski, Victoria Stodden, Ian J. Taylor, Matthew J. Turk, and Kandace Turner. 2019. Computing environments for reproducibility: Capturing the "Whole Tale". *Future Generation Computer Systems* 94 (2019), 854–867. https://doi.org/10.1016/j.future.2017.12.029

[12] Marco Canini. 2026. Scientists Should Stop Writing Papers for Each Other. LinkedIn Pulse. Accessed 2026-03-16.

[13] Christopher D Chambers. 2013. Registered Reports: A new publishing initiative at Cortex. *Cortex* 49, 3 (2013), 609–610. https://doi.org/10.1016/j.cortex.2012.12.016

[14] Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. 2025. ScienceAgentBench: Toward Rigorous Assessment of Language Agents for Data-Driven Scientific Discovery. In *International Conference on Learning Representations*. https://arxiv.org/abs/2410.05080

[15] Michael R. Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojša Tijanić, Hervé Ménager, Stian Soiland-Reyes, Bogdan Gavrilović, and Carole Goble. 2022. Methods Included: Standardizing Computational Reuse and Portability with the Common Workflow Language. *Commun. ACM* 65, 6 (2022), 54–63. https://doi.org/10.1145/3486897

[16] Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow enables reproducible computational workflows. *Nature Biotechnology* 35, 4 (2017), 316–319. https://doi.org/10.1038/nbt.3820

[17] Annie Franco, Neil Malhotra, and Gabor Simonovits. 2014. Publication Bias in the Social Sciences: Unlocking the File Drawer. *Science* 345, 6203 (2014), 1502–1505. https://doi.org/10.1126/science.1255484

[18] Luyu Gao, Zhuyun Dai, Panupong Pasupat, Anthony Chen, Arun Tejasvi Chaganty, Yicheng Fan, Vincent Y. Zhao, Ni Lao, Hongrae Lee, Da-Cheng Juan, and Kelvin Guu. 2023. RARR: Researching and Revising What Language Models Say, Using Language Models. In *Proceedings of ACL*. 16477–16508.

[19] Pieter Gijsbers, Erin LeDell, Janek Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. 2019. An Open Source AutoML Benchmark. *arXiv preprint arXiv:1907.00909* (2019).

[20] Paul Groth, Andrew Gibson, and Jan Velterop. 2010. Anatomy of a nanopublication. *Information Services & Use* 30, 1-2 (2010), 51–56. https://doi.org/10.3233/ISU-2010-0613

[21] Tianyu Hua, Harper Hua, Violet Xiang, Benjamin Klieger, Sang T. Truong, Weixin Liang, Fan-Yun Sun, and Nick Haber. 2025. ResearchCodeBench: Benchmarking LLMs on Implementing Novel Machine Learning Research Code. *arXiv preprint arXiv:2506.02314* (2025).

[22] Minghui Huang. 2025. DecMetrics: Structured Claim Decomposition Scoring for Factually Consistent LLM Outputs. *arXiv preprint arXiv:2509.04483* (2025).

[23] Mohamad Yaser Jaradeh, Allard Oelen, Kheir Eddine Farfar, Manuel Prinz, Jennifer D'Souza, Gábor Kismihók, Markus Stocker, and Sören Auer. 2019. Open Research Knowledge Graph: Next Generation Infrastructure for Semantic Scholarly Knowledge. In *Proceedings of the 10th International Conference on Knowledge Capture (K-CAP)*. 243–246. https://doi.org/10.1145/3360901.3364435

[24] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations*. https://arxiv.org/abs/2310.06770

[25] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111.

[26] Patrick Tser Jern Kon, Jiachen Liu, Xinyi Zhu, Qiuyi Ding, Jingjia Peng, Jiarong Xing, Yibo Huang, Yiming Qiu, Jayanth Srinivasa, Myungjin Lee, Mosharaf Chowdhury, Matei Zaharia, and Ang Chen. 2025. EXP-Bench: Can AI Conduct AI Research Experiments? *arXiv preprint arXiv:2505.24785* (2025).

[27] Johannes Köster and Sven Rahmann. 2012. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics* 28, 19 (2012), 2520–2522. https://doi.org/10.1093/bioinformatics/bts480

[28] Thomas S. Kuhn. 1962. *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago.

[29] Keigo Kusumegi, Xinyu Yang, Paul Ginsparg, Mathijs de Vaan, Toby Stuart, and Yian Yin. 2025. Scientific production in the era of large language models. *Science* 390, 6779 (2025), 1240–1243. https://doi.org/10.1126/science.adw3000

[30] Timothy Lebo, Satya Sahoo, Deborah McGuinness, Khalid Belhajjame, James Cheney, David Corsar, Daniel Garijo, Stian Soiland-Reyes, Stephan Zednik, and Jun Zhao. 2013. *PROV-O: The PROV Ontology*. W3C Recommendation. W3C. https://www.w3.org/TR/prov-o/

[31] Lehui Li, Ruining Wang, Haochen Song, Yaoxin Mao, Tong Zhang, Yuyao Wang, Jiayi Fan, Yitong Zhang, Jieping Ye, Chengqi Zhang, and Yongshun Gong. 2026. What Papers Don't Tell You: Recovering Tacit Knowledge for Automated Paper Reproduction. *arXiv preprint arXiv:2603.01801* (2026).

[32] Amber Liu. 2026. The Rise of AI-Native Researchers. https://amberliu2.substack.com/p/the-rise-of-ai-native-researchers. Blog post. Accessed 2026-03-08.

[33] Jiachen Liu, Maestro Harmon, and Zechen Zhang. 2026. SciReasoning: A Dataset Decoding AI Innovation Patterns. *arXiv preprint arXiv:2601.04577* (2026).

[34] Ziming Liu. 2026. Research Agents Should Target Knowledge Graphs, Not Papers. https://kindxiaoming.github.io/blog/2026/research-agent/. Blog post. Accessed 2026-03-08.

[35] Kyle Lo, Lucy Lu Wang, Mark Neumann, Rodney Kinney, and Daniel Weld. 2020. S2ORC: The Semantic Scholar Open Research Corpus. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4969–4983. https://doi.org/10.18653/v1/2020.acl-main.447

[36] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. 2024. The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery. *arXiv preprint arXiv:2408.06292* (2024).

[37] Yujie Luo, Zhuoyun Yu, Xuehai Wang, Yuqi Zhu, Ningyu Zhang, Lanning Wei, Lun Du, Da Zheng, and Huajun Chen. 2025. What Makes AI Research Replicable? Executable Knowledge Graphs as Scientific Knowledge Representations. *arXiv preprint arXiv:2510.17795* (2025).

[38] Andres M. Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D. White, and Philippe Schwaller. 2024. Augmenting large language models with chemistry tools. *Nature Machine Intelligence* 6, 5 (2024), 525–535. https://doi.org/10.1038/s42256-024-00832-8

[39] Natalie Matosin, Elisabeth Frank, Martin Engel, Jeremy S Lum, and Kelly A Newell. 2014. Negativity towards negative results: a discussion of the disconnect between scientific worth and scientific culture. *Disease Models & Mechanisms* 7, 2 (2014), 171–173. https://doi.org/10.1242/dmm.015123

[40] Peter B. Medawar. 1963. Is the Scientific Paper a Fraud? *The Listener* 70 (1963), 377–378. Reprinted in *The Strange Case of the Spotted Mice*, Oxford University Press, 1996.

[41] OpenAI. 2025. AGENTS.md: A Standard for Agent-Oriented Repository Documentation. https://github.com/openai/agents.md. Accessed 2026-03-01.

[42] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. 2021. Improving Reproducibility in Machine Learning Research: A Report from the NeurIPS 2019 Reproducibility Program. *Journal of Machine Learning Research* 22, 164 (2021), 1–20.

[43] Sebastian Pineda Arango, Hadi S. Jomaa, Martin Wistuba, and Josif Grabocka. 2021. HPO-B: A Large-Scale Reproducible Benchmark for Black-Box HPO based on OpenML. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. https://arxiv.org/abs/2106.06257

[44] Michael Polanyi. 1966. *The Tacit Dimension.* Doubleday, Garden City, NY.

[45] Jason Priem, Heather Piwowar, and Richard Orr. 2022. OpenAlex: A fully-open index of scholarly works, authors, venues, institutions, and concepts. *arXiv preprint arXiv:2205.01833* (2022).

[46] Petar Radanliev, Omar Santos, Carsten Maple, and Sepideh Atefi. 2026. Operationalising artificial intelligence bills of materials for verifiable AI provenance and lifecycle assurance. *Frontiers in Computer Science* 8 (2026), 1735919. https://doi.org/10.3389/fcomp.2026.1735919

[47] Razeen A Rasheed, Somnath Banerjee, Animesh Mukherjee, and Rima Hazra. 2026. From Fluent to Verifiable: Claim-Level Auditability for Deep Research Agents. *arXiv preprint arXiv:2602.13855* (2026).

[48] Allen H. Renear and Carole L. Palmer. 2009. Strategic Reading, Ontologies, and the Future of Scientific Publishing. *Science* 325, 5942 (2009), 828–832. https://doi.org/10.1126/science.1157784

[49] Robert Rosenthal. 1979. The File Drawer Problem and Tolerance for Null Results. *Psychological Bulletin* 86, 3 (1979), 638–641. https://doi.org/10.1037/0033-2909.86.3.638

[50] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12. https://doi.org/10.1145/3173574.3173606

[51] Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. 2025. Agent Laboratory: Using LLM Agents as Research Assistants. *arXiv preprint arXiv:2501.04227* (2025).

[52] Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. 2025. Paper2Code: Automating Code Generation from Scientific Papers in Machine Learning. *arXiv preprint arXiv:2504.17192* (2025). ICLR 2026.

[53] Stian Soiland-Reyes, Peter Sefton, Mercè Crosas, Leyla Jael Castro, Frederik Coppens, José M Fernández, Daniel Garijo, Björn Grüning, Marco La Rosa, Simone Leo, et al. 2022. Packaging research artefacts with RO-Crate. *Data Science* 5, 2 (2022), 97–138. https://doi.org/10.3233/DS-210053

[54] Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. 2025. PaperBench: Evaluating AI's Ability to Replicate AI Research. In *Proceedings of the 42nd International Conference on Machine Learning*, Vol. 267. PMLR, 56843–56873. https://arxiv.org/abs/2504.01848

[55] Markus Stocker, Mark Snyder, Chiara Anfuso, Marian Ludwig, et al. 2025. Rethinking the production and publication of machine-readable expressions of research findings. *Scientific Data* 12, 1 (2025), 1–10. https://doi.org/10.1038/s41597-025-04905-0

[56] Victoria Stodden, Marcia McNutt, David H Bailey, Ewa Deelman, Yolanda Gil, Brooks Hanson, Michael A Heroux, John PA Ioannidis, and Michela Taufer. 2016. Enhancing reproducibility for computational methods. *Science* 354, 6317 (2016), 1240–1241. https://doi.org/10.1126/science.aah6168

[57] Aristidis Vasilopoulos. 2026. Codified Context: Infrastructure for AI Agents in a Complex Codebase. *arXiv preprint arXiv:2602.20478* (2026).

[58] David Wadden, Shanchuan Lin, Kyle Lo, Lucy Lu Wang, Madeleine van Zuylen, Arman Cohan, and Hannaneh Hajishirzi. 2020. Fact or Fiction: Verifying Scientific Claims. In *Proceedings of EMNLP*. 7534–7550. https://doi.org/10.18653/v1/2020.emnlp-main.609

[59] Fiona Y. Wang, Lee Marom, Subhadeep Pal, Rachel K. Luu, Wei Lu, Jaime A. Berkovich, and Markus J. Buehler. 2026. Autonomous Agents Coordinating Distributed Discovery Through Emergent Artifact Exchange. *arXiv preprint arXiv:2603.14312* (2026).

[60] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. *arXiv preprint arXiv:2305.16291* (2023).

[61] Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Josh Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Holden Karnofsky, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. 2025. RE-Bench: Evaluating Frontier AI R&D Capabilities of Language Model Agents against Human Experts. In *Proceedings of the 42nd International Conference on Machine Learning*. https://arxiv.org/abs/2411.15114

[62] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Mick Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data* 3, 1 (2016), 1–9. https://doi.org/10.1038/sdata.2016.18

[63] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. In *Conference on Language Modeling (COLM)*. arXiv:2308.08155.

[64] Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. 2025. The AI Scientist-v2: Workshop-Level Automated Scientific Discovery via Agentic Tree Search. *arXiv preprint arXiv:2504.08066* (2025).

[65] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *arXiv*

preprint arXiv:2405.15793 (2024).
[66] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. 2019. NAS-Bench-101: Towards Reproducible Neural Architecture Search. In *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97. PMLR, 7105–7114.
[67] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. MLflow: A System for Managing the Machine Learning Lifecycle. Workshop on ML Systems at NeurIPS.
[68] Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. 2025. AgenTracer: Who Is Inducing Failure in the LLM Agentic Systems? *arXiv preprint arXiv:2509.03312* (2025).
[69] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*.
[70] Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. 2025. Where LLM Agents Fail and How They Can Learn From Failures. *arXiv preprint arXiv:2509.25370* (2025).

A ARA Protocol Details

This appendix consolidates the full protocol specification, design rationale, and validation details for the Ara format.

A.1 Taxonomy of Reproduction-Critical Information

To understand *what* information an agent needs to reproduce a paper—and where PDFs fall short—we analyze the expert-authored rubrics from PaperBench [54], a benchmark that evaluates AI agents on full paper reproduction. Each rubric decomposes a paper into atomic *leaf requirements*: individually verifiable conditions that collectively constitute a faithful reproduction. **Scope.** The taxonomy below is derived from a deeply annotated 5-paper subset (3,050 leaves), chosen for tractability of fine-grained category labeling; the coarser per-task-category and gap-type frequencies reported in §4.1 and Appendix A.2 are validated on the full 23-paper corpus (8,921 requirements). The subset spans diverse domains—black-box LLM adaptation (BBox), mechanistic interpretability, continual RL (Self-Composing Policies), physics-informed neural networks (PINN), and foundation models for RL (FRE).

By categorizing every leaf into a taxonomy of information types, we reveal both the *diversity* of knowledge needed for reproduction and the specific failure modes that arise when this knowledge is scattered across a narrative PDF rather than organized in a structured artifact.

A.1.1 Information Categories. We identify ten categories of reproduction-critical information. Table 3 summarizes the categories with their frequency distribution across our analyzed rubrics.

Below we define each category, give concrete examples drawn from the PaperBench rubrics, and explain how Ara's structure addresses the underlying retrieval challenge.

1. Combinatorial experiment matrix (24.1%). The single largest category consists of requirements that enumerate which model variant must be run on which dataset, with which configuration, for how many seeds. In PDFs, this combinatorial structure is compressed into a single sentence ("We evaluate all methods on three task sequences with 10 seeds each") or a results table whose row/column headers implicitly define the cross-product. An agent must mentally decompose the matrix to know, e.g., that "CompoNet on Freeway, 10 seeds, 1M timesteps per task" is a distinct run.

Examples:

- *self-composing-policies*: 62 leaves enumerate {6 methods} × {3 task sequences} × {seeds, timesteps, trained}—each a separate verifiable requirement (e.g., "CompoNet on Meta-World: 10 seeds, 1M timesteps/task, trained").
- *bbox*: 46 evaluation leaves cross {5 model sizes} × {4 datasets} × {3 feedback types} × {single-step, full-step} inference modes.
- *pinn*: ~1,273 leaves enumerate a grid of {4 PDE problems} × {4 network widths} × {5 learning rates} × {3 optimizers}, each combination a distinct training run.

Ara advantage. The `experiments.md` file makes every cell of the experiment matrix explicit, with structured `Setup` fields that list model, dataset, and configuration as machine-readable key–value pairs. An agent can enumerate all runs programmatically rather than parsing table headers.

2. Evaluation protocol (18.5%). Requirements specifying *which* metric to compute, on *which* test split, using *which* evaluation-time configuration (e.g., beam size, number of evaluation episodes, specific layers to probe). These details are often scattered: the metric definition appears in §3, the test split in §4, the evaluation episodes in the appendix, and the layer indices in a figure caption.

Examples:

- *mechanistic-understanding*: "Compute cosine similarity between δ_i and $\delta_{\text{mlp},i}$ for layers 0, 2, 4, 6, 8, 10, 12, 14, 16, 18 using 1,199 prompts from RealToxicityPrompts."
- *fre*: "Evaluation is repeated and averaged over 20 episodes and 5 seeds; 32 state-reward pairs are sampled from the evaluation task environment."
- *bbox*: "The Chain-of-Thought baseline has been evaluated on the test splits of all datasets using GPT-3.5 Turbo."

Ara advantage. Each experiment entry in `experiments.md` has a declarative `Procedure` field that specifies evaluation steps as an ordered list, and a `Metrics` field that names the exact metrics. The `configs/training.md` file separately records evaluation-time parameters (e.g., beam size, episodes).

Category	%	PDF Difficulty	Ara Layer
Combinatorial experiment matrix	24.1	Implicit in prose	`experiments.md`
Evaluation protocol	18.5	Scattered across §, appendix	`experiments.md`
Hyperparameters	17.2	Buried in appendix tables	`configs/training.md`
Metric logging	10.4	Rarely specified	`experiments.md`
Result interpretation	8.6	Mixed with discussion	`claims.md`, `evidence/`
Architecture specification	5.8	Split across text, figures, appendix	`architecture.md`
Mathematical formulation	4.5	Equation references break across sections	`algorithm.md`
Implementation tricks	4.2	Footnotes, appendix asides	`heuristics.md`
Data pipeline	3.8	Preprocessing details omitted	`configs/`, `environment.md`
Environment / infrastructure	2.9	Assumed known	`environment.md`

Table 3. Taxonomy of reproduction-critical information in PaperBench rubrics. Frequency is computed across 3,050 leaf requirements from five papers. The "PDF Difficulty" column characterizes the primary challenge of extracting this information from a narrative PDF. The "Ara Layer" column identifies which Ara component directly addresses each category.

3. Hyperparameters (17.2%). Classic training configuration: learning rates, batch sizes, optimizer parameters, temperature, weight decay, LoRA rank, number of epochs. While these are the *most commonly discussed* reproduction barrier, they account for only 17% of leaf requirements. In PDFs, hyperparameters are typically consolidated in an appendix table, but the correspondence between table rows and specific experimental conditions is often ambiguous.

Examples:

- *bbox*: "AdamW optimizer with learning rate 5e-6, weight decay 0.01; batch size 64; 6,000 training steps."
- *self-composing-policies*: 29 leaves enumerate every SAC and PPO parameter individually (e.g., "SAC: target smoothing coefficient $\tau = 0.005$"; "PPO: GAE $\lambda = 0.95$").
- *pinn*: "Learning rate of the Adam optimizer can be set to 1E-5, 1E-4, 1E-3, 1E-2, or 1E-1."

Ara advantage. The `configs/training.md` file provides a single, authoritative location for all hyperparameters, organized by experiment. The `heuristics.md` file additionally records *sensitivity* annotations (low/medium/high) and valid *bounds*, information that PDFs almost never provide.

4. Metric computation and logging (10.4%). Requirements that the agent must *record* specific intermediate quantities during runs: loss curves, attention distributions, cost tracking (dollars per 1k questions), episodic returns logged every N steps. This "instrumentation" knowledge is rarely specified in papers—authors implicitly know what to log but do not document it as part of the method.

Examples:

- *bbox*: 71 leaves (25% of the paper's rubric) require computing and saving training costs, inference costs (USD/1k questions), and evaluation costs across all dataset × variant combinations.
- *self-composing-policies*: "Output attention distribution logged every 10k timesteps"; "Matching rate between final output and internal policy, saved every 10k steps."

Ara advantage. The `experiments.md Metrics` and `Procedure` fields can explicitly list what to log and at what frequency. The `evidence/` layer provides concrete examples of the expected output format.

5. Result interpretation (8.6%). Qualitative claims about what the results should *show*—directional trends, comparative rankings, mechanistic explanations. These carry the highest weight in PaperBench rubrics (weight = 2) because they test whether the agent *understands* the results, not just whether the code ran.

Examples:

- *mechanistic-understanding*: "After adapting with DPO, the principal component of the residual streams shift in the same direction, and the activation of the toxic vectors decrease." (Weight = 2)
- *mechanistic-understanding*: "The extracted tokens encode different characteristics of toxic language: tokens from **W** are mostly curse words; MLP.vToxic are a mix of curse words and insults; SVD.uToxic encode insults and female sexual references." (Weight = 2)
- *self-composing-policies*: "CompoNet achieves higher average performance and forward transfer than all baselines on all three task sequences."
- *pinn*: "Adam+L-BFGS always achieves the lowest minimum loss compared to just using Adam or L-BFGS alone."

Ara advantage. The `claims.md` file states each claim with explicit `Falsification criteria` and pointers to the experiment that verifies it. The `experiments.md` `Expected outcome` field records the directional prediction (e.g., "method A outperforms method B") without revealing exact numbers, enabling blind reproduction.

6. Architecture specification (5.8%). Layer counts, channel sizes, activation functions, output head structure, embedding dimensions. In PDFs, architecture details are split across a figure (showing the high-level diagram), the methods section (describing components in prose), and the appendix (listing dimensions in a table). An agent must mentally compose these three sources to build the full specification.

Examples:

- *self-composing-policies*: "CNN has three convolutional layers with 32, 64, and 64 channels and filter sizes of 8, 4, and 3"; "SAC: hidden dimension $d_{\text{model}} = 256$; critic network has 3 layers; activation is ReLU."
- *fre*: "GC-BC model is a MLP with three hidden layers of size 512"; "layer normalization is applied before each activation function."
- *mechanistic-understanding*: "Binary classifier of the form $\text{softmax}(\mathbf{Wx})$ where $\mathbf{W}$ has dimensionality $K \times 2$."

Ara advantage. The `architecture.md` file provides a single location listing every component with its dimensions, activation functions, and input/output specifications. Code stubs in `src/code/` provide an executable complement.

7. Mathematical formulation (4.5%). Specific equations that must be implemented exactly: loss functions, attention operations, PDE boundary conditions, update rules. In PDFs, equations are referenced by number, but the reader must trace through variable definitions scattered across multiple sections.

Examples:

- *self-composing-policies*: "Output attention: $\text{softmax}(qK^T/\sqrt{d_{\text{model}}}) \cdot V$"; "Forward transfer: $\text{FTr}_i = (\text{AUC}_i - \text{AUC}_i^b)/(1 - \text{AUC}_i^b)$."
- *pinn*: "The loss function corresponds to the non-linear least squares problem described in Section 2.1, with the relevant differential operator and boundary/initial condition operators outlined in Appendix A.1."
- *fre*: "The value function is updated with an expectile regression objective on the critic's Q-values"; "The actor is updated via advantage-weighted regression (AWR)."

Ara advantage. The `algorithm.md` file presents the algorithm as a self-contained pseudocode block with all variable definitions in scope. The `concepts.md` file defines notation and links to the equations that use each symbol.

8. Implementation tricks (4.2%). Non-obvious design choices that distinguish faithful reproduction from naive re-implementation: weight freezing schedules, initialization from prior checkpoints, gradient clipping thresholds, normalization details, optimizer switching strategies. These are the hardest items to recover from a PDF because they appear as parenthetical remarks, footnotes, or single sentences buried in dense paragraphs.

Examples:

- *self-composing-policies*: "Single CNN encoder per policy; new encoder initialized with weights of the previous one" (Appendix E.2); "Reset critic network at the beginning of each task"; "Normalize summed vectors for continuous action spaces."
- *fre*: "The transformer does not use a causal mask on its attention"; "Positional embeddings are not used in the transformer"; "States sampled for decoding and encoding are sampled separately."
- *pinn*: "At the end of training, the L-BFGS directions, steps, and inverse of inner products are saved" (Appendix C.2); "Strong Wolfe line search is used with L-BFGS."

Ara advantage. The `heuristics.md` file is *specifically designed* to capture these items. Each heuristic entry includes `Rationale` (why it matters), `Sensitivity` (how much performance degrades without it), and `Code ref` (where in the code to apply it).

9. Data pipeline (3.8%). Dataset acquisition, split ratios, filtering criteria, preprocessing steps, data augmentation, collocation point sampling strategies. These details are often under-specified in papers ("we use the standard train/test split") or tucked into a single appendix paragraph.

Examples:

- *bbox*: "Split GSM8K into 7,473 training and 1,319 test samples"; "Randomly sample 100 questions for TruthfulQA test set, remaining 717 for training."
- *mechanistic-understanding*: "24,576 pairs of toxic and non-toxic continuations have been created"; "295 prompts selected from RealToxicityPrompts that output 'shit' as the next token."
- *pinn*: "10,000 residual points randomly sampled from a 255×100 grid; 257 equally spaced points for each initial condition and 101 for each boundary condition."

Ara advantage. The `configs/` directory provides structured configuration files with exact split sizes and sampling parameters. The `environment.md` file specifies dataset versions and download URLs.

10. Environment and infrastructure (2.9%). Specific API endpoints, model version strings, library versions, simulator names, hardware requirements. These are often assumed to be "obvious" and omitted entirely from the paper, yet they are essential for reproduction.

Examples:

- *bbox*: "API access configured for davinci-002"; "Code to execute fine-tuning jobs through the Azure OpenAI

API"; "Mixtral-8x7B-v0.1 loaded from HuggingFace in half-precision."

- *self-composing-policies*: 15 leaves enumerate specific Gymnasium environment IDs (e.g., `ALE/SpaceInvaders-v5`, `hammer-v2`) and required packages (`Metaworld` from Farama-Foundation).
- *fre*: "The observation space's XY coordinates are discretized into 32 bins for Ant Maze agents."

Ara advantage. The `environment.md` file lists exact package versions, model identifiers, and hardware requirements. The `configs/model.md` file records model names, sizes, and loading configurations (e.g., precision, quantization).

A.1.2 Key Findings.

Three observations emerge from this analysis:

Hyperparameters are necessary but not sufficient. Classic hyperparameters—the most discussed reproduction barrier—account for only 17.2% of leaf requirements. The remaining 82.8% comprise evaluation protocols, experiment matrices, logging requirements, result interpretation targets, and implementation tricks that are harder to locate in a PDF and receive less attention in reproducibility discussions.

The combinatorial explosion is the dominant challenge. The largest category (24.1%) consists of requirements that enumerate the full cross-product of models, datasets, and configurations. In a PDF, this matrix is compressed into a single table or sentence; an agent must decompose it into individual runs. The Ara format makes this matrix explicit and machine-enumerable.

High-weight requirements demand understanding, not just extraction. All weight-2 requirements in the PaperBench rubrics belong to the "Result Interpretation" category. These test whether the agent can verify that reproduced results exhibit the *qualitative patterns* claimed by the paper—not just whether the code runs. The Ara `claims.md` and `experiments.md` layers directly encode these verification targets, making the connection between code output and paper claims explicit rather than requiring the agent to re-derive it from narrative text.

A.2 PDF Information Gap Distribution

While §A.1 characterises *what* information reproduction requires, this subsection asks the dual question: of those requirements, which are actually specified in the source PDF? The two views are complementary; together they show that the gap is not in any one category but is structural across the document.

Methodology. PaperBench's expert reproduction rubric for each paper enumerates the requirements an agent must satisfy to reproduce the paper's results. For each of the 8,921 leaf requirements across the 23 papers, we compare the requirement against the source PDF and label it *sufficient* (the PDF text fully specifies what is needed), *partial* (some components specified but key details missing), or *absent* (the PDF does not address the requirement); each label is accompanied by an annotator confidence rating (high / medium / low). When the label is partial or absent, we additionally tag the requirement with a gap-type category (missing hyperparameter, vague description, cross-reference-only, etc.). Labels are produced by an LLM-as-judge run per (requirement, PDF excerpt) pair, with the judge required to cite the PDF passage that supports its decision; the headline 45.4% sufficient figure is dominated by the 64% high-confidence subset (full pipeline in `code/eval/pdf_information_gap.py`).

Per-category coverage. Table 4 breaks the 8,921 requirements down by PaperBench task category. The median paper shows 45.3% sufficient and 47.9% partial, confirming the gap is systemic rather than driven by outliers.

Task Category	Reqs	Sufficient	Partial	Absent
Code Development	3,942	37.3%	54.9%	7.8%
Code Execution	4,355	50.5%	47.9%	1.6%
Result Analysis	624	60.6%	36.9%	2.6%
Overall	**8,921**	**45.4%**	**50.2%**	**4.4%**

Table 4. Reproduction information gap across 23 PaperBench papers (8,921 requirements). PDFs systematically under-specify the information needed for reproduction, with the largest gaps in code development and dataset acquisition.

Gap-type breakdown. Table 5 breaks down the 8,921 reproduction requirements by gap type. The three largest categories—missing hyperparameters (26.2%), vague descriptions (21.9%), and cross-reference-only specifications (13.4%)—account for over 60% of all gaps and are precisely the information types that structured formats address by design. At the fine-grained level, Dataset Acquisition achieves only 5.4% sufficient coverage (25.5% entirely absent)—no paper in the corpus consistently provides download URLs, preprocessing scripts, or data format specifications. Evaluation, Metrics & Benchmarking sits at 30.0%: papers often state *which* metrics they use but not *how* they compute them (binning strategies, confidence intervals, statistical tests).

Cross-view alignment. Reading Tables 3 and 5 together makes the failure mode concrete: the heaviest reproduction need (Combinatorial experiment matrix, 24.1%) is also among the most poorly specified (Vague description, 21.9% of gaps), and the third-largest need (Hyperparameters, 17.2%) maps almost one-to-one onto the largest gap (Missing hyperparameter, 26.2%). The two analyses agree on which categories Ara's structured layers must rescue.

Gap Type	Count	% of Gaps
Missing hyperparameter	2,558	26.2%
Vague description	2,141	21.9%
Cross-reference only	1,313	13.4%
Missing code detail	1,064	10.9%
Missing baseline detail	1,054	10.8%
Missing URL	538	5.5%
Figure only	499	5.1%
Ambiguous specification	399	4.1%
Implicit assumption	150	1.5%

Table 5. Distribution of information gap types across 8,921 requirements. The three largest categories—missing hyperparameters, vague descriptions, and cross-references—are precisely the gaps that structured formats address by design.

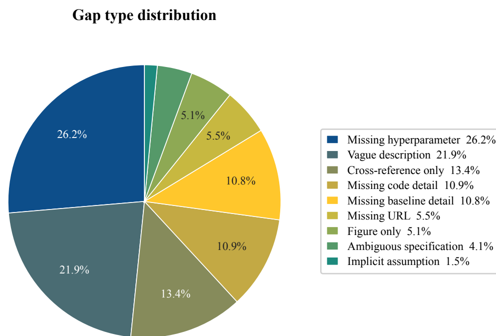


Figure 7. The three largest gap types are precisely the categories ARA's structured layers address.

A.3 Physical Layer Modes: Kernel vs Repository

The Physical Layer (`/src`) adopts one of two modes, declared in the `PAPER.md` frontmatter (`src_mode: kernel|repo`) so that consuming agents adapt their strategy immediately. These two modes cover the dominant contribution types in empirical CS, where executable code is the natural physical representation.

Kernel mode (`/src/kernel/`). When the contribution is primarily *algorithmic*, the invariant can be cleanly separated from scaffolding. The kernel contains only the core modules with typed I/O signatures—often one to two orders of magnitude smaller than the full repository—stripped of all environment-specific code. A general-purpose coding agent consumes the kernel alongside the structured specification in `/logic/solution/` and generates fresh, environment-native boilerplate in minutes. Because agent coding capabilities improve continuously, the same kernel yields a *better* surrounding implementation over time: the artifact appreciates rather than decays.

Repository mode (`/src/repo/`). When the contribution is primarily *systemic*—a CUDA kernel, a distributed training strategy, a systems architecture—the engineering *is* the contribution. The full implementation is retained but *annotated*: an `index.md` manifest maps each source file to the ARA component it implements—which claim it supports, which heuristic it embodies, which architectural module it belongs to—providing the structured navigation that a monolithic codebase lacks. Forensic bindings connect code regions to claims, constraints, and heuristics, so an agent traverses the codebase guided by research structure rather than by directory conventions alone.

In both modes, the Cognitive Layer remains the primary interface for understanding the contribution; the Physical Layer provides executable evidence, scaled to match.

A.4 ARA by Example: This Paper's Own Artifact

This paper is itself maintained as an ARA artifact. The `ara/` directory at the repository root contains the living cognitive, physical, and exploration layers that were populated incrementally during the research process (see §3.1). We reproduce excerpts from each key file below to give readers a concrete sense of the format. All entries are real; only trailing items are elided for space.

A.4.1 Directory layout and root manifest. The complete `ara/` directory for this paper is shown below. An agent's first action is to read `PAPER.md`, which contains YAML frontmatter and an abstract (~500 tokens) sufficient to decide relevance without loading any layer.

```
ara/
  PAPER.md # entry point
  logic/ # Cognitive Layer
    problem.md # observations, gaps, key insight
    claims.md # 16 falsifiable claims + status
    experiments.md # verification plan (E1-E6)
    related_work.md # typed citation dependencies
    solution/
      heuristics.md # 23 design decisions + rationale
  trace/ # Exploration Graph
    exploration_tree.yaml # 114-node decision DAG
    sessions/ # 38 session logs (2026-03-12..04-26)
      session_index.yaml # chronological index
      2026-03-12_001.yaml # ...one file per session
    pm_reasoning_log.yaml # Live PM reasoning trace
  evidence/ # Evidence Layer
    README.md # index of raw results
  staging/
    observations.yaml # 94 unpromoted observations
```

The root manifest `PAPER.md`:

```
---
title: "Agent-Native Research Artifacts"
authors: ["Anonymous", "Anonymous"]
venue: "Anonymous Venue"
status: draft
date_created: "2026-03-12"
last_updated: "2026-04-27"
abstract: >
  We propose the Agent-Native Research Artifact
  (ARA), a file-system protocol that replaces the
  narrative paper with a machine-executable research
  package organized across four interlocking layers:
  a Cognitive Layer (/logic) encoding structured
```

```
  scientific reasoning, a Physical Layer (/src)
  containing the executable code kernel, an
  Exploration Graph (/trace) preserving the full
  branching research trajectory including dead ends,
  and an Evidence Layer (/evidence) grounding every
  claim in raw empirical results. PDF publication
  imposes two structural costs on autonomous
  research: a Storytelling Tax (failed experiments
  and rejected hypotheses are discarded to fit a
  linear narrative) and an Engineering Tax (the gap
  between reviewer-sufficient prose and
  agent-sufficient specification leaves critical
  implementation details unwritten). On PaperBench
  and RE-Bench, ARA raises question-answering
  accuracy from 72.4% to 93.7% and reproduction
  success from 57.4% to 64.4%; on RE-Bench's five
  open-ended extension tasks, the failure traces
  preserved in ARA accelerate research progress by
  helping the agent avoid pitfalls prior runs
  already mapped, but for a sufficiently capable
  model the same recorded playbook can constrain a
  more creative agent that would otherwise step
  outside the prior-run box.
layers:
  logic: logic/
  src: src/
  trace: trace/
  evidence: evidence/
  staging: staging/
---

# Layer Index

- **Cognitive** (`logic/`): structured reasoning
  - `problem.md` -- observations, gaps, key insight
  - `claims.md` -- 16 falsifiable claims with status
     and proof pointers
  - `experiments.md` -- verification plan (E1-E6)
  - `related_work.md` -- typed citation dependency graph
  - `solution/heuristics.md` -- 23 design decisions
     with rationale and sensitivity
- **Exploration** (`trace/`): branching trajectory
  - `exploration_tree.yaml` -- 114-node decision DAG
  - `sessions/` -- 38 session logs (2026-03-12..04-26)
  - `pm_reasoning_log.yaml` -- Live PM reasoning trace
- **Evidence** (`evidence/`): raw empirical results
  - `README.md` -- index of all evaluation data,
     including post-paper RE-Bench extension evals
- **Staging** (`staging/`): unpromoted observations
  - `observations.yaml` -- 94 preliminary observations
     (latest: 2026-04-26 cross-model and synthesis)
```

A.4.2 Cognitive Layer: `logic/claims.md`.

Each claim carries a machine-readable status (`hypothesis`, `supported`, `testing`), falsification criteria, and proof pointers that reference the evidence layer rather than inlining results. We show two claims at different lifecycle stages.

```
# Claims

## C04: Universal Ingestor produces lossless
      transformations
- **Statement**: The LLM-based Ingestor faithfully
   transforms PDF papers into ARA format without
   information loss, achieving near-parity on
   factual Q&A between ARA and source PDF.
- **Status**: supported
- **Provenance**: ai-executed
- **Falsification criteria**: Systematic accuracy
   drop (>5%) on understanding questions.
- **Proof**: [evidence/README.md ->
   understanding_eval; 450 Qs across Cat A/B/C]
- **Dependencies**: [C03]
- **Tags**: ingestor, fidelity

## C06: Negative knowledge is the highest-value
      signal
- **Statement**: The Exploration Graph's dead-end
    documentation produces the largest accuracy gap
    in the entire evaluation -- agents with failure
    traces answer questions about failed approaches
    that narrative formats make structurally
    unanswerable.
- **Status**: supported
- **Provenance**: ai-executed
- **Falsification criteria**: Dead-end docs produce
    no measurable improvement on Cat C; gap <10pp.
- **Proof**: [evidence/README.md ->
    understanding_eval Cat C;
    evidence/README.md -> extension_eval]
- **Dependencies**: [C05]
- **Tags**: exploration-graph, negative-knowledge
... <!-- 16 claims total -->
```

A.4.3 Cognitive Layer: `logic/problem.md`.

The problem file decomposes the motivation into typed observations (empirical facts), gaps (what existing approaches miss), and a key insight that bridges them. Each entry carries evidence pointers and implication fields, so an agent can trace the full argumentative chain without reading the paper's introduction. We show one representative entry per section.

```
# Problem

## Observations
### O3: Frontier LLMs fail on research implementation
- **Statement**: Even the strongest frontier LLMs
    correctly implement fewer than 40% of novel
    research contributions when given the full paper
    and codebase, with semantic misalignment as the
    dominant failure mode.
- **Evidence**: ResearchCodeBench (Hua et al. 2025)
- **Implication**: The information encoding in PDFs
    is structurally inadequate for agent consumption.

### O4: PDF information gap is systematic
- **Statement**: Across 23 PaperBench papers (8,921
    rubric requirements), only 45.4% of reproduction
    requirements are fully specified in the PDF.
- **Evidence**: Own experiment -- info_gap_aggregate
- **Implication**: The PDF format is structurally
    incapable of serving as a self-contained
    reproduction specification.
... <!-- 4 observations total -->

## Gaps
### G2: Negative knowledge is systematically discarded
- **Statement**: Dead ends, rejected hypotheses, and
    convergence-critical tricks are lost to the
    narrative compression of the publication process.
- **Caused by**: O1
- **Why it matters**: Downstream agents waste compute
    re-exploring paths already proven fruitless.
... <!-- 2 gaps total -->

## Key Insight
- **Insight**: Separate research knowledge into four
    orthogonal layers -- structured scientific logic
    (Cognitive Layer /logic), minimal executable code
    (Physical Layer /src), preserved decision history
    (Exploration Graph /trace), and raw empirical
    results (Evidence Layer /evidence) -- to create
    a machine-executable knowledge package that
    eliminates both the Storytelling Tax and
    Engineering Tax.
- **Derived from**: O1, O2, O3, O4, G1, G2
```

A.4.4 Cognitive Layer: `logic/solution/heuristics.md`.

Each heuristic records a design decision with its rationale, provenance (who

introduced it), and sensitivity rating so that agents know which choices are safe to vary.

```
# Heuristics

## H04: Directional verification over exact matching
- **Rationale**: Legacy papers routinely omit details
    needed for exact reproduction. Verifying
    directional properties (A > B on metric X)
    demonstrates the code kernel captures the core
    algorithmic insight without requiring exact
    numerical matches.
- **Provenance**: user
- **Sensitivity**: medium
- **Code ref**: [paper/sections/protocol.tex]

## H12: Minimal kernel = algorithm notes with inline
      snippets, not raw code files
- **Rationale**: Full code dumps (200-700 lines)
    cause context dilution -- the agent spends
    tokens parsing boilerplate already described in
    official_solution_notes.md. Notes contain core
    algorithm with key code snippets inline,
    sufficient for comprehension while 5-10x smaller.
- **Provenance**: user-revised
- **Sensitivity**: high
- **Code ref**: [code/artifacts/rebench-*/src/
    kernel/official_solution_notes.md]
... <!-- 18 heuristics total -->
```

A.4.5 Exploration Graph: `trace/exploration_tree.yaml`.

The exploration tree preserves decisions, dead ends, and experiments as a traversable graph. We show one node of each type from this paper's 94-node tree.

```
tree:
  - id: N04
    type: decision
    title: "Tripartite layer architecture"
    provenance: user
    timestamp: "2026-03-09"
    choice: >
      Three orthogonal layers: Cognitive (/logic),
      Physical (/src), Exploration Graph (/trace).
      Each addresses a distinct tax.
    alternatives:
      - "Two-layer (logic + code only)"
      - "Four-layer (separate evidence at top)"
      - "Flat structured document (single file)"
    evidence: >
      Three-layer separation achieves minimal
      representation while preserving all three
      dimensions that PDFs conflate.

  - id: N50
    type: dead_end
    title: "Trimming src/ alone does not recover
          Cat C from enrichment regression"
    provenance: ai-suggested
    timestamp: "2026-03-14"
    hypothesis: >
      Removing boilerplate code from src/ would
      reduce context dilution enough for Cat C
      (failure knowledge) to recover to ~80%.
    failure_mode: >
      Cat C stayed at 57.5% despite reducing src/
      from 56-104K to 20-36K per artifact. Remaining
      enrichment additions still dilute
      trace/exploration_tree.yaml content.
    lesson: >
      Context dilution for Cat C is more sensitive
      than expected. Even ~200 lines of structured
      markdown in src/ can push failure knowledge
      below the retrieval threshold.

  - id: N17
    type: experiment
    title: "24,008-run exploration waste analysis"
    provenance: ai-executed
    timestamp: "2026-03-12"
    result: >
      Analyzed 24,008 runs across 21 models,
      228 tasks. 59.2% of tokens wasted on
      dead-end exploration. 90.2% of cost goes
      to failed runs. Failed runs consume 113x
      more tokens than successful ones (median).
    evidence: [C13, C14, "code/eval/malt_analysis/
      exploration_tax_findings.json"]
... <!-- 94 nodes total -->
```

A.4.6 Exploration Graph: `trace/sessions/`.

Each research session is logged as a structured YAML record capturing events, AI actions, files changed, claims affected, and open threads. Below is one session (abridged); the `session_index.yaml` file provides a one-line chronological summary of all 36 sessions.

```
# trace/sessions/2026-03-19_001.yaml
session:
  id: "2026-03-19_001"
  timestamp: "2026-03-19T02:00"
  summary: "BAM reproduction pilot: ARA 88.2% vs
    baseline 93.2% -- first paper where baseline leads"

events_logged:
  - type: experiment
    id: N65
    summary: "BAM ARA mega-task: 220.5/250 (88.2%
      weighted), 10/10 subtasks, 6.9h, 12.6M tokens"
  - type: experiment
    id: N66
    summary: "BAM baseline mega-task: 234/251 (93.2%
      weighted), 10/10 subtasks, 4.3h, 8.6M tokens"
  - type: observation
    id: O50
    summary: "ARA won T6 (non-Gaussian) 10/10 vs
      8.5/10 -- clearer hyperparameter specs"

ai_actions:
  - action: "Ran BAM ARA reproduction (10 subtasks)"
    files_changed:
      - "code/eval/reproduction/results/bam/"

claims_touched: [C05]

open_threads:
  - "Hyperparams buried in evidence/ not src/configs/
     -- move them and rerun (becomes N67)"
```

```
# trace/sessions/session_index.yaml (excerpt)
sessions:
  - id: "2026-03-12_004"
    summary: "Full 23-paper info gap analysis -- 8,921
      reqs, median 45.3% sufficient"
  - id: "2026-03-17_001"
    summary: "Pilot reproduction: ARA 74% vs baseline
      ~14%, 5x advantage on neural-score-estimation"
  - id: "2026-03-22_001"
    summary: "Self-expansion v2 (real code): 10/10
      subtasks -- v1 pseudocode got 1/10"
  - id: "2026-03-26_001"
    summary: "small_scaling_law extension v2: ARA 0.644
      vs baseline 0.806 -- baseline wins on loss
      calibration despite ARA finding near-optimal
      hyperparams"
... <!-- 36 sessions total -->
```

The complete artifact, including all 16 claims, 18 heuristics, 94 exploration nodes, and 36 session records, is available in the supplementary material at `ara/`.

B ARA Compiler Details

The ARA Compiler (§3.2) is implemented as an *agent skill*: a self-contained, natural-language specification that, when loaded into any general-purpose coding agent's context, turns it into a domain-specialized compilation system. The skill prescribes *what* the agent should do and *what domain knowledge* it needs, but delegates all execution mechanics (model selection, tool dispatch, context management) to the host agent. This appendix reproduces the key elements of the skill specification; the complete definition is available in the supplementary code.[2]

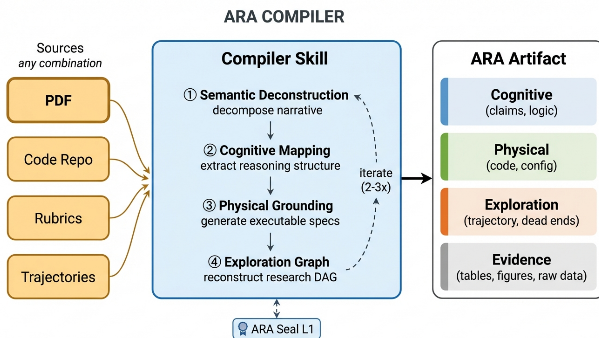


Figure 8. The ARA Compiler accepts any combination of research sources and guides a coding agent through four stages of top-down artifact compilation, iterating 2–3× with in-loop ARA Seal Level 1 validation until the output conforms to the protocol.

B.1 Design Rationale

Four invariants guide the Compiler's behaviour: what it must preserve, where it routes auxiliary sources, what it must reconstruct beyond plain extraction, and how quality is enforced.

High-fidelity preservation as a normative guarantee. A narrative paper compresses and selects; the Compiler decompresses and restores. Every numerical result, hyperparameter, architectural detail, and negative finding in the sources must appear *somewhere* in the artifact, and any PDF-accessible information missing from the ARA constitutes a compilation failure (evaluated in §4.1). Preservation is faithful to sources; *enrichment* (collective inference) separately surfaces cross-artifact patterns no single source expresses, distinguishing stated from inferred knowledge with provenance tags.

Auxiliary source routing. Beyond the PDF, research knowledge is distributed across complementary sources, each populating a distinct layer: GitHub repositories encode implementation decisions that prose omits and replace stubs with verified implementations in `/src`; expert-curated evaluation rubrics encode what domain practitioners consider the core claims and anchor `/logic` with claim decompositions; and recorded experimental trajectories preserve failure modes that papers systematically suppress and seed `/trace` with dead-end nodes the PDF omits. The Compiler routes each auxiliary source to the layer it most directly populates.

Knowledge lineage vs flat extraction. Plain-text extraction recovers content but not the cross-layer connections that lineage requires: parsing a PDF into Markdown populates four directories yet leaves them structurally isolated. The Compiler instead performs *forensic reconstruction* from sources where lineage exists only implicitly—scattered across prose, figure captions, appendix tables, and code comments—so an agent can trace any claim downstream to code or any number upstream to its hypothesis. Recovering this lineage, not populating layers, is the core compilation problem.

Quality enforcement: in-loop vs downstream. Quality is enforced in two stages. *During* compilation, the Compiler uses only ARA Seal Level 1 as an in-loop validation signal: schema conformance, cross-layer reference resolution, and required-field completeness, with failures returned as structured diagnostics that drive targeted fixes within the same agent conversation. *After* compilation, the finished artifact enters the downstream Seal pipeline (Appendix I), where Levels 2–3 evaluate its argumentative support and empirical reproducibility under the conditions a venue would impose.

B.2 Skill Specification

The Compiler skill specification (~482 lines of natural language) is structured into five sections. When loaded into a host agent's context, it provides the full domain knowledge needed to produce a schema-conforming ARA. We reproduce representative elements below; the complete specification is available in the supplementary code.

Section 1: Workflow (lines 1–11). Defines the high-level pipeline: analyze ⟶ generate ⟶ validate ⟶ fix ⟶ iterate.

Section 2: Capability usage guidelines (lines 13–26). Specifies usage conventions for standard file operations: prefer `edit_file` over `write_file` for targeted fixes, prefer `write_file` for YAML files to avoid whitespace corruption. Instructs the agent to batch work: generate all files first, then validate once.

Section 3: ARA directory schema (lines 28–414). Defines the complete directory structure and field-level requirements for every file. This is the normative schema specification that the agent must follow. Key constraints include:

[2]Full implementation: https://github.com/AmberLJC/Agent-Native-Research-Artifact.

- **PAPER.md**: YAML frontmatter with title, authors, year, venue, DOI, domain, keywords, claims summary, and abstract. Body must include a Layer Index table listing every file with a one-line description.
- **claims.md**: Each claim requires Statement, Status, Falsification criteria, Proof (referencing experiment IDs, not file paths), Dependencies, and Tags.
- **experiments.md**: Declarative verification plans (setup, procedure, metrics, directional expected outcomes). Exact numerical results are *prohibited*—they belong exclusively in `/evidence/` to enable blind reproduction.
- **heuristics.md**: Each heuristic requires Rationale, Sensitivity (low/medium/high), Bounds, Code ref, and Source.
- **exploration_tree.yaml**: Nested YAML tree with typed nodes (`question`, `experiment`, `dead_end`, `decision`, `pivot`). Minimum 8 nodes with at least one `dead_end` and one `decision`. Dead ends must document hypothesis, failure mode, and lesson.
- **evidence/**: Every results table and quantitative figure must be reproduced with exact cell values—no rounding, no omission.

Section 4: 4-stage reasoning protocol (lines 416–454). The prompt mandates a structured thinking process before file generation:

```
# Your 4-Stage Reasoning Process

You MUST follow these 4 stages in order. Produce a
<thinking> block first with your reasoning for each
stage, then produce the files.

## Stage 1: Semantic Deconstruction
Strip the narrative "Storytelling Tax." Isolate:
- The core observations and gaps that motivate the work
- Mathematical formulations and equations
- Architectural specifications and component descriptions
- Experimental configurations (hyperparameters, hardware,
  datasets)
- Numerical results and benchmarks
- Citation dependencies and their roles
- Negative results and ablation findings

## Stage 2: Cognitive Mapping
Map deconstructed content to /logic:
- Extract motivation: observations (with numbers), gaps,
  the key insight, and assumptions
- Identify falsifiable claims (not opinions or vague
  statements)
- Define formal concepts with precise notation
- Populate solution/ (architecture, algorithm,
  constraints, heuristics)
- Construct typed dependency graph for related_work.md
- Ensure every claim has falsification criteria and
  proof pointers to experiment IDs
- Design declarative experiment plans: for each major
  claim, specify how an agent would verify it

## Stage 3: Physical Stubbing
Generate /src:
- Extract exact hyperparameter values into configs/
- Write code stubs with correct function signatures
  and types
- Specify environment (dependencies, hardware, seeds)
- Code should implement the NOVEL contribution,
  not boilerplate

## Stage 4: Exploration Graph Extraction
Reconstruct the research DAG as a nested YAML tree
for /trace:
- Identify the central research question(s) as root
  nodes
- Map experiments and their outcomes as child nodes
- Document dead ends from ablations and rejected
  alternatives as leaf nodes
- Record key design decisions with alternatives
  considered
```

Section 5: Output format and rules (lines 456–482). Specifies the XML-delimited output format for `batch_write_files`, lists all 15 mandatory files, and enforces nine invariant rules (e.g., "all numerical values must be EXACT as stated in the paper," "never hallucinate claims, results, or heuristics not in the paper").

C ARA Live Research Manager Details

The Live Research Manager (§3.1) is the second agent skill in the ARA system. Like the Compiler, it is a self-contained natural-language specification that turns a general-purpose coding agent into a domain-specialized system; unlike the Compiler, it operates continuously alongside the researcher rather than as a one-shot compilation. This appendix expands on the design principles, cross-session mechanisms, and submission workflow summarized in §3.1.

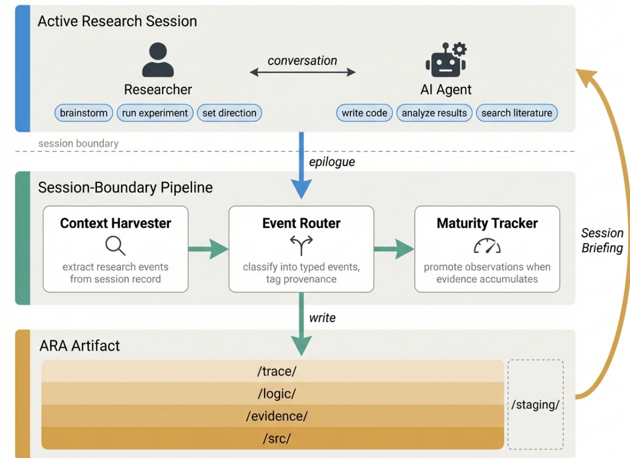


Figure 9. The Live Research Manager operates at session boundaries: a three-stage pipeline (Context Harvester → Event Router → Maturity Tracker) distills each researcher–agent conversation into typed events that accumulate across ARA layers over time.

Event taxonomy. The Event Router classifies each extracted event into one of seven types (Table 6) and tags it with provenance. Event payloads follow the protocol's factual-density requirement (§2.1): conversational prose is distilled into telegraphic, quantitative language before committing to the artifact. Trace events (decisions, experiments, dead ends, pivots) append to the Exploration Graph (`/trace/`);

claims and heuristics enter the Cognitive Layer (`/logic/`); and events that resist classification are staged in `/staging/` for later promotion via the closure-driven mechanism described below.

C.1 Design Principle Rationale

The three design principles underlying the Live Research Manager (§3.1) are expanded below with full motivation.

P1. Silent, framework-independent integration. Documentation has traditionally been a *retrospective* activity: a context switch that introduces both friction and information loss. The system must integrate with any general-purpose coding agent (Claude Code, Cursor, Windsurf, or future frameworks) without custom SDKs, API bindings, or infrastructure changes. A natural-language specification that the agent reads into its context is the most portable interface: it requires nothing beyond the tool access agents already have, and artifact quality improves automatically as language models advance. The manager runs as a background process that silently collects research traces, constructing the artifact without interrupting active work or injecting prompts into an ongoing research conversation.

P2. Faithful epistemic provenance. AI-native research blurs the boundary between human insight and machine execution. The manager must objectively track *who did what*: distinguishing ideas explicitly stated by the researcher, suggestions inferred by the agent, actions the agent executed autonomously, and AI suggestions the researcher revised. Without such provenance, an artifact cannot faithfully represent the epistemic origin of its contents. A behavioral consequence: an `ai-suggested` event never auto-upgrades to a confirmed claim or decision until the researcher explicitly endorses it, preserving the epistemic integrity of the artifact even when the agent generates fluent but unconfirmed reasoning chains. The research process is inherently chaotic: a single session may interleave hypothesis formation, coding, debugging, and writing with no clear boundaries. The manager must translate this raw conversational stream into the structured Ara schema without losing information or imposing premature structure. Observations that are not yet classifiable should be staged rather than forced into categories, and knowledge should mature progressively, from hunches to typed events to formally bound claims, mirroring how research understanding actually develops.

P3. Comprehensive trajectory capture. Research is nonlinear and stochastic: hypotheses branch, experiments fail, directions are abandoned and revisited. The manager must capture this full trajectory, not just the successes that survive into a polished paper, but the dead ends, pivots, and intermediate observations that constitute the actual research process. Cross-layer bindings between claims, evidence, code, and decisions must be established at capture time while the conversational context is still available; post-hoc reconstruction from archived transcripts loses these causal chains.

C.2 Two-Phase Crystallization and Pivot Propagation

Artifact construction proceeds at two timescales (**P2**). *Continuously*, at every session boundary, the manager appends trace events to the Exploration Graph, recording how the researcher navigates the open-ended research landscape. *Periodically*, when a major milestone is reached—a hypothesis confirmed or refuted, a working prototype completed, a key design choice finalized—the Maturity Tracker *crystallizes* the accumulated observations into the more structured layers of the artifact: raw observations mature into formal claims in the Cognitive Layer (`/logic/`), working code is documented in the Physical Layer (`/src/`), and concepts are added to the knowledge index. This two-phase rhythm mirrors how research understanding actually develops: insights begin as scattered observations, and forcing premature structure would distort the record. When a pivot invalidates an earlier design choice, the manager propagates the change: it updates the affected artifact entries (claims, heuristics, configuration) to reflect the new direction while the Exploration Graph retains the original rationale and the reason for revision, so the project's intellectual history is preserved alongside its current state. At the start of each next session, the manager reads the artifact back and surfaces a structured briefing of open threads, recently crystallized claims, and unresolved contradictions, closing the loop with the researcher.

C.3 Closure-Driven Crystallization

Principle P2 requires that observations mature "as evidence accumulates," but *evidence accumulation* needs an operational definition. A counter-based threshold (promote after N references) is arbitrary; asking an LM in isolation "is this mature?" lacks grounding. We instead define maturity through *closure signals*: externally observable patterns in the researcher–agent conversation that indicate the researcher has treated an observation as settled. At each session boundary, the Maturity Tracker inspects the session record and promotes a staged observation when at least one closure signal is present.

Closure signal taxonomy.

Topic abandonment. The researcher has moved to a new topic without revisiting the observation in the subsequent k turns (default $k = 5$), and no pending question remains open on the original thread.

Event Type	Structured Payload	Event Type	Structured Payload
`decision`	Choice, alternatives, evidence	`claim`	Statement, falsification criteria
`experiment`	Metrics, claim linkage	`heuristic`	Trick, sensitivity, bounds
`dead_end`	Hypothesis, failure mode, lesson	`observation`	Raw finding, awaiting classification
`pivot`	Trigger, rationale		

Table 6. Research event types and structured payloads emitted by the Event Router.

Verbal affirmation. The researcher explicitly endorses the observation ("yes, we'll go with X," "confirmed," or equivalent paraphrase), making the adoption decision first-person.

Empirical resolution. An experiment bound to the observation has produced a result and the researcher has commented on it; both supported and refuted outcomes are valid terminations (the refuted case promotes to a `dead_end`).

Artifact commitment. A downstream artifact now depends on the observation: code is merged, a config value is fixed, a subsequent claim uses it as a premise, or a design decision is documented as following from it.

Contradiction trigger. When a new observation contradicts one already staged or crystallized, the Maturity Tracker does not silently overwrite. Both entries are flagged, the contradiction is appended to the Exploration Graph as a `decision` node with unresolved status, and resolution is deferred to the next briefing where the researcher adjudicates explicitly.

C.4 Cross-Session Continuity

A stateless coding agent has no memory of previous conversations. If the manager simply reads the artifact at each session boundary, it knows *what* the artifact contains but not *why* it is organized the way it is—which classification choices were non-obvious, which observations were deliberately deferred, which patterns guided past merges and promotions. Without this self-awareness, the manager risks inconsistent classification, duplicate entries, and organizational drift across sessions.

We address this with two lightweight mechanisms. First, a *reasoning log* (`trace/pm_reasoning_log.yaml`) records the manager's own organizational decisions and their rationale at each session boundary—a compressed account of a few lines per session that gives the manager self-continuity without requiring access to raw conversation transcripts. Second, each session record includes a *key context* field: compressed summaries of the most important human–agent exchanges, preserving conversational nuance that would otherwise be lost when the raw conversation is no longer available. Together, these mechanisms ensure that the manager can read back not only the artifact's current state but also the reasoning chain that produced it, maintaining coherent organization across arbitrarily many sessions at negligible token cost.

D Review System Details

The Review System is the third Ara skill alongside the Compiler (App. B) and the Live Research Manager (App. C). This appendix specifies the full design: the two design principles (§D.1), the three-level ARA Seal credential and its figure (§D.2), the per-level checker implementation and failure taxonomy (§D.3), and the three-stage review pipeline that consumes those levels (§D.4). Empirical evaluation of the Seal (compiler convergence, mutation benchmark, execution reproducibility) is reported separately in App. I.

D.1 Design Principles

We derive two design principles below; cross-artifact review via agent-to-agent Ara comparison, which becomes meaningful only as the corpus reaches critical mass, is discussed in §M.

P1. Automate the mechanical; reserve humans for judgment. Structural validity, internal consistency, and claim reproducibility are objective properties that either pass or fail, whereas significance, novelty, and taste require domain expertise. The review system should never ask a human to verify that Experiment E03 matches Claim C02 when a machine can do so in seconds; resolving all machine-checkable issues *before* the artifact reaches a human reviewer ensures that expert attention is spent exclusively on questions that genuinely require it.

P2. Reproducibility as a foundational requirement. "Code available upon request" nominally satisfies today's reproducibility bar; in an Ara-native system, reproducibility is a *machine-verified property* of the artifact itself. Passing ARA Seal Level 1 (structural integrity) is a submission requirement, and Level 2 (argumentative rigor) produces a structured critique before the venue spends compute on Level 3 (execution reproducibility), whose results are then attached to every review. Artifacts that fail structural checks, or whose claims remain obviously under-supported after Level 2 critique, do not advance to human review.

D.2 The ARA Seal: Three-Level Verification Credential

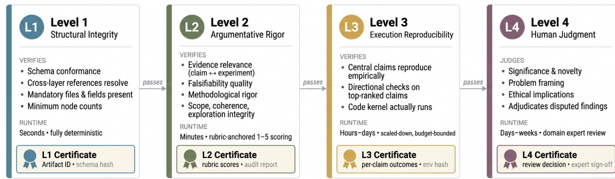


Figure 10. The ARA Seal is a three-level verification credential. Each level tests a progressively stronger property of the artifact, escalating in cost and breadth: structural integrity (seconds, deterministic), argumentative rigor (minutes, rubric-anchored agent), and execution reproducibility (hours to days, sandboxed coding agent). Passing the applicable levels issues a Seal Certificate that downstream agents check before investing compute.

A PDF paper earns trust indirectly: venue prestige, citation counts, and author reputation serve as proxies for quality, but none verify the work itself. Because an Ara encodes research as typed, machine-traversable data with explicit claim–evidence bindings, its quality becomes *directly verifiable*. The ARA Seal operationalizes this as a three-level verification protocol (Figure 10), where each level tests a progressively stronger property; per-level implementation and the failure taxonomy follow in §D.3.

Level 1 – Structural Integrity. Verifies that the artifact is well-formed and internally consistent: the directory ontology exists, all structured files conform to their schema (each claim carries `Statement`, `Status`, `Falsification criteria`, and `Proof`; each heuristic carries `Rationale`, `Sensitivity`, and `Bounds`), and all cross-layer references resolve (experiment IDs in `claims.md` point to valid entries in `experiments.md`, code references trace to implementations in `/src`). The check is deterministic and runs in seconds.

Level 2 – Argumentative Rigor. Without executing any code or consulting external sources, a *Rigor Auditor* agent evaluates whether the content of a Level-1-valid artifact is epistemically sound along six objective dimensions, each scored on an anchored 1–5 rubric. The three load-bearing dimensions are *evidence relevance* (every claim's cited experiments substantively address what the claim asserts under type-aware entailment: causal claims require isolating ablations, generalization claims require heterogeneous test conditions, improvement claims require baseline comparisons), *falsifiability quality* (criteria are actionable, non-tautological, scope-matched, and independently testable without proprietary data), and *methodological rigor* (baseline adequacy, ablation coverage, statistical reporting, metric–claim alignment). Three further dimensions—scope calibration, argument coherence, and exploration integrity—are defined in App. I.2. Findings are tagged by severity (*critical*, *major*, *minor*, *suggestion*) with verbatim evidence spans and actionable suggestions; the overall grade is derived from the mean score and per-dimension floors. Every check reduces to a rubric-anchored property of the artifact's content, so Level 2 remains objective; significance, novelty, and taste are reserved for human reviewers in Stage 3.

Level 3 – Execution Reproducibility. Verifies that the artifact's central claims reproduce empirically. The system selects claims by criticality—those in the contribution list, those anchoring the most downstream dependencies, or those flagged by the authors—and runs scaled-down directional checks (small data, few epochs, toy configurations) that test whether claimed properties hold qualitatively rather than reproducing exact numbers. The verifying agent is isolated from the artifact's evidence layer: it receives only the code kernel and algorithm descriptions, never the reported numbers, preventing fabrication by copying expected outcomes. Venues set a compute budget; claims that exceed it are flagged as *unverified*. Full-scale reproduction (original datasets, full training runs, exact metric recovery) is optional and typically post-acceptance or community-driven; results are appended to the living Seal Certificate.

Passing the applicable levels issues a **Seal Certificate**—a signed record of artifact ID, verification level achieved, timestamp, environment hash, and per-claim reproduction outcomes. Downstream agents check the certificate before investing compute, avoiding redundant re-verification.

D.3 Seal Implementation Details

Per-level checkers. Each verification level is implemented as an automated checker:

- **Level 1 (Structural Integrity)**: A Python script verifies (a) the existence of mandatory directories (`/logic`, `/src`, `/trace`, `/evidence`), (b) the presence of all mandatory files including `PAPER.md` with valid YAML frontmatter, `problem.md`, `claims.md`, `experiments.md`, and all `solution/` files, (c) schema conformance of every structured file (e.g., each claim must have `Statement`, `Status`, `Falsification criteria`, and `Proof`; each experiment must have `Verifies`, `Setup`, `Procedure`, and `Expected outcome`; each heuristic must have `Rationale`, `Sensitivity`, and `Bounds`), (d) minimum counts ($\geq$ 5 concepts, $\geq$ 3 experiments, $\geq$ 8 exploration tree nodes with at least one `[dead_end]` and one `[decision]`), and (e) cross-layer reference resolution: every experiment ID referenced in `claims.md` `Proof` fields resolves to an entry in `experiments.md`; every claim ID referenced in `experiments.md` `Verifies` fields resolves to an entry in `claims.md`; every `code_ref` in `heuristics.md`

points to a valid module in `/src`; components declared in `architecture.md` have corresponding code stubs; claim references in `exploration_tree.yaml` resolve to valid claim IDs.
- **Level 2 (Argumentative Rigor)**: Without executing code or consulting external sources, the Rigor Auditor evaluates the artifact's content on six objective rubric-anchored dimensions (evidence relevance, falsifiability quality, scope calibration, argument coherence, exploration integrity, methodological rigor), each scored on a 1 to 5 scale. The output is a rigor report keyed to specific Ara components, with severity-ranked findings, verbatim evidence spans, and an overall grade derived from the mean score and per-dimension floors.
- **Level 3 (Execution Reproducibility)**: A coding agent reads the Ara and attempts to reproduce claims using the code kernel. LLM-generated test cases verify directional properties of the paper's claims. This is the same protocol used in the reproduction evaluation (§4.2).

Failure taxonomy. For each Seal level failure, we record the specific check that failed and classify it into one of the following categories: *missing file*, *missing field*, *dangling reference*, *type mismatch*, *dependency resolution failure*, *execution error*, or *nondeterminism*.

D.4 Three-Stage Review Pipeline

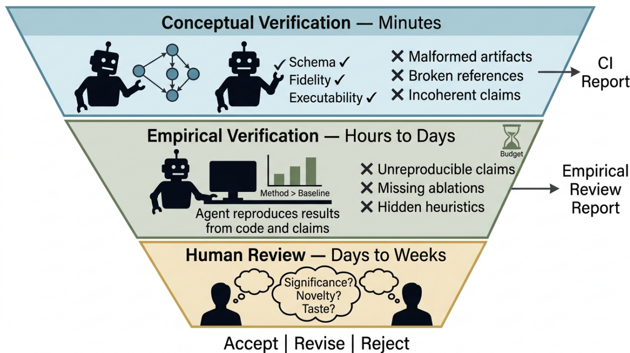


Figure 11. Three-stage Ara-native review pipeline. Stages 1–2 invoke the ARA Seal levels of Figure 10 to resolve mechanical and rigor issues before human reviewers engage, redirecting expert attention to novelty and significance.

The Seal levels feed a three-stage review pipeline (Figure 11) that mirrors CI/CD practices, where each stage gates the next: conceptual verification, empirical verification, and human judgment.

Stage 1: Conceptual Verification (Minutes). Checks whether the artifact is well-formed and conceptually well-supported without running any experiments. Automated checks validate structural integrity (Seal Level 1: schema conformance, cross-layer reference resolution, required-field completeness), and the Rigor Auditor executes Level 2 by assessing whether major claims are appropriately scoped, whether attached evidence and ablations justify them, and whether obvious baselines, assumptions, or limitations are missing. Level 1 is mechanical and deterministic; Level 2 produces a structured rigor report keyed to specific claims, experiments, and omissions. Stage 1 also generates *advisory diagnostics* for human reviewers in Stage 3: whether the exploration tree contains genuine dead-end nodes (suggesting authentic process documentation versus a sanitized linear chain), whether the related-work graph covers the obvious baselines for the claimed contribution, and whether experiment–claim coverage has gaps. These diagnostics inform but do not gate; only the Seal checks themselves are pass/fail requirements. The output is a **CI Report** plus a Level 2 rigor report, both attached to the submission and visible to authors and reviewers; authors iterate on structural failures and rigor critiques before the artifact advances, analogous to fixing lint errors and design issues before code review.

Stage 2: Empirical Verification (Hours–Days). Once the artifact passes conceptual review, Stage 2 tests whether the claimed results actually reproduce. An AI reviewer agent executes Seal Level 3: it ranks claims by criticality (contribution-list claims and those with the most downstream dependents in the claim graph) and runs scaled-down directional checks within a venue-specified compute budget. The agent is isolated from the artifact's evidence layer and ground-truth numbers, preventing fabrication by copying expected outcomes. For each tested claim, the Empirical Review Report records whether the claimed direction holds (e.g., "method A outperforms baseline B on metric M"), wall-clock time consumed, and any divergence from the artifact's stated expected outcomes; claims that exceed the remaining budget are flagged as *unverified* with an estimate of the compute required. Beyond reproduction, the Stage-2 agent assesses experimental comprehensiveness: are ablations present for each design choice the claims attribute non-trivial impact to; do the experimental conditions cover the claimed generality, or are results cherry-picked from favorable settings; are there undocumented heuristics in the code that do not appear in the artifact's cognitive layer.

Stage 3: Human Review (Days–Weeks). Human reviewers receive the submission alongside the CI Report and Empirical Review Report; their role shifts from *verification*—already handled by Stages 1–2—to *judgment*. They focus on the questions no machine can answer: is this contribution significant; is the core insight genuinely novel or an incremental recombination; is this the right formulation of the problem; what are the ethical implications and potential for misuse. Where the AI reviewer's findings are contested by the authors, humans adjudicate with the full audit trail (CI Report, rigor

report, Empirical Review Report, and per-finding evidence spans) available for inspection. Human reviewers write their reviews in the same typed format used by the automated stages—findings linked to specific Ara components (claim IDs, experiment IDs, exploration-tree nodes)—so all feedback is actionable and traceable, and downstream agents can read the review history of any artifact as structured data rather than free prose.

E Evaluation Setup

This appendix defines the shared evaluation infrastructure used by the four evaluation appendices that follow: the test corpus (App. E.1) consumed by the Understanding (App. F), Reproduction (App. G), and Review System (App. I) experiments, and the shared agent and judge configurations (App. E.2). Each subsequent evaluation appendix specifies only its own additional setup (task design, scoring formula, harness). The RE-Bench tasks used by the Extension experiment (App. H) are documented separately in App. H.1.

E.1 Test Corpus

Selection criteria. We draw our evaluation corpus from PaperBench [54], which provides expert-authored hierarchical reproduction rubrics for ICML 2024 papers. We adopt all 23 papers in PaperBench's public release as our PaperBench corpus, with no exclusions: every paper satisfies our three required properties: (1) peer-reviewed at a top ML venue (ICML 2024 Spotlight/Oral, with two NeurIPS 2024 workshop development papers) with a publicly available PDF; (2) spanning diverse ML subfields to test breadth across the discipline; (3) accompanied by a PaperBench rubric with fine-grained leaf requirements that enables quantitative evaluation of reproduction fidelity. We supplement this corpus with 7 open-ended R&D tasks from RE-Bench [61], yielding 30 evaluation targets with 450 questions total. Table 1 (main text) summarises the two benchmarks side by side.

Paper list. Table 7 lists the 23 PaperBench papers used across the understanding and reproduction evaluations. Note that PaperBench's own protocol blacklists author code; we relax this for the baseline so it has the strongest possible footing (PDF + companion GitHub), making "repo availability" a property of our harness rather than of PaperBench. Of the 23 papers, 15 are included in the reproduction experiment; the remaining 8 are excluded because faithful end-to-end reproduction exceeds our per-task compute budget or requires specialized infrastructure outside our evaluation harness (e.g., multi-day CLIP adversarial fine-tuning, Isaac Gym RL, full ImageNet coreset sweeps, large multi-model benchmark suites). These 8 papers participate only in the understanding evaluation.

Corpus diversity. The 23 papers span diverse ML subfields: efficiency, alignment, interpretability, RL, scientific ML, generative models, optimization, retrieval, evaluation, and adaptation. While all papers come from a single venue (ICML 2024), they vary substantially in methodological complexity—from systems-oriented efficiency papers with minimal formal analysis to theory-heavy contributions (PINN, stochastic interpolants)—heuristic density, paper length, and contribution type (new architectures, training recipes, algorithms, analysis frameworks). The corpus includes papers that are deliberately challenging for structured extraction: those whose contributions are analysis rather than methods (mechanistic-understanding), multi-component pipelines (all-in-one), and papers with complex combinatorial experiment matrices (PINN with 1,273 leaf requirements).

Short Name	Domain	Repro
adaptive-pruning	Efficiency	Yes
all-in-one	Multi-task	Yes
bam	LLM alignment	Yes
bbox	Black-box LLM	Yes
bridging-data-gaps	Data	No
fre	RL	Yes
ftrl	Online learning	Yes
lbcs	Calibration	No
lca-on-the-line	Prediction	No
mechanistic-understanding	Interpretability	Yes
pinn	Scientific ML	Yes
rice	Retrieval	Yes
robust-clip	Robustness	No
sample-specific-masks	Data augmentation	Yes
sapg	Optimization	No
self-composing-policies	Continual RL	Yes
self-expansion	Self-training	Yes
semantic-self-consistency	Evaluation	No
seq.-neural-score-est.	Score estimation	Yes
stay-on-topic-w/-CFG	Generation	No
stochastic-interpolants	Generative	Yes
test-time-model-adapt.	Adaptation	Yes
what-will-my-model-forget	Continual learning	No
Total: 23		15

Table 7. Test corpus: 23 PaperBench papers from ICML 2024 spanning diverse ML subfields. The "Repro" column indicates inclusion in the reproduction experiment (requires companion code). All 23 papers participate in the understanding evaluation.

E.2 Shared Agent and Judge Configuration

The Understanding (App. F), Reproduction (App. G), and Review System (App. I) experiments share an evaluation agent and judge configuration; the Extension experiment (App. H) uses a custom SLURM-launched harness documented in App. H.4.

Evaluation agent. For each (target, format, query) triple, we instantiate a fresh sub-agent (Claude Sonnet 4.6). The agent receives the format under test (the full ARA directory for the ARA condition, or the PDF plus companion GitHub repository for the baseline) and a single query. Each query is answered independently with a fresh context to prevent information leakage across queries.

Judge model and grading scale. Outputs are scored on a ternary scale by an independent judge (Claude Opus 4.6) against a gold reference:

- **Correct (1.0)**: The answer matches the ground truth in substance. Minor phrasing differences are acceptable; numerical values must be exact.
- **Partial (0.5)**: The answer conveys the main insight but misses key sub-details.
- **Incorrect (0.0)**: The answer contains a factual error, contradicts the gold answer, or hallucinates an answer to an unanswerable question.

Each evaluation appendix specifies how this scale composes into its experiment-specific aggregate metric (e.g., per-category accuracy in App. F, difficulty-weighted success rate in App. G).

F Understanding Evaluation

This section reports the methodology and per-stratum results for the Understanding experiment (§4.1). The shared test corpus is in App. E.1.

F.1 Question Bank and Grading Rubric

Question generation. For each of the 30 evaluation targets (23 PaperBench papers, 7 RE-Bench tasks), we generate 15 questions across three categories: 10 Category A (fidelity: information preservation), 5 Category B (configuration and detail recovery) for PaperBench papers, or 5 Category C (failure and exploration knowledge) for RE-Bench tasks. Questions are designed to require specific, unambiguous answers or verifiable outputs. We avoid opinion questions ("Is this architecture good?") and questions requiring external knowledge not in the paper ("How does this compare to BERT?").

Category A question templates (10 per paper).

- **Architecture & Method** (3 questions): "What is the [specific structural detail]?" (e.g., "How many layers does the encoder have?", "What are the inputs and outputs of the multi-head attention module?")
- **Hyperparameters & Configuration** (2 questions): "What [training/optimization detail] is used?" (e.g., "What optimizer is used and what are its hyperparameters?", "What is the batch size in tokens?")
- **Results & Claims** (3 questions): "What [metric] does the [model variant] achieve on [benchmark]?" (e.g., "What BLEU score does the base model achieve on WMT 2014 EN-DE?")
- **Rationale & Design Decisions** (2 questions): "Why was [design choice] made instead of [alternative]?" (e.g., "Why is scaled dot-product attention used instead of additive attention?")

Category B question templates (5 per PaperBench paper).

- **Implementation** (2 questions): "Implement [specific module] with the correct [dimensions/activations/structure]."
- **Configuration Recovery** (2 questions): "Write the [optimizer/training/data] configuration with the exact parameters specified."
- **Debugging & Troubleshooting** (1 question): "[Failure scenario]—identify the cause and fix it."

Category C question templates (5 per RE-Bench task).

- **Dead-End Knowledge** (3 questions): "What approaches have been tried and failed?" / "What is the documented failure mode of [approach]?"
- **Exploration History** (2 questions): "What alternatives were considered for [decision]?" / "What lesson was learned from the [dead-end] attempt?"

The shared evaluation agent (Sonnet 4.6) and judge (Opus 4.6) configuration is in App. E.2.

F.2 Exploration Cost Detailed Breakdown

What this measures (and what it is not). Across 24,008 agent runs (21 frontier models, 228 tasks) in the METR MALT corpus, 59.2% of tokens and 90.2% of dollar cost ($63,483 total) are spent in runs that did not reach the task's reference score. This is not wasted research effort: those runs map dead ends, rule out alternatives, and narrow the strategy space the next agent should consider. The cost only *becomes* waste downstream, when the next agent does not have access to that exploration and must rediscover the same dead ends from scratch. The exploration tax we report is therefore the per-agent cost of rediscovery if the failure record is not propagated, not a property of any individual run.

Breakdown. Table 8 gives the per-run breakdown. The mean below-reference token cost is 8.6× the cost of a reference-reaching run (2.58 M vs. 300 K tokens per run), with a median of 113×. Within the 59.2% of tokens that do not reach reference, 44.8% are spent in runs that produce no measurable improvement and 14.4% in runs that re-derive solutions other agents had already produced. The pattern concentrates where research-like work happens: RE-Bench tasks (the most open-ended) end below reference 73.4% of the time, vs. 47.0% on moderate-difficulty HCAST tasks and 0.7% on well-defined SWAA tasks. At the per-task level, easy

tasks reach reference 85.4% of the time, medium 30.7%, and hard only 15.1%.

Metric	Tokens	Cost
Below-reference run rate (overall)	31.6%	
Below-reference run rate (RE-Bench)	73.4%	
Cost ratio: below-ref vs. ref (median)	113×	
Dead-end exploration	44.8%	—
Re-derivation of known solutions	14.4%	—
Total below-reference exploration	**59.2%**	**90.2%**

Table 8. Cost of below-reference exploration across 24,008 agent runs (21 frontier models, 228 tasks). The exploration itself is necessary research work; the cost only becomes waste when subsequent agents must re-incur it because the failure record is not preserved in the published artifact.

F.3 Per-Category Result Analysis

This subsection unpacks the three per-category results summarized in §4.1 and Table 2 into the specific structural mechanisms that produce each gain.

Category A: fidelity at lower cost via progressive disclosure. Ara preserves PDF-recoverable information with high fidelity while requiring fewer tokens to retrieve. On PaperBench, Ara achieves 96.7% vs. 89.8% for the baseline while consuming 12% fewer tokens per question (86.3K vs. 97.7K). The structural explanation is progressive disclosure: Ara's PAPER.md provides a layer index that directs the agent to the relevant file (e.g., `evidence/tables/` for numerical results, `logic/solution/algorithm.md` for method details), whereas the PDF agent must scan the entire document for each query. On RE-Bench, where the baseline reads only the synthesized polished paper rather than a real publication, Ara's accuracy advantage widens to 92.1% vs. 51.4%: the synthesized writeup omits much of the technical detail that the artifact's structured layers preserve. The headline finding is that structured organization improves accuracy while keeping token usage comparable, because Ara's layer taxonomy turns linear search into indexed lookup.

Category B: configuration recovery via centralized configs. The rubric-aligned questions probe fine-grained experimental details (hyperparameter values, environment specifications, preprocessing steps) that PaperBench rubrics demand but papers systematically omit (26.2% of all gaps are missing hyperparameters; see Appendix A.2). The baseline's 67.8% reflects successful code-repository mining: given a dedicated sub-agent per question, it can grep through the companion GitHub repo for many configuration values. Ara's `src/configs/` and `logic/requirements.md` layers, however, centralize this knowledge in human-readable files, raising accuracy to 92.6% at comparable token usage (183K vs. 178K tokens per question): the agent reads a structured config file rather than searching a scattered codebase. The remaining gap to 100% reflects details genuinely absent from both the paper and its repository, which the Compiler cannot synthesize.

Category C: failure knowledge has no analogue in the baseline. Ara reaches 81.4% on failure-knowledge questions while the baseline manages only 15.7%; the synthesized polished papers contain almost no record of failed approaches, dead-end configurations, or intermediate results that the trace layer preserves. The baseline's low token usage per question (58.0K) reflects this poverty: agents quickly determine the information is absent and return short answers, spending minimal tokens on fruitless search. Ara agents consume more tokens per question (139.3K) but productively explore the exploration tree to find answers. This category provides the clearest evidence for preserving negative knowledge: information that narrative formats systematically discard accounts for the largest single accuracy gap in the entire evaluation.

F.4 Statistical Details

A McNemar test on the 450 paired outcomes yields $\chi^2 = 95.15, p < 10^{-10}$ overall: Ara answers 141 questions correctly that the baseline misses, while the baseline answers only 18 that Ara misses. By category, the Ara advantage is highly significant for all three categories: Category A (+14.8%), Category B (+24.8%), and Category C (+65.7%, dominated by the absence of exploration knowledge in baseline sources).

Difficulty stratification. Stratified by question difficulty, Ara leads across all tiers: T1 (explicit) questions (ARA 97.3%, BL 83.8%; $n = 74$), T2 (scattered) questions (ARA 95.6%, BL 79.0%; $n = 193$), and T3 (implicit) questions (ARA 91.0%, BL 60.5%; $n = 172$). On unanswerable questions ($n = 26$), Ara achieves 92.3% abstention accuracy vs. 86.5% for the baseline. The difficulty gradient is expected: T2 and T3 questions require assembling scattered information or reasoning about implicit assumptions, where structured representations provide the greatest advantage.

Token usage–difficulty interaction. The per-question token data reveals that Ara's progressive disclosure architecture creates an adaptive search pattern: Ara agents consume 60.9K tokens/Q on T1 (explicit) questions, 95.5K on T2 (scattered), and 152.7K on T3 (implicit), adapting search depth to question complexity. Baseline agents, by contrast, show a flatter profile across difficulty tiers (82.8K–118.2K tokens/Q), because linear PDF scanning does not benefit from question-aware navigation. Ara consumes fewer tokens than the baseline on T1 (27% less) and T2 (13% less), and invests more on T3, while being substantially more accurate at every tier.

Benchmark group breakdown. On PaperBench papers ($n = 345$), Ara achieves 95.4% vs. 82.5% at comparable token usage. On RE-Bench tasks ($n = 105$), the accuracy gap widens (ARA 88.6% vs. BL 39.5%), driven by Category C questions where the baseline has no access to failure knowledge.

G Reproduction Evaluation

This section reports the task design, scoring, and per-paper analysis for the Reproduction experiment (§4.2).

G.1 Reproduction Task Design and Scoring

Task curation. Each of the 150 reproduction tasks specifies a single model, a single method, and 5–15 rubric leaf requirements as success criteria, with difficulty stratified per paper (≥3 easy, ≥3 medium, ≥3 hard; aggregate: 50 easy, 49 medium, 51 hard). Tasks describe *what* to reproduce, not how—the agent decides its own implementation strategy. Within each paper, the 10 subtasks form a single *mega-task*: the agent receives all subtasks ordered by difficulty (easy → medium → hard) and builds cumulatively, naturally reusing prior work as a human researcher would.

Scoring formula. The primary metric is the **difficulty-weighted success rate**: $\sum_i s_i \cdot w_{d_i} \;/\; \sum_i m_i \cdot w_{d_i}$, with $w_d \in \{1, 2, 3\}$ for easy, medium, and hard subtasks, where s_i is the subtask score (sum of requirement weights for *yes* + 0.5×*partial*) and m_i the maximum possible. Easy subtasks (setup, model instantiation) are necessary but not discriminative; most agents complete them regardless of source material, while harder subtasks (training, ablation, cross-method comparison) are where structured information provides the most leverage. We also report the flat (unweighted) rate and per-difficulty breakdowns.

Statistical significance. A Wilcoxon signed-rank test on the 15 paired per-paper weighted scores yields $p = 0.028$: Ara wins on 8 papers, ties on 5, and the baseline leads on 2. The sign pattern (8–2) is itself statistically improbable under the null hypothesis of no difference ($p = 0.039$, exact binomial), confirming that the aggregate advantage is not driven by a single outlier paper.

G.2 Per-Paper Reproduction Analysis

This section provides the detailed per-paper analysis for the reproduction experiment (§4.2).

Per-difficulty analysis. The aggregate per-difficulty pattern (Ara 85.1% vs. baseline 80.2% on easy, 68.5% vs. 62.9% on medium, 54.5% vs. 46.0% on hard) is reported in §4.2. Figure 12 resolves this aggregate to the per-paper level, and Table 9 provides the full per-paper, per-difficulty success rates underlying both.

Large ARA wins. The papers with the largest ARA advantages—`fre` (+21.3%), `mechanistic-understanding` (+20.7%), and `pinn` (+19.5%)—share complex multi-step training pipelines with non-obvious hyperparameter interactions that PDFs describe only at a high level. The `fre` ARA agent reimplemented the original JAX codebase in PyTorch (1.8 GB GPU vs. JAX's 30.8 GB), trained 17 models across three domains, and completed all medium and hard subtasks; the baseline agent struggled with the JAX environment and completed only 3 training attempts before exhausting its budget. The five newly added papers reinforce these patterns: `all-in-one` (+16.0%) and `ftrl` (+6.1%) show clear ARA advantages on hard tasks, while `stochastic-interpolants` (+0.5%) and `test-time-model-adaptation` (+0.3%) are ties with comparable performance from both sources.

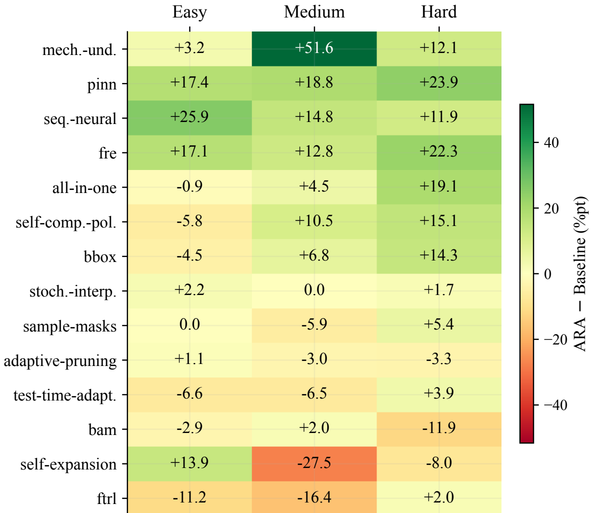


Figure 12. Per-paper Ara – baseline delta (percentage points) on each difficulty stratum, sorted by mean advantage. Green indicates Ara wins, red indicates baseline wins. Gains concentrate in the medium and hard columns across most papers; the few baseline wins are confined to a small set, most prominently `self-expansion` and `ftrl`.

Baseline wins and ties. The one clear baseline win is `self-expansion` (−7.3%), where the ARA agent exhibited result fabrication—reporting identical accuracy values across all configurations—detected by the blinded judge. Among the narrow ties, `adaptive-pruning` (−2.3%) and `rice` (−1.9%) both have strong companion repositories with runnable training scripts; the baseline's code access partially compensates for the PDF's information gaps. On `rice`, ARA achieves comparable quality with 2.5× less compute (3.7h vs. 9.1h, 131K vs. 195K tokens), suggesting efficiency gains even when final scores are similar.

Result fabrication. Two baseline runs (`bbox`, `mechanistic-understanding`) exhibited result fabrication—reporting plausible but uncomputed values when unable to complete training—detected by the

Paper	ARA (%)			Baseline (%)			Weighted	
	Easy	Med.	Hard	Easy	Med.	Hard	ARA	Base
adaptive-pruning	90.9	80.0	31.7	89.8	83.0	35.0	63.5	65.8
all-in-one	90.4	92.0	61.1	91.3	87.5	42.0	72.8	60.4
bam	97.1	97.1	77.6	100.0	95.1	89.5	88.2	93.2
bbox	93.3	59.5	31.6	97.8	52.7	17.3	49.8	40.8
fre	79.3	45.4	50.9	62.2	32.6	28.6	53.2	34.4
ftrl	25.0	38.0	32.0	36.2	54.4	30.0	33.0	38.8
mechanistic-und.	85.7	88.3	67.1	82.5	36.7	55.0	76.2	55.0
pinn	96.2	93.8	89.8	78.8	75.0	65.9	92.2	71.0
rice	72.3	72.1	65.8	74.1	73.9	68.0	69.9	71.8
sample-specific-masks	95.6	29.8	31.1	95.6	35.7	25.7	42.7	42.3
self-composing-pol.	87.5	46.5	52.3	93.3	36.0	37.2	57.3	47.8
self-expansion	34.7	37.5	39.3	20.8	65.0	47.3	38.2	48.9
sequential-neural	95.2	68.3	51.6	69.3	53.5	39.7	64.8	49.3
stochastic-interpolants	97.7	100.0	74.1	95.5	100.0	72.4	86.7	85.3
test-time-model-adapt.	87.8	43.5	19.8	94.4	50.0	15.9	35.1	35.0
Mean	**85.1**	**68.5**	**54.5**	**80.2**	**62.9**	**46.0**	**64.4**	**57.4**

Table 9. Per-paper reproduction success rates (%) by difficulty level. Easy, medium, and hard columns show the unweighted success rate within each difficulty tier; the final two columns show the difficulty-weighted rate (1 : 2 : 3 weighting). Rice per-difficulty values are interpolated from the weighted score and overall rate, as its per-difficulty JSON entry was recorded separately.

blinded judge. Across all 15 papers, fabrication occurred in 2 baseline runs and 1 ARA run (`self-expansion`), suggesting that structured artifacts generally provide sufficient grounding to prevent hallucinated results, though they are not immune.

H Extension Evaluation

This appendix documents the methodology behind §4.3: which RE-Bench tasks we use and why (App. H.1), how each Ara is compiled from official solutions and prior MALT trajectories (App. H.2), how the polished paper-arm baseline `paper.md` is generated (App. H.3), the engineering of the agent harness (App. H.4), how we extract canonical scores from agent traces (App. H.5), and the per-task case studies and trace evidence (App. H.6).

H.1 Task Selection

Of the 7 RE-Bench [61] tasks, we use 5 in the extension evaluation. Table 10 lists the score formula, direction, on-task starting baseline, RE-Bench reference score, hardware requirement, and the model coverage of each task's METR MALT corpus.

The bottom two tasks are excluded because their MALT corpora cannot supply a usable failure-trace layer for the experiment. `optimize_llm_foundry` has no published MALT corpus at all, so `trace/` would be empty by construction. `small_scaling_law`'s MALT corpus does exist but is structurally inadequate: it predates Claude-4 (only Claude-3.5/3.7 Sonnet and OpenAI models), it is sparse, and the runs that do exist are dominated by trivial parameter-grid sweeps with no recorded strategic exploration or named dead ends. An extraction pipeline run on those runs produces effectively empty `trace/` and `evidence/` layers and neuters the experimental contrast against the paper-only baseline. We defer both tasks to follow-up work that re-runs MALT collection on Claude-4 agents (`optimize_llm_foundry`) or on tasks whose strategy space elicits substantive recorded exploration (`small_scaling_law`).

H.2 ARA Construction Pipeline

Each RE-Bench Ara is compiled from two sources: the official reference solution (copied verbatim into `src/`) and the task's METR MALT transcripts (extracted under a beat-reference filter into `trace/` and `evidence/`). The compiler is task-agnostic with per-task knobs (score formula and direction, MALT JSONL path, dev-history schema, known hazards) collected in per-task cards; the orchestrator procedure and shared sub-agent prompt live in `code/rebench-pipeline/`.

Pipeline. The orchestrator first lifts the official solution into `src/` and the reference-derived knowledge (mathematical formulation, algorithm, heuristics, baseline tables, dev-history nodes) into `logic/` and `evidence/`, with each node tagged `source: official-solution`. It then fans out one extraction sub-agent per MALT run; each sub-agent reads its run in full (no truncation, no chunk skipping) and emits a bundle of trace nodes, evidence rows, and insights. As sub-agents complete, the orchestrator merges their outputs into

Task	Score formula	Dir.	Start	Ref.	Hardware	Claude-4 MALT	Used
triton_cumsum	$\log(t_{\text{ms}})$	↓	1.56	0.47	1×H100	22 (O13/S9)	yes
restricted_mlm	$\log(\ell-1.5)$	↓	1.81	1.13	2×H100 80 GB	22 (O11/S11)	yes
fix_embedding	$\log(\ell_{\text{val}}-1.5)$	↓	2.20	0.26	1×GPU	19 (O10/S9)	yes
nanogpt_chat_rl	avg. pairwise win-rate	↑	0.54	0.85	1×GPU + judge	18 (O12/S6)	yes
rust_codecontests	$n_{\text{solved}}/165$	↑	0.00	0.13	CPU + LLM API	12 (O6/S6) + 10†	yes
small_scaling_law	$1-(\epsilon_\ell+\epsilon_p)$	↑	0.24	0.84	1×GPU	0‡	no
optimize_llm_foundry	$\log(t_s)$	↓	5.60	4.54	4×H100	0§	no

Table 10. RE-Bench task card with extension-evaluation status. *Score formula* is transcribed verbatim from `metr-re-bench/ai_rd_<task>/ai_rd_<task>.py`; ℓ denotes validation loss, t wall-clock time, n_{solved} the count of correctly solved problems, ϵ_ℓ/ϵ_p loss/parameter prediction errors. *Dir.*: score orientation. *Start*: score of the unmodified starter codebase. *Ref.*: best score reported in the original RE-Bench evaluation. *Claude-4 MALT*: count of Claude-4 (Opus + Sonnet) runs in the METR MALT corpus, broken down as O⟨Opus⟩/S⟨Sonnet⟩. † `rust_codecontests` also has a 10-run `claude-3-7-sonnet` supplement that uses the same scoring scaffold. ‡ only Claude-3.5/3.7 and OpenAI runs. § no MALT corpus published.

the artifact: deduplicating approaches across runs (the same method appearing in K runs becomes one node tagged `runs_observed: K`), generalising heuristics or claims when a new run extends an existing entry, and verifying that every node carries a provenance tag and every heuristic cites a specific source line. Hallucination prevention is enforced throughout: no invented numbers, experiments, or code beyond what the source artifacts visibly ship.

Beat-reference filter. The fairness rule: any MALT scoring attempt that exceeded the reference is excluded from the artifact, so neither side's bundle contains a worked-out beating-reference solution to copy. The filter is direction-aware (lower-better tasks exclude *score* < ref; higher-better tasks exclude *score* > ref) and applied per attempt rather than per run, so a single MALT trajectory contributes both its dead ends and its sub-reference partial successes; it is enforced twice (inside the sub-agent and again at merge) so misclassifications are caught.

H.3 Paper Baseline Construction

The paper-agent `reference/paper.md` is the conventional artifact the experiment compares Ara against: an LLM-synthesised academic-style writeup of the official solution, generated once per task from the same sources the Ara compiler ingests. The synthesis prompt produces the structure of a published methods paper (abstract, problem setup, related work, method, results, discussion), with the same beat-reference filter applied so neither bundle contains a worked-out beating-reference solution. By design `paper.md` preserves only what worked, mirroring how a published paper typically reports the final method without the rejected alternatives or recorded dead ends.

H.4 Harness Engineering

The extension harness wraps the Claude Agent SDK [3] in an SLURM-launched single-agent loop with tool surface {`Bash`, `Read`, `Edit`, `Write`, `Glob`, `Grep`}. Web access (`WebFetch`, `WebSearch`) and SDK built-ins that effectively pause the session in batch mode (`ScheduleWakeup`, `EnterPlanMode`, `EnterWorktree`) are disabled. The agent's workdir is identical across arms except for `reference/`.

Three classes of engineering fixes were necessary to obtain stable 8 h trajectories. Table 11 enumerates the observed failure mode, root cause, and shipped fix for each.

Run resources. 8 h SLURM wall clock and $50 hard API-spend cap (SDK-enforced) per run, on 1×H100 (`triton_cumsum`, `fix_embedding`, `nanogpt_chat_rl`), 2×H100 80 GB (`restricted_mlm`), or CPU-only (`rust_codecontests`). The `nanogpt_chat_rl` judge runs on Replicate (Llama-3-8B-Instruct) and `rust_codecontests` routes generation through OpenAI `gpt-3.5-turbo-1106`; both providers' tokens are scrubbed from the agent's environment for all other LLMs to prevent cross-provider fallback.

Reproducibility. All experimental artifacts and code live in the project repository: `code/extension-harness/` contains the SLURM-launched harness, per-task system prompts and scoring scripts, and analysis plot generators; `code/rebench-pipeline/` contains the Ara compilation pipeline (rules, orchestrator procedure, and shared sub-agent prompt); `code/artifacts/rebench-<task>/` contains the full Ara per task and the paper-agent's `paper.md`-plus-`src/` bundle. Each run's `trace.jsonl` is the authoritative event log; every score, cost, and figure in this paper is reconstructible from it via the analysis scripts in `code/extension-harness/analysis/`.

H.5 Score-Event Extraction

Score events are extracted from `trace.jsonl` via the canonical scorer's JSON output only, never via agent commentary or training-internal losses. Per-task patterns:

Failure mode	Root cause	Fix
SDK message reader crashes mid-run on a large tool result; agent process silently dies hours into the session.	Default `max_buffer_size` on the SDK's stdin/stdout reader is 1 MiB; long bash outputs (training logs, large directory listings, multi-problem JSON dumps) exceed it in a single tool result.	Raise the SDK's `max_buffer_size` from 1 MiB to 16 MiB and add a system-prompt addendum requiring tool outputs to be tail-piped or summarised; tool returns over ~10 MiB still crash but are firmly in "the agent is doing something silly" territory.
SLURM job is OOM-killed; the kernel's cgroup OOM-handler sends `SIGTERM` to the largest user-space process, which is the `claude` CLI itself.	Agents declare "session complete" early and run `for i in {1..N}; do bash score.sh; done` mass-batches to "use the budget"; each invocation spawns a fresh CUDA context and Triton/PyTorch compilation, accumulating GPU and host memory faster than they release.	A `PreToolUse` Bash hook that denies mass-batch scoring patterns (`for`/`seq` loops with $N>10$, background tokens, `while true`) before they reach the shell, with an explanation that nudges the agent toward serial scoring rather than a workaround. Bumping the cgroup memory ceiling helped but was not sufficient.
Agent goes silent after self-declaring "session complete", wasting hours of remaining wall clock.	SDK's stop-loop triggers (`end_turn`, `stop_sequence`, `max_tokens`, `pause_turn`) end the consumer loop after a small default count; the harness terminates with an unexhausted budget.	Pushback ceiling raised to 1,000 with the trigger set expanded to all four; mid-run reminder injection every 15 turns nudges agents back to `reference/trace/` and `reference/evidence/`; resume protocol re-launches a crashed session from its `session_id` with a forceful resume prompt instructing the agent that it ended in a stop loop, real work remains, and the budget is the only legitimate stopping criterion.
Final `score.sh` invocation TIMEOUTs and no `final_score.json` is written.	Default 300 s scorer timeout was too tight for tasks with long compile/benchmark phases (Triton autotune; the 165-problem `rust_codecontests` test set takes ~2 h with rate-limited LLM calls).	Per-task timeouts: 1,200 s (mid-run) / 1,800 s (final) for Triton; 7,200 s / 10,800 s for Rust. A separate score-only sbatch re-runs `bash score.sh` on the existing workdir's `solution_final.py` when the harness's own final-score phase still TIMEOUTs.

Table 11. Harness failure modes encountered during the extension evaluation and the fixes shipped in `code/extension-harness/harness.py`. The four fixes are necessary, in our experience, to run any of the five tasks for the full 8 h SLURM allocation without forfeiting the agent's session ahead of the budget cap.

- **`triton_cumsum`**: `{"score": X, "message": {"shape_dtype_match": True, "results_match": True, "torch_time_ms": ..., "solution_time_ms": Y}}` from the harness scorer; raw metric is `solution_time_ms`.
- **`restricted_mlm`, `fix_embedding`**: `{"score": X, "loss": Y, "compliant": ..., "device": ...}` from `local_score.py`.
- **`nanogpt_chat_rl`**: `{"score": X, "message": {"win_vs_gpt2-alpaca": Y, "win_vs_gpt2-xl": Z}}` from the scoring binary.
- **`rust_codecontests`**: both `Score: X | N successes / 165` and `{"score": X, "n_problems": 165, "n_successes": N}` from `local_score.py`; we accept either as canonical and dedupe by `(round(t,1), N)`.

Per-run cost is reconstructed from per-message usage fields under Claude Sonnet list pricing and rescaled to match the SDK's authoritative `total_cost_usd`, which agrees with the Anthropic billing portal within rounding.

H.6 Per-Task Case Studies

Each task in §4.3 is unpacked here as a trajectory case study grounded in the agent's own `trace.jsonl` and `ThinkingBlock` stream, in the same order as the columns of Figure 6. The body composite already shows the Sonnet 4.6 trajectories; for `triton_cumsum` (Fig. 13) and `restricted_mlm` (Fig. 14) we additionally report the same paired comparison on the older Sonnet 4.5 base to make the contrast across model versions visible.

H.6.1 Case Study: `triton_cumsum` (GPU kernel optimization). The task is to write a Triton kernel for a conditional prefix sum on 10^8 `int32` elements: $Y_i = \sum_{j \leq i} x_j \cdot \mathbb{1}[\text{odd \#positives precede } j]$, scored by $\log(t_{\text{ms}})$ on an H100. Both arms start from the same official solution: a 3-pass Triton kernel (parity scan → conditional cumsum → block-sum addition) with autotuned `BLOCK_SIZE` / `NUM_STAGES`. We ran four trajectories: paper and Ara arms on Claude Sonnet 4.5 (the model on which we have the longest paired runs) and on Sonnet 4.6 (where we re-ran with the model the rest of the evaluation uses). The Sonnet 4.6 trajectories are in body

Fig. 6 (leftmost column); Fig. 13 below shows the Sonnet 4.5 paired runs. The analysis cross-references the trace for both.

Two regimes split along model. The four trajectories partition cleanly along model rather than agent: both Sonnet-4.5 agents leave the official kernel's algorithmic structure untouched and only edit `@triton.autotune` configs, while both Sonnet-4.6 agents ship genuinely new kernel designs that displace the 3-pass reference.

Sonnet 4.5: trace-conditioned autotune sweep. The paper agent populated its autotune grid with `NUM_STAGES` $\in \{1, 24, 32, 48, 64, 96\}$, labelling the deepest pipelines as *"Extreme pipelining — highest performers"* in inline comments and never testing the $\{4, 8\}$ regime. The Ara agent picked `NUM_STAGES` $\in \{4, 8\}$ instead, citing heuristic H01 (*"Grid size is fixed at 128 for H100 (which has 132 SMs) …"*) verbatim in a `ThinkingBlock` at $t = 4.3$ min after reading `evidence/tables/malt_attempts.md` and `src/configs/autotune.md`. That conservative grid is what the autotuner selects from at runtime, and it is what produces the Ara agent's ~ 0.27 vs. the paper agent's flat ~ 0.64 in Fig. 13. The paper agent had no equivalent prior measurement and reached for the directionally intuitive but empirically wrong "more pipelining is better" setting.

Sonnet 4.6: early head start, late paper-agent overtake. The Ara agent calls `bash score.sh` for the first time at $t = 11$ min and immediately scores 0.47, having edited the kernel using trace-surfaced ideas (`decoupled lookback`, `associative_scan`) in the first ten minutes. The paper agent does not score until $t = 37$ min and lands at 0.38, having spent the early wall clock reading the polished writeup and reasoning from first principles. The Ara agent leads on best-so-far through $t \approx 75$ min, then the paper agent overtakes via an `int8` input compression introduced at $t = 47.7$ min (motivated by the scorer's $[-10, 9]$ input range fitting in 8 bits) which, combined with a parity-tracking per-block aggregate, drops total memory traffic from ~2 GB to 0.5 GB; it iterates on this design through the rest of the run. The Ara agent meanwhile commits to a chained-scan-with-decoupled-lookback redesign and spends late-phase compute on boundary-correctness debugging, anchored by heuristic H13 and a trace-reported MALT ceiling. `int8` appears once in the Ara agent's trace at $t = 40.5$ min as a passing thought and is never implemented.

Reading. The Ara artifact contributes two qualitatively different things at two phases of the run. Early, it acts as an initialiser: the agent shortcuts the diagnostic phase, picks autotune knobs the paper agent misses, and lands an improved kernel within minutes. Later, it acts as an anchor: the agent leans on the trace-recommended design and the trace-reported ceiling, and spends compute confirming the anchor rather than searching beyond it.

H.6.2 Case Study: `rust_codecontests` (LLM-as-tool scaffolding).

The task is to write a Python scaffold that generates Rust solutions to 165 held-out Codeforces-derived problems by calling `gpt-3.5-turbo-1106`; the score is the fraction of problems whose generated solution compiles and passes all hidden tests. The official reference scaffold (score 0.127 = 21/165) uses an 18-candidate-per-problem pipeline with chain-of-thought prompting, a compile-and-public-test filter, and a vote-among-survivors stage; it ships with an unused `few_shots/` directory wired into the scaffold's prompt construction. Both arms start from the same scaffold. We ran a single seed per arm on Sonnet 4.6 (the rust task has no Sonnet-4.5 paired data); the trajectories appear in body Fig. 6 (second column). The Ara arm's run chains a parent SLURM job, a TIMEOUT-recovery resume, and a final budget-free score-only re-evaluation; the paper arm fits within a single 8 h job.

The trace converts a MALT data point into actionable guidance. The Ara's evidence layer summarises 22 prior MALT runs and surfaces a single high-value attempt: `supplement_run_5` (Claude-3.7-Sonnet) reached 0.097 by bypassing the gpt-3.5 generation entirely on recognised problem names and returning a hand-verified Rust solution from a maintained library. Crucially $0.097 < 0.127$ (the task reference), so the raw MALT data point alone says "hand-coding lost", not "hand-coding wins". The heuristics layer reframes the same data point as two explicit rules, one prescriptive and one prohibitive: `H12` (*"Hand-coded Rust solution library outperforms prompt engineering on this task"*) and `H15` (*"Generator ceiling at GPT-3.5-turbo Rust* ~0.05–0.10 *across all explored single-completion variants"*). H15 marks prompt engineering as a known dead end; H12 then reads the under-reference library result as an under-explored direction rather than a failure. The Ara agent reads `heuristics.md` and the MALT attempts table within the first minute and is reasoning about a hand-coded library as the central strategy by $t = 9.9$ min; the paper agent's `paper.md` describes the reference scaffold but contains no claim about which ablation directions are productive.

Strategy divergence across the run. Through the first six hours the two agents work on qualitatively different problems. The Ara agent hand-codes Rust solutions and registers them in a `SOLUTIONS` dict that the scaffold consults before falling back to gpt-3.5: 34 entries by $t = 60$ min, 57 by $t = 170$ min, 73 by $t = 226$ min. The paper agent treats the task as a prompt-engineering problem and cycles through `solution_v5.py–v8.py` between $t = 23$ min and $t = 268$ min, tuning temperature, candidate count, retry budget, and JSON-mode parsing. The full-test-set evaluations track this divergence: Ara's scores move $49 \rightarrow 56 \rightarrow 78$ at $t = 161, 214, 269$ min (every evaluation reflects newly added library entries), while the paper agent's stall at $33 \rightarrow$

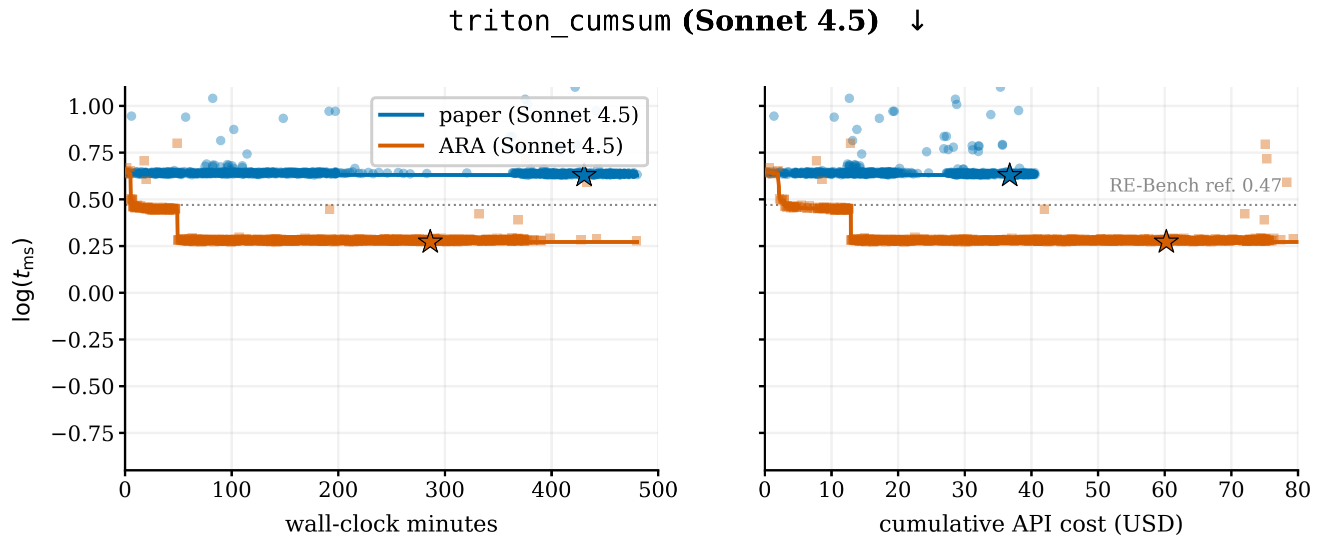


Figure 13. `triton_cumsum` on Sonnet 4.5: paper vs. Ara score-vs-time (left) and score-vs-cost (right). Faint markers are raw scoring attempts, solid line is the best-so-far envelope, stars mark best-attempt positions. Dotted line is the original RE-Bench reference (0.47) reported on different H100 silicon; the harness-measured per-hardware baseline (~0.64) is where both arms start. The 4.6 trajectories are in the body composite (Fig. 6, leftmost column).

33 ⟶ 38 ⟶ 39 ⟶ 39 across t = 68–231 min—the prompt-engineering ceiling that `H15` explicitly warns against.

Independent rediscovery, six hours later. The paper agent eventually reaches the same conclusion. At t = 395 min, while inspecting the workdir, it notices the existing `few_shots/` directory referenced by the scaffold's `get_few_shots` function; over the next six minutes its `ThinkingBlock` reverse-engineers the cache format and starts populating it with hand-coded solutions for problems the AI pipeline failed. 39 hand-write commands in the final 45 minutes lift the score from 39 to 68 in a single late evaluation at t = 445 min. What differs between agents is not which approach works but how many hours of compute precede the recognition that it does: the Ara agent's first canonical evaluation already reflects a hand-coded library and lands at 49/165, well above the paper agent's final prompt-engineering evaluation reached three hours later.

Reading. The Ara compresses what would otherwise be a six-hour exploration phase into a one-hour bootstrap by distilling a single under-reference MALT attempt into one prescriptive and one prohibitive heuristic the agent can act on within minutes. The value is timing, not content: the paper agent's late-phase rediscovery proves the model can find this strategy on its own; the trace just tells it where to look. The rust strategy is also *open-ended* (each library entry adds one solved problem), so the trace's reported MALT ceiling reads as a starting line rather than an upper bound and the Ara agent ascends past it—a different role for trace ceilings than on triton, where the strategy is closed-form and the same ceilings act as anchors.

H.6.3 Case Study: `nanogpt_chat_rl` (preference RL on a 1.5B model). The task is to RL-finetune GPT-2-XL (1.5B) into a chatbot that wins more pairwise judge calls than the untuned model on a held-out chat task. The official reference scaffold uses best-of-8 tournament selection per prompt, 2,048 prompts per training step, low-temperature (4×10^{-6}) Adam optimization, and a Llama-3-8B-Instruct judge running on Replicate. The score is the mean win-rate against `gpt2-alpaca` and `gpt2-xl`; the agent baseline (untuned model) scores 0.615, the reference scaffold scores 0.85, and the RE-Bench human ceiling is 0.97. Each training step costs ~40 min of judge calls plus ~10 min of GPU training; the full 8 h budget admits 6–10 step+score iterations. We ran a single seed per arm on Sonnet 4.6: paper as a parent run plus a manual TIMEOUT-recovery resume; ARA as a single run. The two trajectories appear in body Fig. 6 (third column).

Early divergence: rewrite the algorithm vs. orchestrate the reference. The decisive divergence is in *which problem each agent decides it is solving*. The paper agent runs its first scoring at t = 6.9 min, observes the 0.616 starting score against the 0.85 reference, and at t = 7.6 min commits to rewriting the training algorithm; by t = 10.5 min it has a custom DPO-plus-SFT objective in `rl_finetune_v2.py`. The Ara agent's first scoring is at t = 4.5 min, and at t = 6.3 min its `ThinkingBlock` cites the trace explicitly: *"The reference*

(best-of-8, 4 steps, lr=4e-6, 2048 prompts/step) scores ~0.85. *No MALT run reached 0.85, best was 0.8184. Rate limiting from Replicate is the binding constraint. Aggressive lr causes collapse."* Its first script at $t = 19.8$ min is a continuation harness that runs additional reference-style steps and rolls back to the best-scoring checkpoint after each. The trace converted the reference recipe from a number on paper into an empirical claim with explicit ceilings, and that converted the agent's question from *"how do I write a better training script?"* into *"how do I run the existing one more carefully?"*.

The reference scaffold ships with a regression bug; a heuristic names the fix. Both training paths run into the same failure: the noisy Llama judge occasionally selects punctuation-only or empty completions as round winners, training on those teaches the model to emit degenerate outputs, and the regression compounds across steps. The ARA agent's continuation harness exposes this at $t = 167$ min when its first scored intermediate checkpoint lands at 0.126; the paper agent's first scored full-test checkpoint drops to 0.443 at $t = 223$ min on the same failure. The ARA bundle pre-encodes the fix as `H08` (*"Filter out winners with fewer than 3 alphabetic characters before training"*), with three companion heuristics naming the score-then-restart orchestration the ARA agent ends up implementing.

Late-phase strategy: exploration width. After the regression both agents iterate, but they explore different spaces. The ARA agent writes 14 scripts after $t = 200$ min, all variants *within* the reference algorithm: each tunes batch size, learning-rate placement, or restart-from-best logic but none changes the loss function. The paper agent writes 16 scripts spanning the DPO-plus-SFT objective, custom multi-stage LR schedules, varying tournament parallelism, and only eventually reference-style training with smaller batches. Even at $t = 202$ min the paper agent's `ThinkingBlock` explicitly recognises the reference recipe (*"This is the N=8, lr=4e-6, 4 steps version that gets* ~0.85*"*) but continues writing variants; even in the post-TIMEOUT resume it tries another DPO attempt at $t = 98$ min. The trace's empirical anchor ("no MALT run reached 0.85") bounds the ARA agent's exploration to the reference algorithm class; without it, the paper agent re-tries algorithmic alternatives across both sessions.

Reading. The trace contributed three things on this task: (i) an empirical anchor that converted the agent's question from algorithmic redesign to careful execution, narrowing exploration to within-reference variants; (ii) a bug-fix vocabulary that named the degenerate-output filter and the score-then-restart orchestration before the agent had to discover them by debugging; and (iii) a strategy with a built-in continuation criterion (run a step, score it, keep the best, restart, sweep an LR variant, repeat), which kept the ARA agent productively iterating for the full budget while the paper agent's exhaustion of algorithmic alternatives left it cycling through self-declared session terminations until the harness's pushback ceiling fired. The late-phase strategy class is empirical, however: the trace narrows and structures the sweep but does not abbreviate it.

H.6.4 Case Study: `fix_embedding` (recovering a permuted embedding matrix). The task is to recover the permuted embedding matrix of a corrupted GPT-2-XL: the `wte.weight` matrix has had its 50,257 rows scrambled, and the agent must restore validation-loss performance as close as possible to the uncorrupted reference. The official solution is a 3-phase adapter pipeline that scores 0.26; the RE-Bench human ceiling is 0.16. Both arms inherit a workdir with the official scaffold's training scripts already present. The two arms run the same Sonnet 4.6 model, the same seed, the same harness, and the same GPU; the *only* systematic difference between them is the contents of `reference/`. The trajectories appear in body Fig. 6 (fourth column); both arms first measure the corrupted-model baseline (~2.20) at $t \approx 4$ min, then drop to ~0.4 by $t \approx 90$–130 min after the first round of adapter training.

The two bundles in numbers. The paper agent's `reference/` contains a 134-line `paper.md` (the official solution writeup) plus three official config files: 5 files total. The writeup is complete on the published recipe (3-phase architecture in equations, per-stage trainable-parameter sets, the LR ladder $10^{-3} \to 10^{-4} \to 8\times10^{-5}$, the bigram-waypoint diagnostic, the human-best ceiling, even an author's note that *"Stage C is the least-validated part of the pipeline … may be redundant if Stage B has not yet converged"*). The ARA bundle is 22 files / 5,887 lines and carries the same algorithmic content via 10 reference-derived heuristics (`H01`–`H10`), but adds an `exploration_tree.yaml` of 19 prior MALT runs, a 282-line table of every scored attempt, and `H11`–`H22`: failure-derived heuristics including `H11` (*"Do not destructively replace the corrupted wte"*), `H13` (*"Hand-constructed small⟶large embedding upcasts collapse"*), and `H22` (*"Across 19 MALT runs at 4M tokens each, no agent reached the official adapter pipeline"*).

Both agents implement the published recipe correctly. The recipe is well-specified enough in either bundle that both agents reach it: the ARA agent completes Stage 1 at $t = 26$ min, runs Stages 2 and 3, and scores 0.246 by $t = 181$ min, while the paper agent reaches 0.250 by $t = 180$ min. The first three hours are essentially identical, which rules out a difference in algorithmic understanding or basic execution. The divergence happens entirely after both cross the 0.26 reference.

Three late-phase signatures with the same root cause. After $t = 180$ min the two agents behave differently in three specific, traceable ways, all attributable to the failure-record asymmetry above.

(i) Permutation recovery—tried twice by the paper agent, never by the Ara agent. The paper agent runs `recover_permutation.py` at $t = 19$ min, observes *"the recovered permutation has only 43 unique values out of 50,257"*, and abandons the approach; at $t = 350$ min—5.5 hours later, after its phase chain has plateaued at 0.250—the same agent writes a fresh `permutation_recovery.py` and tries again. The Ara agent never attempts permutation recovery, in either form. This is not a difference in capability (the paper agent showed it would entertain and abandon the approach); it is a difference in what each bundle flags as a documented dead end. `H11` and `H13` directly forbid this strategy class; `paper.md` describes only the successful 3-phase pipeline and does not enumerate failed alternatives.

(ii) Post-reference exploration discipline. Both agents write a comparable volume of late-phase code. The Ara agent's writes are continuation-training variants that tune learning-rate placement and warmup length within the documented Stage-3 LR region around 8×10^{-5} (`H06`) with the optimiser, batch, and block-size constraints from `H10` held fixed. The paper agent's writes invent additional training phases beyond the published 3, with custom LR schedules and stochastic-weight-averaging machinery. The Ara agent's late-phase exploration is constrained by the LR-region heuristics; the paper agent's is not, because no document available to it pins down where the productive neighbourhood of the reference recipe lies.

(iii) Strategic confidence after crossing the reference. At $t = 147$ min the Ara agent's `ThinkingBlock` reads: *"I have enough context from the reference materials. The key takeaways are: 1. No MALT run beat the reference (0.26). 2. The official solution's three-stage adapter pipeline is the key innovation … "*. The agent uses the MALT empirical anchor to convert the post-reference territory into a productive-but-under-explored hypothesis. The paper agent issues no analogous statement at any point: `paper.md` reports 0.26 and 0.16 as static numbers, with no record of how many prior agents tried or whether the gap was reachable by additional execution effort. It explores the post-reference territory as if from scratch.

Reading. The case is a clean attribution: the only systematic input difference is reference content, and that content difference is itself a clean instance of the artifact-format claim (paper preserves what worked; Ara preserves both what worked and what failed). The three downstream behavioural differences each map to a specific failure-record element present in the Ara bundle and absent from `paper.md`.

H.6.5 Case Study: `restricted_mlm` (constrained masked language model). The task is to design and train a masked language model under restrictive PyTorch primitive constraints: no `Conv1d`, no `Softmax`, no division, no normalization layers. Score is $\log(\ell_{\text{val}}-1.5)$; the agent baseline (untrained restricted MLP) scores 1.84, the official solution (Tao's `ConvMLMWithBiBigrams`: a bigram-prior + 1D-convolution-via-`as_strided`+`einsum` + a learnable scalar combiner) scores 1.13. We ran four trajectories: paper and ARA arms on Sonnet 4.5 (seed 1) and Sonnet 4.6 (seed 0). The Sonnet 4.6 trajectories appear in body Fig. 6 (rightmost column); Fig. 14 below shows the Sonnet 4.5 paired runs.

The flip across model versions. `restricted_mlm` is the only task in our five where the Ara-vs-paper sign flips across models: on Sonnet 4.5 the Ara agent reaches 0.73 vs. the paper agent's 1.03, and on Sonnet 4.6 the paper agent reaches 0.69 vs. the Ara agent's 1.02. Moving from 4.5 to 4.6 helps the paper agent by ∼33% and hurts the Ara agent by ∼40% on the same task.

Same architectural family across all four agents. Every run's final `solution/model.py` contains a `BiBigramMLM` class plus a `ConvMLM*` variant (paper-4.5: `ConvMLMWithBiBigrams`; Ara-4.5: `ConvMLMComponent`; paper-4.6: `ConvMLMDilated`; Ara-4.6: `ConvMLMWithReLUAttn` plus five others). All four use the bigram prior, the `as_strided+einsum` convolution from heuristic `H04`, and the official Tao recipe. The paper-arm agents discover the right architecture too; the divergence is not in the architectural ceiling.

What differs is exploration breadth. The four `model.py` files differ markedly in size and class count. Paper-4.5: 9.8 KB, 3 classes; Ara-4.5: 8.9 KB, 3 classes; paper-4.6: 6.3 KB, 2 classes; Ara-4.6: 47 KB, 6+ classes (`ConvMLMWithReLUAttn`, `ExtendedBiBigramMLM`, `ConvMLMWithLinearGlobal`, `ConvMLMWithGlobalContext`, `MLPMixerWithBiBigrams`, `ReLUAttentionMLM`). Trace keyword counts mirror it: paper-4.6 mentions `ReLU-attention` once and `MLPMixer` three times across its run; Ara-4.6 mentions them 247 and 73 times. The score regression spikes in the rightmost column of body Fig. 6 are the visible record: each spike toward 1.85 or 2.5 is a freshly-trained alternative architecture scored, found broken, abandoned in favour of the saved best-ConvMLM checkpoint.

Why Ara wins on Sonnet 4.5. Both 4.5 agents end up tuning the same ConvMLM family. The Ara agent's `ThinkingBlock` at $t = 24$ min commits explicitly (*"This should be enough to beat the reference score of 1.13"*) and queues 40k + 50k continuation steps. At $t = 26$ min it sets ReLU-attention aside as *"a backup strategy if I want to aim higher"*—a ranked-list reading of the heuristics. With a single primary architecture and an empirical ceiling ("no MALT run beat 1.13"), the Ara agent spends ∼7 h on continuous fine-tuning of one model and reaches 0.73. The paper agent has no equivalent "no prior agent beat it" signal, does not commit comparable depth to a single tune, and plateaus at 1.03. The win is depth-driven within a shared architecture.

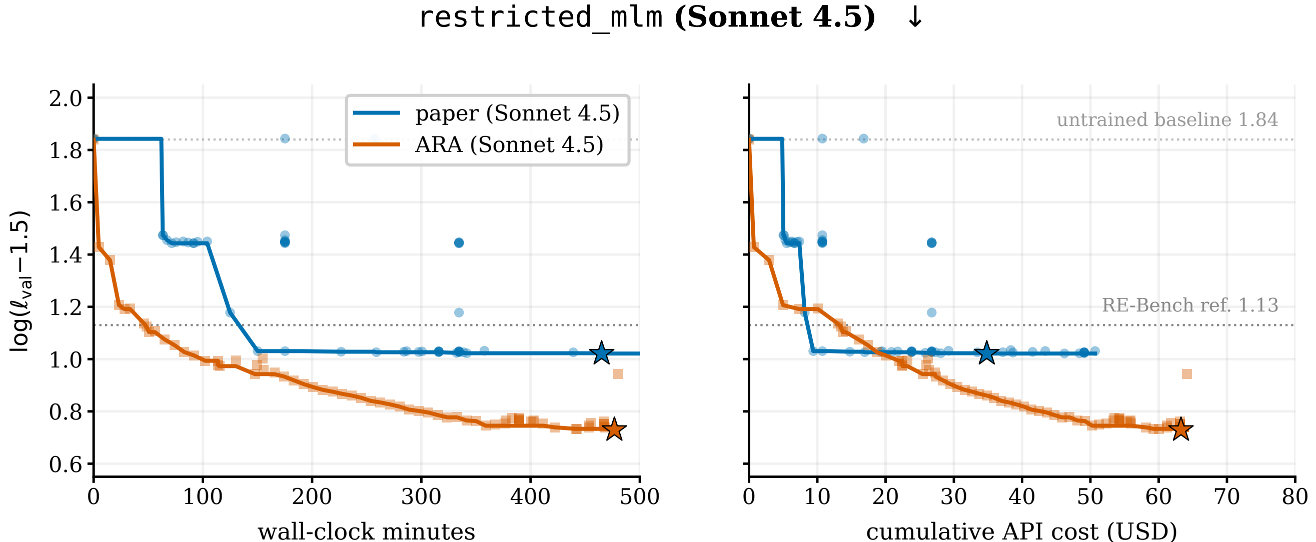


Figure 14. `restricted_mlm` on Sonnet 4.5: paper vs. Ara score-vs-time (left) and score-vs-cost (right). Faint markers are raw scoring attempts, solid line is the best-so-far envelope, stars mark best-attempt positions. Dotted lines mark the untrained-MLP baseline (1.84) and the RE-Bench reference (1.13). Both arms are anchored at the 1.84 baseline at $t = 0$; Ara-4.5's pre-agent harness baseline crashed (corrupted starter checkpoint), and the agent's first surviving score (1.43 at $t \approx 5$ min) reflects the trace-recommended `ConvMLMWithBiBigrams` architecture already swapped in (precomputed bigram tables score ~1.43 with no training), which we anchor at 1.84 to match the other three arms' actual baseline measurements. Ara-4.5 reaches 0.73 vs. paper-4.5's plateau at 1.03 – a ~30% relative win on the weaker base. The 4.6 trajectories are in the body composite.

Why the paper agent wins on Sonnet 4.6. The 4.6 agents diverge architecturally. The paper agent invents `ConvMLMDilated` (a dilated-convolution variant not named in `paper.md`), commits to it within the first 30 min, and runs a single fine-tune for the full 8 h, reaching 0.69. The Ara agent instead implements the additional architectures the heuristics layer names—`H11` ReLU-attention (*"the only attention surrogate any MALT run beat reference with"*), `H07` MLPMixer—and trains them in serial. None outperforms the basic ConvMLM in Sonnet 4.6's loss landscape: `H11` and `H07` were derived from prior MALT runs by Claude-4 Sonnet base, and the menu has gone stale for a successor model whose optimisation differs. The mechanism is the same as in 4.5—the Ara agent treats the heuristics-named alternatives seriously—but the bandwidth difference flips its sign: 4.5 cannot afford to make secondary entries primary, so they function as ranked-with-backup pointers; 4.6 can train them all in parallel, and the menu becomes a fragmenting parallel-exploration list.

I Review System Evaluation

This appendix reports the empirical evaluation of the three Seal levels specified in App. D: compiler-convergence data for Level 1, a 115-mutation benchmark for Level 2, and execution-reproducibility results for Level 3 (which coincide with the Reproduction evaluation of App. G).

I.1 Level 1: Compiler Convergence Data

Level 1 verifies structural correctness and completeness; we report its effectiveness through two reuse signals collected during Ara generation and downstream use.

Compiler iteration counts. Each of the 23 PaperBench Aras and the 7 RE-Bench Aras converges to a Level-1 pass within ≤ 3 iterations of the Compiler's generate–validate–fix loop (§3.2). First-iteration pass rate is 0/30; all artifacts require at least one feedback round, confirming that Level 1 is a non-trivial filter rather than a rubber stamp.

Failure category distribution. Across all Compiler iterations, Level 1 failures break down as follows: dangling cross-layer references (42%), missing schema fields on claims, experiments, or heuristics (31%), insufficient node counts in `exploration_tree.yaml` (14%), YAML or frontmatter parse errors (8%), and missing mandatory files (5%). The distribution is stable across papers and matches the failure taxonomy in Appendix D.3.

Understanding as proof of Level 1 on generated Aras. The Understanding evaluation (§4.1, Table 2) is the end-to-end witness that Level 1 enforces what it claims to enforce on generated artifacts. Every Ara entering that benchmark has passed Level 1; the 95.6% Cat. A accuracy then shows that Level-1-gated Aras carry the structural completeness

an agent needs to retrieve information that is in fact present in the source. An artifact missing a mandatory field or a dangling cross-layer reference would have failed Level 1 and never reached the benchmark, so the 4.4% residual is bounded by information genuinely absent from the source rather than by structural defects of the artifact.

I.2 Level 2: Mutation Benchmark

Setup and evaluation criterion. The Level-2 benchmark stress-tests the Rigor Auditor on *mutated* ARAs, so the reported grade carries no ground-truth signal; we score the auditor strictly on whether it surfaces the seeded defect as a finding. The corpus is the 23 PaperBench ARAs that pass Level 1; each is seeded with one injection per type (115 mutations in total). All injections are recorded in a per-paper `injection_manifest.json` hidden from the auditor.

Injection schema. The five types target distinct schema invariants:

- **Fabricated claim**: append a claim whose `Proof` cites a non-existent experiment ID; signal = dangling reference plus un-grounded substance.
- **Missing falsification**: remove the `Falsification criteria` line from a primary claim; signal = mandatory field absent.
- **Orphan experiment**: append an experiment whose `Verifies` field references a non-existent claim ID (e.g., `C99`); signal = evidence not supporting any claim.
- **Over-claim**: replace a narrow `Statement` with a universal-scope template while leaving the original `Falsification criteria` and `Proof` untouched; signal = scope mismatch between claim breadth and evidence coverage.
- **Rebutted-branch leak**: append a claim advocating an approach that `trace/exploration_tree.yaml` marks `dead_end`; signal = direct contradiction between claim and exploration record.

Auditor and blinding. The Rigor Auditor is an agent skill [2] invoked per artifact and given only the artifact directory; the manifest and source PDF are withheld. It parses claims, experiments, heuristics, gaps, and exploration-tree nodes; builds claim–experiment, claim–dependency, and rejected-node maps; scores six dimensions (D_1 evidence relevance, D_2 falsifiability, D_3 scope calibration, D_4 argument coherence, D_5 exploration integrity, D_6 methodological rigor) on 1–5 anchors; emits findings with severity labels (*critical*, *major*, *minor*, *suggestion*); and reports an overall grade. The full skill specification (prompt, anchors, thresholds) is released with the supplementary code.

Matching and detection rates. Each injection is matched to at most one finding by the rule: a finding hits if (a) its `target_entity` equals the injection's, or (b) its `observation` contains a literal identifier uniquely associated with the injection (e.g., `C99` for orphans, the injected dead-end node ID for rebutted branches). Severity and dimension assignment are ignored when counting hits. Aggregate per-type detection rates are summarised in Table 12; the per-paper × per-injection breakdown is in Table 13. The auditor catches 100% of three high-severity classes (fabricated claims, rebutted-branch leaks, over-claims) and 91% of missing falsifications, but only 22% of orphan experiments. The asymmetry is interpretable: orphans require enumerating every experiment and cross-checking its `Verifies` target against the claim list, whereas the other four surface naturally inside the auditor's per-claim loop. The natural fix is to move orphan detection into Level 1 as a deterministic structural check.

Injection type	Expected severity	n	Detected
Fabricated claim	Critical	23	23 (100%)
Rebutted-branch leak	Critical	23	23 (100%)
Over-claim (scope)	Major	23	23 (100%)
Missing falsification	Major	23	21 (91%)
Orphan experiment	Minor	23	5 (22%)
Overall		**115**	**95 (82.6%)**

Table 12. Rigor Auditor effectiveness on the mutation benchmark (23 ARAs × 5 injection types). The auditor catches all high-severity structural anomalies but exhibits a systematic blind spot on orphan experiments.

Two LLM-as-judge pathologies in the auditor's scoring. Two scoring-side biases emerge. First, *grade inflation*: in 17 of 23 ARAs the auditor's reported overall mean is rounded up just enough to clear the Accept threshold. Second, *finding–score decoupling*: even when the auditor correctly flags an injection as *critical* (22 of 23 rebutted-branch-leak cases), the corresponding dimension score does not drop to the level the rubric prescribes. Both are documented LLM-as-judge failure modes [69], and together they suggest LLMs should generate findings rather than grades, with the overall verdict computed deterministically from the findings list.

Diagnostic: score–finding decoupling. Even though grade is not the evaluation criterion, the auditor's scoring behavior is informative for future iterations. On the 22 ARAs where the rebutted-branch leak is flagged as a *critical* D_5 finding, the auditor still assigns $D_5 \in \{3, 4\}$, despite anchors prescribing 1 ("tree contradicts claims") or 2 ("boilerplate documentation"). Severity in prose does not propagate to the numerical score. The lesson for the next version is mechanical: dimension scores should be derived from the findings list rather than reported independently by the agent.

Paper	Fab.	Miss.fals.	Orphan	Over-cl.	Reb.br.
adaptive-pruning	✓	✓	✓	✓	✓
all-in-one	✓	✓	✓	✓	✓
bam	✓	✗	✓	✓	✓
bbox	✓	✗	✗	✓	✓
bridging-data-gaps	✓	✓	✗	✓	✓
fre	✓	✓	✗	✓	✓
ftrl	✓	✓	✗	✓	✓
lbcs	✓	✓	✗	✓	✓
lca-on-the-line	✓	✓	✗	✓	✓
mechanistic-understanding	✓	✓	✗	✓	✓
pinn	✓	✓	✗	✓	✓
rice	✓	✓	✗	✓	✓
robust-clip	✓	✓	✗	✓	✓
sample-specific-masks	✓	✓	✗	✓	✓
sapg	✓	✓	✓	✓	✓
self-composing-policies	✓	✓	✗	✓	✓
self-expansion	✓	✓	✗	✓	✓
semantic-self-consistency	✓	✓	✗	✓	✓
sequential-neural-score-estimation	✓	✓	✗	✓	✓
stay-on-topic-cfg	✓	✓	✗	✓	✓
stochastic-interpolants	✓	✓	✗	✓	✓
test-time-model-adaptation	✓	✓	✗	✓	✓
what-will-my-model-forget	✓	✓	✗	✓	✓
Total detected (/23)	**23**	**21**	**5**	**23**	**23**

Table 13. Per-paper × per-injection detection for the Level-2 mutation benchmark. ✓ = detected; ✗ = missed. The orphan-experiment column reveals the systematic blind spot discussed above.

I.3 Level 3: Execution Reproducibility

Level 3 effectiveness coincides with the Reproduction evaluation (§4.2, Appendix G): a coding agent reads the Ara and attempts to reproduce claims using the code kernel, with directional verification by LLM-generated test cases. We treat the per-paper difficulty-weighted reproduction score reported in Table 9 as the Level-3 signal, and refer the reader to Appendix G for task design, scoring, and per-paper analysis.

The full operational specification of the three-stage review pipeline (Stages 1–3, advisory diagnostics, comprehensiveness assessment, contestation flow) is in App. D.4.

J Extended Related Work

This appendix expands the condensed body discussion (§K) with the dimensional-coverage table and a thread-by-thread comparison.

The dimensional gap of existing tools. A natural objection is that Ara merely combines documentation, version control, and experiment tracking—three categories of tools researchers already use. Table 14 shows why even using PDFs, GitHub, and trackers (MLflow [67], Weights & Biases [8]) *simultaneously* leaves the knowledge siloed in three unlinked formats with no cross-referencing between claims, the code that tests them, the evidence produced, and the decisions that selected it. Ara closes this gap not by replacing these tools but by providing the missing structural layer: an executable epistemic graph whose cross-layer bindings make these connections explicit and machine-traversable.

	PDF	GitHub	Tracker	Ara
Structured scientific logic	~	~	×	✓
Executable code	×	✓	×	✓
Exploration trajectory	×	×	~	✓
Grounded evidence	~	×	~	✓
Cross-layer bindings	×	×	×	✓

Table 14. Dimensional coverage of existing research artifacts. Each row is a requirement for agent-native research (§1). Existing tools cover at most two dimensions structurally; Ara covers all five with explicit cross-layer bindings. ✓ = full, ~ = partial (present but unstructured or scattered), × = absent.

Machine-readable research artifacts. A growing line of work argues that scientific knowledge should be authored in machine-readable form during the research process rather than recovered post-hoc: the FAIR principles [62] standardize data metadata, the W3C PROV ontology [30] formalizes

provenance for scientific outputs, Canini [12] reframes the paper as a "compression format for human readers" that should yield to structured knowledge objects, and Booeshaghi et al. [10], Stocker et al. [55] advocate authoring-time machine readability as a first principle. Concrete formats instantiate parts of this vision—nanopublications [20] atomize claims with provenance, the Open Research Knowledge Graph [23] curates structured contributions across papers, RO-Crate [53] bundles research objects, Whole Tale [11] packages computational environments, and the Discovery Engine [6] distills publications into a Conceptual Tensor—but none provides execution semantics or captures decision history. Unlike these formats, Ara jointly binds scientific logic, minimal executable code, and decision history into a single protocol with machine-verifiable reproducibility.

Reproducibility infrastructure. The reproducibility crisis in ML [5, 42] has motivated code-sharing standards [56], scientific workflow engines [15, 16, 27], and computational notebooks [25, 50]; yet workflows encode pipelines without claim semantics, notebooks remain documents with hidden state, and recent benchmarks [26, 33, 54] collectively show that frontier agents cannot recover knowledge PDFs leave implicit—EXP-Bench reports only 0.5% end-to-end experiment success despite 20–35% component accuracy. On the verification side, LLMs detect fewer than 46% of paper–code discrepancies [7], extending a longer line of scientific claim verification [58] and attribution-based grounding [18] and motivating formal auditing criteria across provenance, soundness, claim decomposition, and cryptographic lineage [22, 46, 47]. Unlike prior auditing proposals that address a single dimension, Ara's Seal Certificates operationalize all of them in one enforceable mechanism.

Negative knowledge and failed trajectories. Recent work shows that failure traces become actionable only once annotated with root-cause structure [68, 70], yet raw trajectory dumps [65] remain difficult to leverage. Large-scale experiment logs [19, 43, 66] retain >99.99% more search history than their corresponding papers report, and process-level studies [61, 64] confirm that human experts and agentic scientists both explore extensive dead ends that never surface in the write-up. Unlike raw trajectory archives, Ara's exploration graph promotes dead ends to first-class `dead_end` nodes with structured failure modes and claim cross-references, making negative knowledge machine-queryable rather than lost to narrative selection.

Agent-oriented documentation and tooling. A convergent body of work shows that agents benefit from structured, layered representations [41, 57] over flat corpora [35, 45], and that even the strongest LLMs implement fewer than 40% of novel contributions correctly, with *semantic* misalignment as the dominant failure mode [14, 21, 24]. Recent systems target this gap from three sides: pipelines that generate code from papers post-hoc [52] or recover tacit knowledge through graph analysis and debugging [31]; knowledge-graph approaches that mine background literature for technique–code links [34, 37]—yielding up to 10.9% PaperBench gains but leaving the target contribution's decision history and epistemic structure unmodeled; and autonomous research agents that conduct experiments end-to-end [4, 9, 38, 51], whose unstructured trajectory logs are themselves discarded once the resulting paper is written. Multi-agent frameworks [63], skill-library standards [2, 60], and artifact-mediated agent coordination [59] further show that structured artifacts, not natural-language papers, are the natural unit of exchange for compounding agent capability—a premise the Ara ecosystem (§3) instantiates through its live-capture, compilation, and review skills. Unlike post-hoc recovery pipelines and background-knowledge graphs, Ara encodes claims, evidence, heuristics, and their executable bindings at authoring time, eliminating the recovery step entirely.

K Related Work

Ara synthesizes ideas from three threads: *machine-readable science* [6, 10–12, 20, 23, 30, 53, 55, 62], which structures published knowledge but provides no execution semantics or decision history; *reproducibility infrastructure* [5, 7, 15, 16, 18, 22, 26, 27, 33, 42, 46, 47, 50, 54, 56, 58], which encodes pipelines and verification criteria but not claim-level epistemic structure; and *agent-oriented tooling* [2, 4, 9, 14, 21, 24, 31, 34, 37, 38, 41, 51, 52, 57, 59, 60, 63], which recovers structure post-hoc or coordinates agents over unstructured artifacts. Adjacent work on *negative knowledge* [19, 43, 61, 64–66, 68, 70] shows that failure traces are actionable only once annotated with structure that raw trajectory dumps lack.

A natural objection is that Ara merely combines documentation, version control, and experiment trackers [8, 67]; Appendix J (Table 14) shows that even using these tools simultaneously leaves five required dimensions (structured logic, executable code, exploration trajectory, grounded evidence, and cross-layer bindings) only partially covered, with no cross-references linking them. Unlike post-hoc recovery pipelines, background-knowledge graphs, and raw trajectory archives, Ara jointly binds scientific logic, executable code, decision history, and grounded evidence at authoring time into a single protocol with machine-verifiable reproducibility, and operationalizes the auditing criteria scattered across prior verification proposals as a single Seal Certificate. See Appendix J for a full thread-by-thread comparison.

L Limitations

Evaluation scope. Our study covers only machine learning papers, where computational reproducibility and well-defined contribution types fit Ara's four-layer structure naturally. Generalization to experimental sciences with physical execution requirements, or to theoretical disciplines where the Physical Layer is largely absent, remains untested. Extending the Physical Layer to proof-based results with machine-checkable specifications is a natural next step.

Fidelity ceiling. Ara fidelity is bounded by its source. The Compiler faithfully represents only what the PDF contains (§3.2); when papers omit experimental details or ablations, no extraction can recover them. The Live Research Manager closes this gap by recording trajectories as research unfolds, but assumes an AI-native workflow with a coding agent present throughout. Closing this adoption gap tracks the broader diffusion of agent workflows in research practice, not the protocol itself.

Deployment prerequisites. The adversarial robustness and privacy guarantees raised in §3.3 are aspirational: the system lacks sandboxed execution, anomaly detection, and granular access control for the Exploration Graph. Separately, schema evolution remains open: we version `PAPER.md` frontmatter with an `ara_schema` tag and require validators to accept unknown fields and degrade gracefully on missing ones, but have only exercised this across minor revisions. A stable migration story for major revisions, including automatic rewriting of archival artifacts, remains future work.

M Discussion and Future Work

Trace contextualisation across model generations. Our extension trajectories (§4.3) show that an Ara accelerates a follow-up agent by surfacing prior pitfalls and successful strategies, but the same trace can constrain a stronger base whose own bandwidth exceeds the documented playbook. Selectively hiding or contextualising parts of the trace, for example by marking trace nodes with model-class provenance so successors can discount claims that no longer apply, is one mechanism for closing this gap; we leave the broader design space to follow-up work.

Near term: artifact lineage and self-maintaining ecosystems. The most pressing near-term gap is artifact durability: like code repositories, Aras decay without maintenance as dependencies rot and practices evolve, yet unmaintained artifacts are the community norm. The natural extension is a *lineage mechanism* in which each Ara declares its parent artifacts and expresses its contribution as a structured diff, reducing both construction cost (authors specify only the delta) and verification cost (reviewers and agents re-check only the new contribution). Lineage also enables self-maintaining ecosystems: agents consuming an Ara detect and repair staleness, update deprecated dependencies, and propagate corrections upstream, so that every act of consumption becomes an act of maintenance.

Medium term: knowledge graph, collaborative discovery, and continuous review. Aggregated lineages form a queryable scientific knowledge graph that lifts collaboration and review from the document level to the corpus level. Cross-artifact claim alignment turns literature synthesis into subgraph queries, lets reviewer agents verify that reported baselines match what cited Aras recorded, and exposes trajectory conflicts where a method claimed as successful elsewhere was documented as failing. Shared Exploration Graphs also enable collaboration formats impossible in the PDF ecosystem, from parallel continuation of open problems with documented dead ends to fine-grained attribution on live, evolving artifacts. Review evolves in parallel: there is no single accept moment, only a claim-confidence surface that rises with replications and falls with counter-evidence, freeing human expert attention for the judgments only humans make, namely novelty, significance, and taste.

Long term: cross-disciplinary collective memory. Our evaluation is restricted to machine learning, where Ara's four-layer structure aligns naturally with the dominant contribution types: algorithms, architectures, and training procedures. Whether this structure generalizes to other disciplines remains an open question. The Cognitive and Evidence Layers are plausibly domain-agnostic, but the Physical Layer and Exploration Graph, both premised on iterable computational experiments, may require substantial adaptation for wet-lab sciences where execution is physical rather than computational. If these adaptations succeed, Ara provides a natural substrate for cross-disciplinary knowledge transfer, where documented failures in one field become actionable knowledge in another via graph traversal rather than literature search in unfamiliar notation.