ARI: An Autonomous Research Infrastructure

Takanori Kotama

v0.9.0, 2026-06-12

Abstract

We describe **ARI**, an autonomous research infrastructure that couples a tree-search-based exploration loop with a paper-generation pipeline. The exploration loop is a best-first tree search (BFTS) over research-step nodes; expansion uses a Reason-and-Act (ReAct [1])-style agent that issues tool calls into a registry of fourteen domain skills. The paper pipeline turns the surviving lineage into a draft, then iteratively refines it under a hierarchical rubric. Every artefact lives inside a single checkpoint directory, which is the unit of reproducibility, lineage, and cost accounting. This report formalises the algorithms and the design rationale, reports preliminary observations, and discusses the determinism / novelty trade-off that constrains the design.

1 Introduction

1.1 Motivation

Research workflows in machine learning and adjacent computational fields share a common skeleton: brainstorm a research idea, design and run an experiment, analyse the result, write a paper, and iterate. Each step is increasingly tractable for large language models (LLMs), and published systems now automate substantial spans of this loop. The AI Scientist [2, 3] closes the full idea-to-paper cycle and scores the result with an automated reviewer; Agent Laboratory [4] executes a human-provided idea under LLM-judged gates; AIDE [5] casts the engineering inner loop as tree search over candidate solutions, the formulation MLE-bench [6] found to dominate general-purpose scaffolds given the same model; VirSci [7] shows that multi-agent deliberation produces more novel research ideas than a single agent, stopping at the abstract; Story2Proposal [8] converts finished experimental evidence into a manuscript under a persistent structural contract; and PaperBench [9] measures, post hoc, how faithfully an agent replicates a published paper.

What none of these systems makes binding is the relation between what the generated paper *says* and what its experiments *produced*. Verification, where present, is advisory: the AI Scientist's reviewer scores finished drafts while hallucination control stays at the prompt level — its authors document misreported hardware and a worsened metric presented as an improvement, and name "a more in-depth automatic verification of the reported results" as key future work [2]; Agent Laboratory's simulated reviewers overestimate human reviewer scores by 2.3 points on a ten-point scale [4]; and PaperBench's central finding is an execution gap — agents earn substantial credit for written code, almost none for executing it and matching the paper's results [9].

ARI exists to close this gap. The user supplies a research question and a compute budget; the system returns a publishable paper, the experimental code behind every claim, and a complete audit trail: every node of the search tree, every tool call, every model invocation, and every dollar of API spend is recorded inside a single *checkpoint directory*, the unit of reproducibility.

1.2 Verified claims as a release condition

ARI's distinguishing thread runs through the whole pipeline. At idea time the run mints a *metric contract*: the falsifiable claims the eventual paper must support, each naming the exact measurement names that count as evidence. The contract is minted once and frozen, because LLM naming does not survive regeneration (section 4.3). During exploration, nodes emit measurements under those names, the names steer expansion toward uncovered claims, and *lineage chaining* lets evidence *computed* from earlier measurements — a fit, its held-out validation, a model-based selection — execute along one lineage instead of regressing to repeated probes (section 3.8). At writing time, every numeric mention is bound to the node and measurement it rests on, and a deterministic claim–evidence gate — no language model in the loop — re-derives each reported number from the recorded results. Objective falsehoods block release at the final gate; the subjective findings of a complementary semantic review only advise the refine loop.

1.3 Contributions

This report formalises four contributions:

1. An **idea-owned metric contract** enforced by a **deterministic claim–evidence gate** (section 4.3): verification as a structural property of the pipeline, not a reviewer's opinion of its output.

2. A **best-first tree search** (BFTS) over research-step nodes (section 3.1) whose selection and expansion are LLM-driven over structured candidate descriptions, with a deterministic fallback ranking (section 3.3).

3. A **paper-generation pipeline** keyed off a hierarchical rubric R whose leaf judges grade individual claims; the system score is $\text{score}(P) = \sum_i w_i \, \text{judge}_i(P)$ (section 4.4). The PaperBench replicator is integrated as a tool call (section 4.6).

4. A **checkpoint scoping** discipline: every external state (memory, cost ledger, lineage pointers) is written below `$ARI_CHECKPOINT_DIR`, never the user's home directory. This is the operational consequence of the *determinism* design principle, one of five that section 2.5 lays out in full (we refer to it as P2 there and in the rest of the report).

1.4 Scope of this report

This report describes `ari-core` and the fourteen `ari-skill-*` packages at version v0.9.0, whose theme is end-to-end claim verification: the mint-once contract and gate hardening of section 4.3; lineage chaining (section 3.8); an opt-in ideation path that drives the published VirSci deliberation engine unmodified over a live literature snapshot (section 6.2); platform-aware ideation, folding probed platform-capability facts into idea generation so a plan cannot promise measurements the execution platform does not provide; and a released sample paper that is itself gate-verified, all twelve of its declared falsifiable claims carrying machine-verified evidence (section 5). These additions build on the v0.8.x foundations — the protocol-faithful three-stage PaperBench reproduction contract (agent rollout, reproduction, grading), host-truthful environment guardrails, and configurable search-evaluation layers; the operational reproduction pass on a lossy-compression target reported in section 5 ran on the then-current v0.8.0 pipeline. The body describes the algorithms and data flow at the module level; the verbatim LLM prompts the system uses are listed in section B. The empirical study is preliminary at this version, and a controlled benchmark is left to future work.

1.5 Organisation

Section 2 gives a top-down view of ARI: the orchestrator, the fourteen skills, the pipeline DAG, and the design principles. Sections 3 and 4 are the two technical cores, each with its own formal definitions; the claim-verification thread of section 1.2 is developed across both. Section 5 reports preliminary observations; section 6 positions the work; section 7 collects open problems.

2 System Overview

2.1 The full stack

Figure 1 draws the full ARI stack at v0.9.0 as five layers; this section walks them top to bottom. The system splits into one driver and many tool providers: the driver (`ari-core`) owns the orchestrator, the search and agent engines, the lineage walker, the checkpoint serialiser, the cost tracker, and the configuration loader, while every domain capability — writing code, running benchmarks, reviewing drafts — lives in a skill.

Entry points. A run starts from the command-line interface (CLI), from the public Python interface (the software development kit, SDK), or from the visualisation server — an HTTP dashboard exposing live search state, per-checkpoint files, and launch controls to a browser. All paths converge on the same orchestrator entry (badge 1); interactive use does not get a second code path.

Engine layer. The *best-first tree search* (BFTS) engine maintains the tree of research-step nodes and decides which node to run next and how to expand it (section 3) — a search-over-candidate-solutions framing ARI shares with AIDE [5]. Each expansion hands a node to the `AgentLoop`, a Reason-and-Act (ReAct) agent [1] that interleaves free-text reasoning with tool calls until the node's work is done: the engine supplies direction, the agent execution — one iteration loop (badge 2). The lineage walker beside the loop is consulted rather than driven: it climbs parent pointers across checkpoints so a derivative run can inherit its ancestors' idea pool and context (section 3.7).

MCP dispatch. Every tool call crosses the dispatch layer (badge 3). The *Model Context Protocol* (MCP) is the JSON-RPC-over-stdio interface that Anthropic standardised for exposing tool-style capabilities to a language-model agent loop; ARI uses it as the universal seam between `ari-core` and every skill. Each skill is a separate Python package whose server runs as a pooled child process, under the skill's own interpreter when it ships one. A skill is wired in by a configuration entry naming its package path and the pipeline phases in which its tools are exposed; the tool manager filters tools by the active phase, so an exploration node cannot call the paper-writing tools (section 3.6).

The fourteen skill packages currently in tree are: `benchmark`, `coding`, `evaluator`, `hpc`, `idea`, `memory`, `orchestrator`, `paper`, `paper-re`, `plot`, `replicate`, `transform`, `vlm`, and `web`. The registry in fig. 1 groups their tool surfaces by lifecycle phase: generation (idea, coding), evaluation (evaluator, review), writing & replication (paper, replicate, paper-re), and infrastructure and I-O — where the `hpc` skill is the seam to a cluster batch scheduler, so experiments can run as dispatched jobs. Two boxes deserve a note: *review*

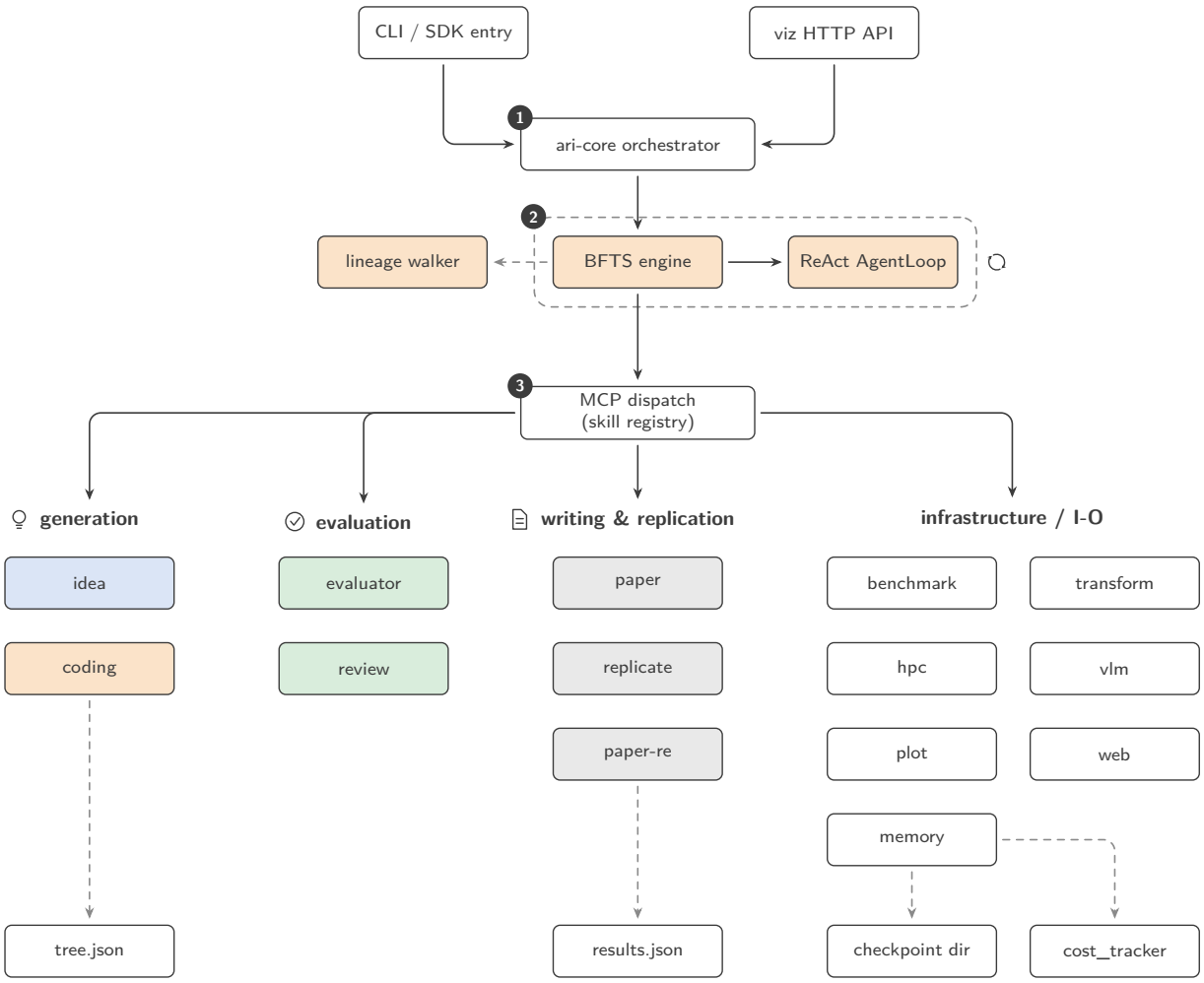

Figure 1: The full ARI stack at v0.9.0 as five layers: entry points; the `ari-core` orchestrator (badge 1); the engine layer, where BFTS engine and ReAct `AgentLoop` form the iteration loop (badge 2) with the lineage walker beside it; the MCP dispatch hub (badge 3) fanning into the skill registry's four lifecycle columns; and checkpoint persistence. Solid edges are control; dashed edges are secondary flows (lineage lookup, persistence).

names a tool surface rather than a package (draft-review tools are hosted by the evaluator and paper skills; section 4.5), and the `orchestrator` skill exposes run management over MCP, so ARI itself can be driven as a tool by an external agent.

The orchestrator dispatches calls but performs no domain work itself. The separation matters because skills can be added or swapped without touching the search algorithm: the BFTS engine is unaware of which skills participate.

The bottom layer is persistence: every byte a run produces lands in a single checkpoint directory, alongside the serialised tree and the cost ledger (section 2.5). The smaller diagram in fig. 2 is the "operational" view used in the rest of the report (control buses and state edges only); fig. 1 is the "layered" view (driver → engines → MCP → skills → persistence).

2.2 Pipeline DAG

A run reduces to a four-station pipeline (fig. 3). The idea station selects or generates research questions; the exploration station spawns nodes under BFTS; the paper station compiles the best lineage into a draft; and the review station grades the draft against the rubric. The graph is a DAG plus one back edge from review to paper: only this edge can introduce non-monotonicity (refining can worsen an axis), and the *convergence criterion* (section 4.5) bounds the back-edge count. The pipeline is driven by a single runner in the orchestrator; a scientific-data assembler collates the per-node artefacts the writer needs (section 4.2).

The write-and-review band hides a fixed *gate chain* the draft must clear before finalisation. After writing, the draft's claims are linked to recorded evidence by stable identifiers; a deterministic claim–evidence gate (no language model) re-derives every reported number from the recorded results within a tolerance; a complementary semantic review flags overclaims and feeds the refine loop; and after refinement the linking, the review, and the gate run once more as a final check that, under the strict policy, blocks finalisation while any objective mismatch stands (section 4.3). The blocking gate is core-side code reached through a thin skill wrapper, so swapping a skill cannot weaken it; blocking is reserved for objective falsehoods, while subjective review findings steer the refiner but never gate publication. The chain realises Story2Proposal's

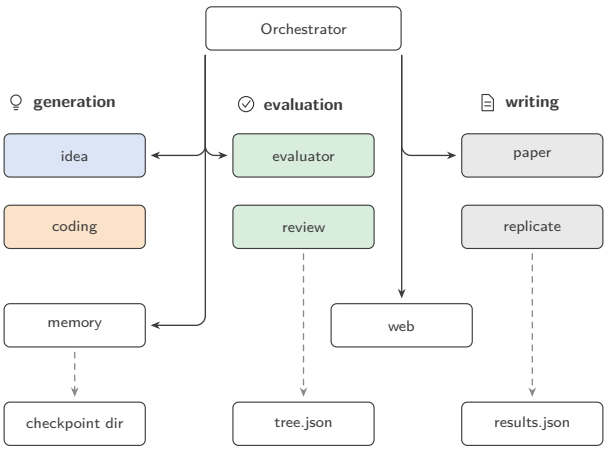

Figure 2: Operational view. The orchestrator fans out over two dispatch buses into three lifecycle columns (fill encodes phase) plus the phase-free support skills. Dashed edges are state: each producing group persists into the checkpoint artefacts on the bottom row.

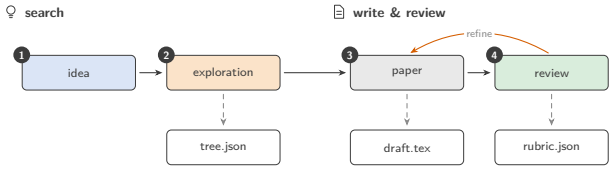

Figure 3: Pipeline DAG: four stations (badges 1–4) under two band headings, *search* and *write & review*. Each station persists its artefact to the checkpoint (dashed). The single accented edge is *refine*, the back edge that rewrites the draft and re-runs the review.

contract-threaded generation in execution-grounded form [8].

2.3 BFTS control flow

Figure 4 traces a single BFTS step end-to-end. The implementation is laid out in section 3; the diagram fixes the vocabulary used by the rest of the report (frontier, selection, fallback, expand, prune, stagnation) and shows the prompt files behind the two LLM-driven stations (`bfts_select.md`, `bfts_expand.md`); a third prompt, `bfts_expand_select.md`, drives the choice of *which* completed node to expand in the full loop (section 3.3). The two hint boxes below the spine are new at v0.9.0 and implement lineage chaining for computed evidence: the coverage hint steers the expansion target toward the node already holding the required measurements, and the lineage hint tells the new child which inherited measurement files it holds. The hints carry only names, never values, so branch isolation is preserved (section 3.8).

2.4 paper-re component view

Figure 5 is the second-level view of `paper-re`, the skill that carries the PaperBench-protocol reproduction path [9]. ARI's contribution at this layer is small but specific: a thin adapter wraps each agent tool so its output is size-bounded, and redirects the vendored solver's hard-coded paths to ARI's per-node workspace; everything else flows through the PaperBench upstream verbatim, which keeps ARI aligned with the upstream replication-task framing and grade semantics. Section 4.6 develops the three-stage rollout–reproduce–grade contract the bridge drives.

2.5 Checkpoint scoping and the design principles

ARI is built around five *design principles* (P1–P5). The binding one is **P2: determinism**. Every randomised step records its seed; every LLM call records its model identity, prompt, and output; every tool call is tagged with its node identity (a copy-on-write guard) so its writes land inside the right node's working tree. Re-running a checkpoint must reproduce the same artefacts byte-for-byte, except for fields explicitly marked as wallclock or as model-side non-determinism.

The other principles, in summary:

- **P1 single artefact tree**: every output of a run lives below `$ARI_CHECKPOINT_DIR`. Nothing leaks to `$HOME` or `/tmp`.
- **P3 small components**: each skill is a separate package; replacing one skill does not require rebuilding `ari-core`.
- **P4 explicit lineage**: every node and every checkpoint record their parent, so a derivative experiment can recompute only its delta.
- **P5 cost transparency**: the cost tracker attaches dollar costs to every model call so the user can bound a run; skill subprocesses append to the same run-level ledger.

The checkpoint serialiser writes three top-level files: the full search tree to `tree.json`, a lightweight per-node export consumed by the paper pipeline to `nodes_tree.json`, and end-of-run aggregates to `results.json`. Lineage walks across checkpoints climb from a child to its ancestors through the parent pointer each checkpoint records in its run metadata.

Figure 6 draws the on-disk shape of a checkpoint: shared run-level files on the left, per-node working trees on the right. A node's working tree starts as a copy of its parent's, so each tree is a self-contained snapshot of the experiment at that step, and the self-report that grounds evidence assembly (section 4.2) sits next to the artefacts it describes. The shape is the contract: any tool that takes ARI's output as input — a follow-up run, a regression check, a paper draft — points at a single such directory.

3 Exploration Algorithm

The exploration phase is the experimental core of a run: given a selected idea and the metric contract it owns — the run-level declaration of falsifiable claims,

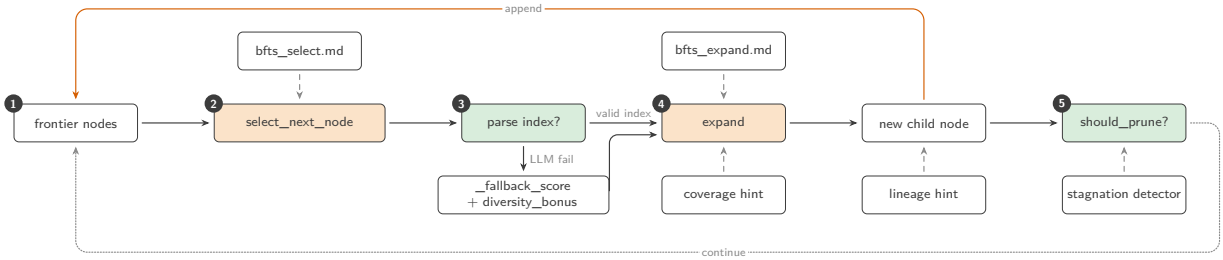

Figure 4: BFTS control flow: one step as a spine of five numbered stations. The two LLM-driven stations each consult their own prompt file (dashed, above); when the parse gate rejects the returned index, selection falls back to the deterministic `_fallback_score` (scientific score plus `diversity_bonus`). Below the spine: the v0.9.0 lineage-chaining hints into the expansion, and the stagnation detector into the prune gate. The accented edge appends the child to the frontier; the dotted margin path continues to the next iteration.

each naming the exact measurement names that count as its evidence (section 4.3) — exploration must turn a bounded node budget into measurements that make those claims evaluable. ARI organises the phase as a best-first tree search (BFTS) over research steps. The skeleton descends from AIDE, which framed LLM-driven engineering as search over a tree of candidate solutions under a hard-coded draft/debug/improve policy and a single scalar objective [5]; ARI replaces the policy with LLM-driven selection backed by deterministic fallbacks (section 3.3), steers expansion with contract-derived coverage and lineage hints (sections 3.4 and 3.8) — a structural answer to the local-optima plateau reported for greedy single-objective policies [5] — and executes each node as a guarded Reason-and-Act (ReAct) agent whose obligations and feedback come from the same contract (section 3.6). The closing subsections cover the substrate the search runs on: the platform probe (section 3.9), the research memory (section 3.10), and the ideation stage that seeds the root (section 3.11).

3.1 Formalisation

The exploration loop maintains a tree $\mathcal{T} = (\mathcal{N}, E)$ in which each node $n \in \mathcal{N}$ represents a single research step: an idea revision, a code change, a benchmark run, an analysis, etc. A node carries

- a discrete state $s(n) \in \{\text{open}, \text{eval}, \text{done}, \text{failed}\}$,
- a categorical *label* drawn from the `NodeLabel` enumeration,
- per-axis evaluation metrics in a freeform dict `node.metrics`, including a distinguished scalar `_scientific_score` $\in [0, 1]$,
- a Boolean `has_real_data` that flags whether the node carries actual measurements rather than provisional text, and
- the bookkeeping field `depth` surfaced both in the selection prompts and to the pruning predicate. ARI does not retry failed nodes, so no `retry_count` signal is surfaced.

Figure 7 shows a small tree of this kind. The aggregation of axis-level signals into the scientific score is delegated to any implementation of the `Evaluator` protocol; ARI does not pin a fixed linear combination. The protocol is the only seam between BFTS and downstream rubric evaluation, which keeps the search engine agnostic about how axes get reduced to a scalar.

Run-loop state. Each iteration of the outer loop transforms the tuple

$$S \;=\; (\mathcal{F},\; \mathcal{P},\; e,\; H),$$

where $\mathcal{F} \subseteq \mathcal{N}$ holds completed nodes (done / failed) eligible for expansion, $\mathcal{P} \subseteq \mathcal{N}$ holds nodes in state open ready to execute, $e : \mathcal{N} \to \mathbb{N}$ counts how many times each frontier node has been expanded, and $H = (l_1, \ldots, l_t)$ is the sliding window of recently-executed labels (length capped at $W = 20$, see section 3.3). The initial state is $S_0 = (\emptyset, \{n_{\text{root}}\}, \mathbf{0}, ())$.

Pruning predicate. The hard exploration budget is the predicate

$$\Pi(n;\, |\mathcal{N}|) \;=\; \mathbb{1}[|\mathcal{N}| \geq B_N] \;\vee\; \mathbb{1}[\text{depth}(n) \geq B_D] \;\vee\; \sigma(n), \quad (1)$$

where B_N = `config.max_total_nodes`, B_D = `config.max_depth`, and the *sterile* predicate

$$\sigma(n) \;=\; \mathbb{1}[\,|\Delta_n| = 0\,] \quad (2)$$

(with Δ_n the set of files added, modified, or deleted by n) fires when a node finishes its agent loop without writing any new artefact relative to its parent. The run loop computes this file diff and records the result as a Boolean `_sterile` flag on the node; `should_prune` reads that flag together with the two caps. The caller passes the live $|\mathcal{N}|$ each iteration so there is exactly one source of truth for the node count. Step / cost budgets are enforced separately by the cost tracker at the call site, not inside the BFTS engine.

Retire predicate. On top of Π, the run-loop applies a softer *frontier retire predicate* ρ to drop a parent f

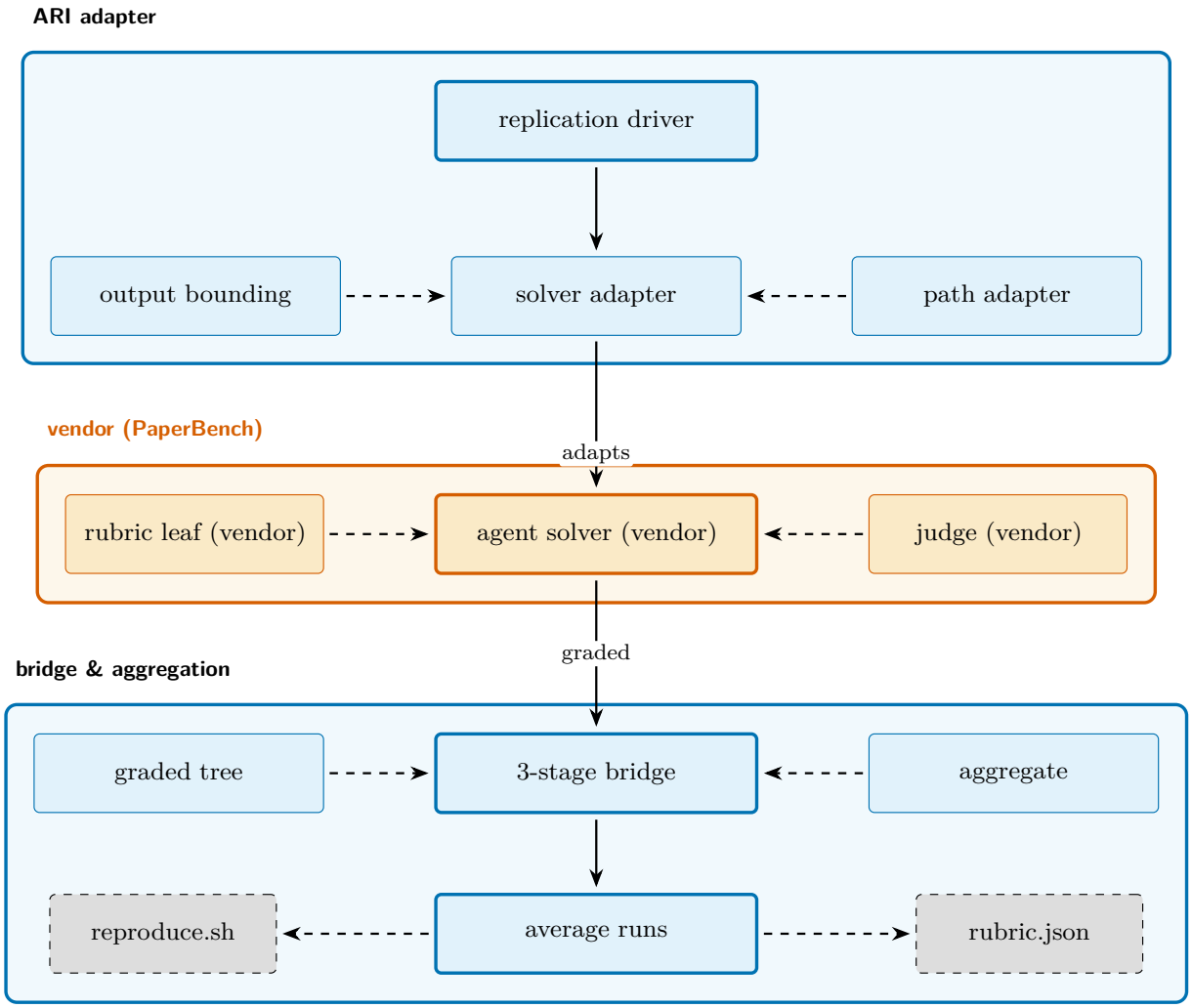

Figure 5: The `paper-re` skill's components in three layers. The top row is ARI's adapter layer (replication driver, solver adapter, output bounding, path adapter); the middle row is the vendored PaperBench package (agent solver, rubric leaves, judge); the bottom row is the bridge that drives the three protocol stages and collapses a per-submission graded tree — averaging repeated runs — into a single `rubric.json` score plus an ARI-side `reproduce.sh`. Detail in section 4.6.

from $\mathcal{F}$ once it is no longer the most productive lead:

$$\rho(f, c) = \underbrace{\not\Vdash[r(c) > r(f)]}_{\text{Rule A}} \vee \underbrace{\not\Vdash[e(f) \geq E_{\max}]}_{\text{Rule B}}, \quad (3)$$

with $E_{\max}$ = `config.max_expansions_per_node` (default 4). Rule A retires f as soon as a child c outscores it; Rule B caps the per-parent budget so the search spreads. ρ does *not* delete the node — its history stays on $\mathcal{T}$ — only its eligibility to be re-expanded is revoked.

The full step decomposition of one outer iteration (inner expansion sub-loop, parallel ReAct batch, post-execution retire) is shown in fig. 8; the explicit pseudocode is algorithm 1 in section 3.2.

3.2 Run-loop algorithm

Algorithm 1 writes the iteration as explicit pseudocode; fig. 4 renders the same step as a control-flow spine with the prompt files drawn in. The inner WHILE fills empty worker slots one expansion at a time (so workers always start running before the next expansion is proposed); the outer block then assembles a batch by calling `select_next_node` repeatedly — one node per call — until the batch reaches the worker cap, and executes that batch in parallel; the post-execution block (a) records the executed label into H so the diversity bonus reflects what actually ran, and (b) applies ρ Rules A and B.

3.3 Node selection (LLM-driven)

ARI deliberately does *not* reduce node ranking to a closed-form scalar utility — the design choice that most separates its BFTS from AIDE's hard-coded greedy policy [5]. Instead, both `select_next_node` ("which pending node should the next agent step run?") and `select_best_to_expand` ("which completed node is most worth expanding?") hand a list of candidate descriptions to the LLM and parse back a single 0-based index — each call selects exactly one node. The candidate description for each node n (fig. 9) is a one-liner of the form

```
[i] id=..., depth=..., label=...,
```

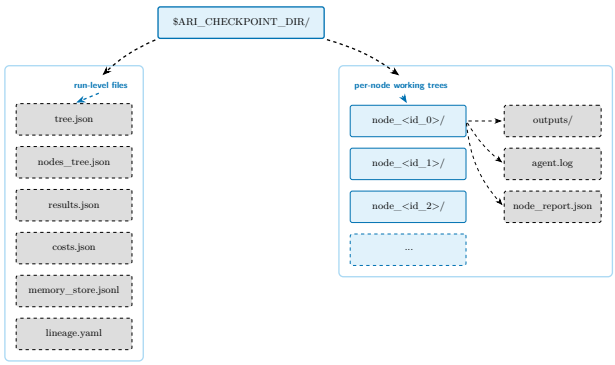

Figure 6: Checkpoint directory layout. Run-level files (left band) are written by the orchestrator; per-node working trees (right band) hold the artefacts produced inside one BFTS expansion, including the structured self-report each node leaves behind. Reproducing a run is exactly "re-execute against this directory", and the layout guarantees that no other state is needed.

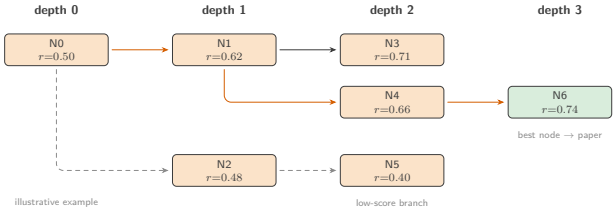

Figure 7: A small BFTS exploration tree (synthetic, illustrative). Columns are search depths; each box is one node with its scalar reward $r(\cdot)$. Accented edges trace the best node's lineage back to the root — the chain the paper phase consumes (section 4.2); dashed edges mark a low-score branch the selection policy stops picking, whose history stays on the tree.

```
has_real_data=..., metrics={...},
summary=...
```

The run-selection variant (`select_next_node`) additionally appends `diversity_bonus=+0.05` to the line for any node that currently earns the bonus; `select_best_to_expand` omits it. The prompts that consume the list are reproduced verbatim as section B.3 (selection) and section B.2 (expansion target). The selection criteria the prompt enumerates are: nodes with `has_real_data=True` and strong metrics are high-value; metrics are weighed holistically (multi-objective); deeper nodes with excellent results are worth continuing; the `diversity_bonus` is treated as a soft tiebreaker only. The `label` field matches the information shown to `select_best_to_expand`, and there is no spurious "low retry" criterion.

Diversity bonus

The diversity bonus `BFTS.diversity_bonus` tracks recent labels in `_recent_label_history` (a sliding window populated by `record_run`). For a node whose label is *under*- represented relative to the most common label in that window, the function returns

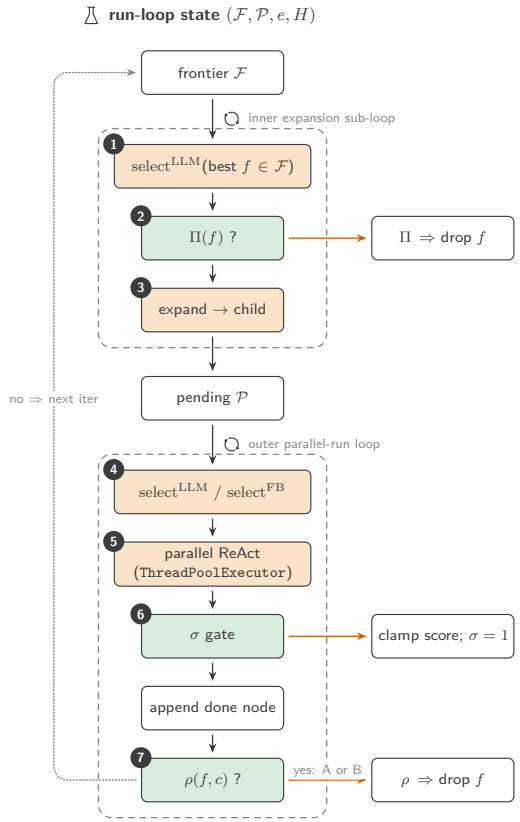

Figure 8: State-machine view of one outer BFTS iteration as a single top-to-bottom flowchart. Steps 1–3 are the inner expansion sub-loop; steps 4–7 are the outer parallel-run loop. Yellow diamonds are the predicate guards (Π, σ, ρ); red dashed boxes on the right are the side-effects each predicate triggers (frontier drop / score clamp / retire). Auxiliary state not drawn here: the per-parent expansion counter $e(\cdot)$ is bumped at step 3 and consulted by ρ at step 7, and the label history H (window W) is appended at step 5 via `record_run` and consulted at step 4 to compute the diversity bonus $\text{div}(\cdot)$. Compare with fig. 4, which shows the same loop as a finer-grained control-flow spine with the LLM prompts drawn in.

$$\text{div}(n) = \begin{cases} +0.05 & \text{cnt}(n) \le \tfrac{1}{2}\,\text{cnt}_{\max}, \\ 0 & \text{otherwise}, \end{cases} \tag{4}$$

where $\text{cnt}(n)$ is the number of times $\text{label}(n)$ occurs in the sliding window and $\text{cnt}_{\max} = \max_{\ell} \text{cnt}(\ell)$. The constant 0.05 is small by design: scientific score still dominates ranking, and the bonus only breaks near-ties.

LLM fallback

If the LLM returns an unparseable index, `select_next_node` falls back to a deterministic ranking via the `_select_fallback` helper (which ranks candidates by the `_fallback_score` expression). The default scoring expression is

$$\text{fb}(n) \;=\; r(n) + \text{div}(n), \tag{5}$$

where $r(n) \equiv$ `_scientific_score`(n) is the node's scalar reward (section A). `_select_fallback` prefers

Algorithm 1 BFTS run-loop (one outer iteration).

Input: state $S = (\mathcal{F}, \mathcal{P}, e, H)$; config $(B_N, B_D, E_{\max})$ and worker cap `max_parallel` $= \min(\texttt{max\_parallel\_nodes}, 4)$
Output: updated state S'

▷ — inner expansion sub-loop —
while $\mathcal{F} \neq \emptyset \wedge |\mathcal{P}| < \texttt{max\_parallel} \wedge |\mathcal{N}| < B_N$ **do**
  $f \leftarrow \text{select}^{\text{LLM}}(\mathcal{F})$ ▷ `select_best_to_expand`
  if $\Pi(f; |\mathcal{N}|)$ **then**
    $\mathcal{F} \leftarrow \mathcal{F} \setminus \{f\}$; **continue**
  end if
  $c \leftarrow \texttt{expand}(f, \texttt{depth\_note}, \texttt{budget\_note})$
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{c\}$; $e(f) \mathrel{+}= 1$
end while
▷ — outer parallel-execution batch —
$B \leftarrow \emptyset$
while $|B| < \min(\texttt{max\_parallel}, |\mathcal{P}|)$ **do**
  $n \leftarrow \text{select}^{\text{LLM}}(\mathcal{P})$ ▷ `select_next_node`: one node
  $\mathcal{P} \leftarrow \mathcal{P} \setminus \{n\}$; $B \leftarrow B \cup \{n\}$
end while
for all $n \in B$ **in parallel do**
  $n' \leftarrow \text{ReAct}(n)$ ▷ `AgentLoop.run`, may fail
  $H \leftarrow (H \parallel \text{label}(n'))[-W:]$ ▷ `record_run`
  $\mathcal{F} \leftarrow \mathcal{F} \cup \{n'\}$
  for all $p \in \mathcal{F} : p = \text{pa}(n')$ **do** ▷ Rule A
    if $r(n') > r(p)$ **then**
      $\mathcal{F} \leftarrow \mathcal{F} \setminus \{p\}$
    end if
  end for
  for all $p \in \mathcal{F}$ **do** ▷ Rule B
    if $e(p) \geq E_{\max}$ **then**
      $\mathcal{F} \leftarrow \mathcal{F} \setminus \{p\}$
    end if
  end for
end for
return $S' = (\mathcal{F}, \mathcal{P}, e, H)$

nodes with `has_real_data=True` when any are available, then returns the candidate with the highest fb(·). This guarantees the loop always makes progress even when the LLM call degrades.

Configurable evaluation layers

The four evaluation surfaces in sections 3.1 and 3.3 are independently configurable; the defaults reproduce the equations above, so an unmodified configuration is a no-op.

Composite formula. Selected by `evaluator.composite`. The reduction $\{a_k\} \to$ `_scientific_score` is one of: weighted harmonic mean (default), arithmetic mean, bottleneck `weighted_min`, or weighted geometric mean. The harmonic and geometric forms apply a floor $\epsilon = 0.01$ to a zero-valued axis so the denominator stays finite.

Frontier-score strategy. Selected by `bfts.frontier_score`. eq. (5) is the `scientific_plus_diversity` case. The alternatives are `scientific_only` (drops div); `depth_penalized`, adding $-\lambda\,\text{depth}(n)$ on top of div(n) with $\lambda =$ `depth_penalty_lambda`; and `ucb_like`, adding $c_{\text{ucb}}\sqrt{\log N/(e(n)+1)}$ on top of div(n), where $c_{\text{ucb}} =$ `ucb_c` and $N = \sum_{n'} e(n') + |\mathcal{F}|$.

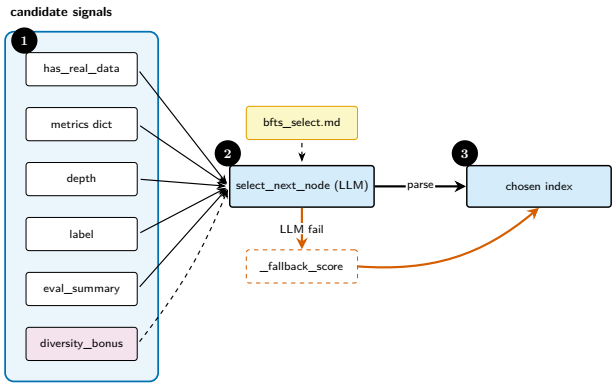

Figure 9: Selection signals fed to the LLM that picks the next node to operate on. Inputs are concatenated into a structured candidate description; the soft `diversity_bonus` is a tiebreaker, not a primary signal.

Axis set. Selected by `evaluator.axis_mode`. `dynamic` (default) builds the axis set from the generic 5-axis floor plus the active rubric and `idea.json` plan keywords. `legacy` pins to the 5 canonical axes. `custom` feeds an explicit $\{(\text{name}, \text{description}, w)\}$ list verbatim through to the judge LLM.

Selection prompts. Set via `bfts.select_prompt` and `bfts.expand_select_prompt`: a pair of `FilesystemPromptLoader` keys lets the user swap in alternative templates for the two $\text{select}^{\text{LLM}}$ calls in algorithm 1 without editing source code. The selection template must accept the placeholders `experiment_goal`, `memory_context`, and `candidates`; the expansion-target template accepts `experiment_goal` and `candidates`.

3.4 Expansion (one child per call)

The `expand` function generates *at most one* new child per invocation (no pre-batching). Callers wanting more children for the same parent call `expand` repeatedly, threading the previously created children through the `existing_children` argument so the LLM can avoid proposing duplicate directions.

The label of each new child is decided by the LLM rather than by a hardcoded template; the expansion prompt, reproduced verbatim as section B.1, is fed:

- the parent's status (`succeeded` / `failed/no-real-data`) and `_scientific_score`;
- the parent's structured `node_report` (`delta_vs_parent` / `concerns` / `hints`, produced by the node-report builder) when present, so the planner can target specific weaknesses;
- sibling scores at the same depth, and ancestor scores from root to parent;

- tree-wide diversity metrics (unique labels seen so far, depth distribution); and
- the labels of children already spawned at this parent, so duplicates are not proposed.

A child is created in state open, handed to the agent loop for execution (section 3.6), and transitions to done when all axes are populated or to failed if the agent loop raises.

Coverage steering. The expansion-target selector is inherently a cross-branch scheduler — it sees every branch's score, metrics, and summary — but score alone said nothing about the run's declared claims, and in one real run a ten-node tree produced ten variations of the same headline experiment while the declared mechanism claims received no dedicated experiment. The goal text handed to `select_best_to_expand` is therefore augmented with a run-level *claim-coverage* block derived from the metric contract: which declared claims already have supporting measurements somewhere in the run and which remain uncovered, so that "covers a still-uncovered claim" can inform *which* node to expand. The measurement names a claim requires are its *contract-evidence names*. Coverage is computed conservatively on two counts: a claim is covered only when a *majority* of its required names have been emitted — deliberately stricter than the final gate's any-name rule, since for steering one accidentally shared generic name would suppress the dedicated experiment forever, whereas erring uncovered merely risks a redundant measurement — and only nodes the evaluator credited with real data contribute, aligned with the gate, so a faulty branch's mere measurement *names* cannot tell siblings "already covered." The same status, recomputed per node, also reaches each executing agent (section 3.6); the companion hints that chain *computed* evidence along a single lineage are the subject of section 3.8.

3.5 Pruning and termination

`should_prune` implements the pruning predicate Π (eq. (1)): total-node cap, depth cap (the `max_depth` config field), and the sterile flag. The frontier retire predicate ρ (eq. (3)) is applied after each node completes; it does not delete the node — its history stays on $\mathcal{T}$ — but the frontier pool shrinks so the search spreads rather than mining the same parent indefinitely.

The loop terminates when any of:

- the global `max_total_nodes` cap is hit;
- *every* frontier node fails Π (depth cap, sterile, or budget exhausted) and the pending pool is empty;
- the per-call step / cost budgets enforced by the cost tracker run out;

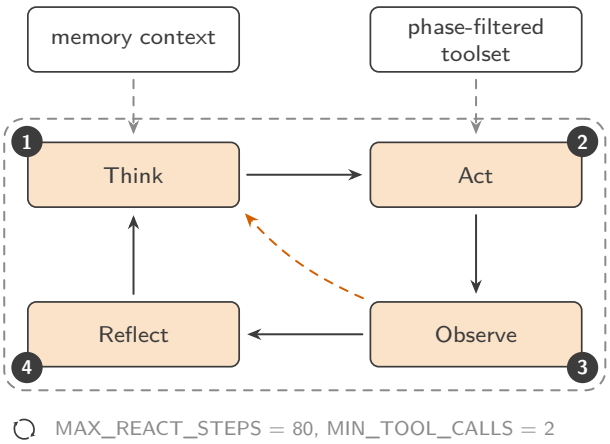

Figure 10: The ReAct loop. Solid edges are the main cycle; the dashed back edge represents an early-exit when the observation already closes the open question.

- the *stagnation detector* `detect_stagnation` observes that the running max of `_scientific_score` has not improved over a sliding window;
- a `LineageDecision` from the lineage-decision module explicitly halts the branch (see section B.4).

Which of these fires most often is an empirical question that section 5 is meant to answer once a frozen checkpoint is in place; the present report does not claim a distribution.

3.6 ReAct driver

Each node's expansion runs a ReAct agent loop [1] in the `AgentLoop` class (fig. 10): the model reasons over its accumulated context, issues tool calls, observes the structured results, and repeats. The maximum step count is `MAX_REACT_STEPS = 80`; it can be overridden per-instance via the `max_react_steps` keyword. A separate guard `MIN_TOOL_CALLS = 2` prevents trivially short trajectories. The agent's system prompt is reproduced verbatim as section B.6.

The set of tools available to the agent is filtered per phase by the tool manager; for example, the exploration phase masks the paper-writing tools so that BFTS expansion cannot short-circuit into a draft. A small set of MCP tools is reserved for `ari-core` itself (memory CoW operations) and is hidden from the LLM, ensuring that the model cannot set an arbitrary node id and bypass the memory skill's copy-on-write check.

Obligations at node start

A node does not begin from a blank prompt. Before the first model call, the loop injects deterministic context: the *working context* of section 3.10 — the experiment's fixed core plus its ancestors' conclusions — and, once the run's metric contract exists, a *contract obligation* message stating, in domain-neutral

terms, what the node must do for an eventual claim to be scientifically valid rather than merely plausible: verify correctness against an independent reference on the problem sizes the headline numbers use, recording the residual as a measurement; *measure* — never hardcode — any ceiling used as a normalisation denominator; tag the provenance of each ceiling and correctness check so the gate can confirm they were run; declare how the headline metric is recomputed from its raw operands; and emit, under the *exact* declared names, the measurements that make each declared claim evaluable — a negative outcome is acceptable, an unevidenced claim is not. The text carries no domain knowledge; the agent fulfils each duty as its domain calls for (a microbenchmark for a throughput ceiling, a baseline run for an accuracy ceiling, an analytic check for a numerical method).

Three run-level annexes ride the same message: the node's claim-coverage status (section 3.4), which asks it to prioritise one or two still-uncovered claims it can feasibly measure; the inherited-data note of section 3.8; and the platform note of section 3.9. These run-level invariants are *pinned*: the sliding context window always retains them — together with the survey, idea-generation, and metric-spec tool results — while older turns fall away. At the root, where the contract does not yet exist at context-build time, the obligation is injected mid-loop by the handler of the contract-minting tool call, so no node runs blind to the declared claims and none receives the obligation twice.

Emission feedback

Measurements leave a node through a single structured emission tool (`emit_results`), which writes the node's results file, validates it against the metric contract, and returns *contract warnings* naming any required evidence still missing. Early runs exposed a failure shape at exactly this point: the warnings arrived inside a tool result, the harness force-finished the node right after its first completed job, and the unmet obligations were left with no steps in which to act — one observed node ended with 66 of its 80 steps unused. The loop therefore turns point-of-emission feedback into an actionable turn: when an emission carries contract warnings, it injects a continuation nudge — the missing measurements, the steps remaining, and the instruction to run them now and re-emit (a re-emission overwrites) or to conclude only if they are infeasible in this node — and places the node in a *contract hold* that suppresses the force-finish guard described next. The hold expires ten steps before the step cap, preserving the anti-doom-loop backstop, and a warning-free emission clears it.

Force-finish guards

The loop guards both ends of a trajectory. Against premature success claims, a node may not finish before `MIN_TOOL_CALLS` tool calls, and — whenever its toolset contains an execution tool — not before it has executed something and read the output. Against the repetitive-loop failure mode documented for ReAct agents (the original study attributes a large share of reasoning failures to re-issued, near-identical actions [1]), the loop *force-finishes*: once the node has a completed job whose output was read, at least five steps have elapsed, and no contract hold is active — or in the final three steps before the cap once the minimum guards are met — the tool-call requirement is dropped, so the model's next output must be its final structured report. Repetition is also damped at the tool level (the idea-generation tool is suppressed after its first successful call), and injected user messages are deferred to message-block boundaries so the replayed conversation never splits a tool call from its responses.

3.7 Lineage and reproducibility

A lineage is the chain of ancestors of the best node, written into the checkpoint as the `lineage` key in `tree.json`. The `LineageState` dataclass captures the running statistics that BFTS uses to decide whether to continue, and the cross-checkpoint walker `walk_ancestor_ckpts` provides parent pointers across runs. Idea pools are inherited via `get_idea_pool_for_ckpt`, and a ranked root-idea selection is recorded by the `RootChoice` dataclass so the choice is auditable post hoc (prompt: section B.5).

Reproducibility hinges on three invariants. (i) Every randomised operation seeds itself from the node identifier. (ii) Every tool call records its arguments and outputs into the node's working tree, never the parent's — this is enforced by the MCP layer's `cow_node_id` check. (iii) The cost ledger attaches a per-node dollar amount at `CostTracker.init`, so the budget check is deterministic given the same prompts.

Offline by default. Reproducibility also dictates what the search loop is allowed to read. By default a node's agent loop runs *offline*: the web-access skill is phase-gated to the paper-writing and replication stages, so a node cannot consult time-varying live search results during the search itself, and the per-node trajectory therefore does not depend on the state of the web on the day it ran. Literature lookup still happens, but earlier and out of band — a bounded survey performed at idea time, before the search begins (section 3.11). A run that genuinely needs live information can opt in (`bfts.allow_web`) to expose web access to the node agent during exploration; when it does, the run writes a non-reproducible-trajectory marker into its checkpoint, so a later reproduction is never silently treated as exact. The determinism contract stays honest either way: the default run is byte-reproducible, and a run that trades reproducibility for live information records that trade in its own provenance.

3.8 Lineage chaining for computed evidence

Not every claim a run sets out to support is measured fresh. Some evidence is *computed* from measurements that already exist — a parameter fit over earlier probes, a held-out validation of that fit, a model-based selection among configurations. Such claims turned out to be structurally unreachable by independent tree nodes: in a real run, children directed at fitting or validation but expanded from a parent that carried no measurements regressed to re-running single probes, because "compute from the data you already have" was never a visible option — the child had no data, and nothing told the selector which node did.

The mechanism that closes this gap uses only the parent→child working-directory inheritance the tree already has (a child executes inside a copy of its parent's working tree; section 4.2 relies on the same fact), as one hint per side of the expansion. On the parent side, the claim-coverage block appended to the expansion-selection goal (section 3.4) gains a lineage hint that names the node whose working tree holds the most contract-evidence names, so a fit-or-validate child is preferentially expanded from the node that already holds its inputs. On the child side, a child whose inherited directory contains lineage measurement files is told, at start (the inherited-data note of section 3.6), which files are present and which exact contract names they contain, and is instructed to compute the derived evidence from those files rather than re-run the underlying experiments.

Only names cross node boundaries: file names and measurement names, never values and never a sibling's conclusions. The hints are run-level coordination metadata of the same kind as the contract itself, so the branch isolation of section 3.10 is preserved — a faulty branch's numbers still cannot leak into a sibling's reasoning — while the computed-evidence chain (fit, then validate, then select) can execute along a single lineage instead of collapsing back into repeated probes.

3.9 Platform probe

A research plan can be wrong about its platform before it is ever wrong about its science. In one real run the declared claims required hardware-counter profiles, and the assumed profiler was verifiably absent from the compute partition the experiments ran on: those claims were permanently unsatisfiable, and the final gate — correctly — blocked the paper for evidence no node could ever produce. Platform feasibility must therefore be *measured*, not assumed — and measured once, before anything downstream commits to a measurement vocabulary.

When the run is configured with a batch partition, run start issues a one-shot *platform probe*: a short scheduler job, executed on the compute partition itself, that checks by name the availability of a small configurable list of measurement tools (profilers, hardware-counter utilities, and similar) and records the machine architecture. The result is cached as a checkpoint artefact (`platform_capabilities.json`) and not re-probed within the run. The probe is best-effort by design: no partition configured, a missing scheduler, or a queue wait beyond its timeout all degrade to a recorded skip that writes nothing — a probe failure leaves every consumer unconstrained and changes nothing downstream.

The probed facts are then relayed — as data, never as built-in hardware knowledge — at the three layers where a platform-blind plan would otherwise arise. At *idea* time they are folded into the research topic, so the ideation stage (section 3.11) plans only measurements the platform can take: feasibility is fixed at the source rather than patched downstream. At *contract* time the claims extractor that derives the metric contract from the selected idea's plan receives them as a verified capability note, so the contract never declares evidence that depends on absent tooling — doubly important because the contract is minted once and then frozen (section 4.3). At *node* time the pinned obligation message (section 3.6) carries a platform note listing the verifiably absent tools, so the executing agent does not spend its ReAct steps discovering missing commands the hard way. Knowledge of *which* tools are worth probing lives in the HPC skill's configurable list; the core orchestrator and the gate stay domain-free and merely relay what was measured.

3.10 Research memory

Nodes do not run in isolation: each one inherits what its ancestors established and, in turn, leaves a record for its descendants and for the paper writer. ARI keeps this record in a per-run *research memory* governed by two requirements a free-form log does not meet. First, *branch isolation*: a node may read only the memory of its own ancestors, never that of a sibling or an unrelated checkpoint, so two parallel leads cannot contaminate each other — the same ancestor-scope predicate that defines a lineage in section 3.7. Second, *groundability*: because the memory ultimately feeds a paper, an entry must be able to point at the evidence behind a result rather than merely restate the result, so a claim can be traced back to the artefact that supports it.

Memory as an index over evidence

The single source of truth for what a node produced is its structured self-report — the `node_report.json` written when the node completes (section 4.2 details its fields). A memory entry does not copy those fields. It carries a short searchable text, a *kind*, a pointer back to the originating node report, and content-hash references to the specific artefacts — logs, source files, output files — a result rests on. A memory entry is thus an *index into evidence*, not the evidence itself: the artefacts it cites can be re-hashed at any time to confirm they still exist and still match, an integrity

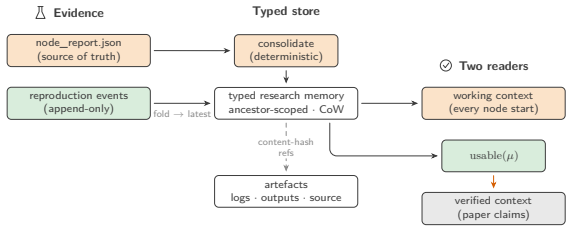

Figure 11: Verifiable research memory. A deterministic consolidation of each node's `node_report.json` — the source of truth — populates a *typed* store that *indexes* artefacts (logs, outputs, source) by content hash. Reproduction events are appended and folded down to a latest status, and an admissibility predicate usable(μ) (eq. (6)) gates which records reach the verified context. The same store yields two readers: the *working context* injected at the next node's start (the inheritance path of exploration) and the *verified context* that grounds the paper's quantitative claims (section 4.2). Ungrounded reflection may steer exploration but never the paper body.

check that classifies every reference as verified, missing, or mismatched. The *kind* is drawn from a fixed vocabulary — observation, experiment result, failure case, procedure, reflection, artefact summary, paper claim, and reproducibility event — so a reader can ask for exactly the records it needs: the failures along a branch, say, or only the results that are backed by artefacts.

Consolidation at node end

When a node completes, a deterministic *consolidation* step reads its node report and emits typed entries: an experiment-result entry, with provenance references to the artefacts it measured, for a node that produced measurements; a failure-case entry for one that failed. The step invokes no language model, so the same node report always consolidates to the same entries — consolidation is a pure function of recorded state, and re-running a checkpoint reproduces the memory it produced. It is a loop hook, not an agent action: the search agent is never asked to "save to memory", and in practice the agent does not reach for the recall tools on its own. The memory reads the loop depends on are performed by the loop itself, deterministically — which is what lets the memory carry provenance without putting a non-deterministic retrieval on the critical path (section 7.1).

Two readers: working and verified context

The store feeds two consumers (fig. 11), and the distinction between them is the heart of the design. At the *start* of every node the loop injects a *working context*: the experiment's fixed core — goal, target metric, hardware — together with the conclusions its ancestors recorded. A child therefore begins from what its lineage has already established, deterministically and scoped to its own branch, rather than from an aggregate text dump. At *paper* time the pipeline builds a *verified context* from the best node's lineage: it folds each result's reproducibility events down to a latest status and ranks entries so that reproduced results outrank merely-artefact-grounded ones, which in turn outrank ungrounded reflection. An entry μ is *admissible* as a quantitative paper claim only when it is artefact-grounded and not contradicted by a reproduction attempt:

$$\text{usable}(\mu) \;=\; \text{grounded}(\mu) \;\wedge\; \big(\text{repro}(\mu) \neq \text{failed}\big). \tag{6}$$

Ungrounded reflection can still steer the search through the working context, but eq. (6) holds it out of the paper body: the writer's quantitative claims rest only on admissible, reproduced-first entries. This is the memory-side counterpart of the evidence assembler (section 4.2); where the assembler decides which artefact *bytes* the writer may read, the verified context decides which *results* are allowed to become claims.

Reproducibility as an appended event

Because past entries are byte-stable — a node writes only under its own identifier and existing entries are never edited in place — a result's reproducibility status is not patched when the result is later re-run. Instead the re-run *appends* a reproducibility-event entry, and any reader folds the events for a target down to their latest state. A result that fails to reproduce is thereby demoted out of the claim set of eq. (6) — and can be reported honestly as a limitation — without rewriting the history that recorded it as promising in the first place.

3.11 Ideation: seeding the root

The tree's root node does not begin with code; it begins by deciding what to investigate. The root's agent runs the idea stage: a bounded literature survey followed by multi-agent idea generation, converging on a selected idea (`idea.json` in the checkpoint) whose plan yields the run's metric contract (section 4.3). Because the survey happens here, at idea time, the search itself can stay offline (section 3.7): the survey retrieves live literature from Semantic Scholar — keyword search enriched by a two-hop traversal of the citation graph — and the deliberation then works over the retrieved abstracts.

Ideation is deliberately multi-agent. Single-agent, archive-conditioned brainstorming is known to collapse onto a narrow idea distribution — the AI Scientist reports that its idea generation "often results in very similar ideas across different runs and even models" [2] — whereas VirSci [7] simulates a scientist team (collaborator selection, round-table discussion, idea generation, novelty voting) and reports that the multi-agent system improves on a single-agent executive by, on average, +13.8% in alignment with and +44.1% in potential impact on contemporary research.

ARI wraps VirSci at two fidelities. The default path re-implements VirSci's discussion flow inside the idea skill, using the vendored VirSci prompt templates directly where available, with retrieval re-based

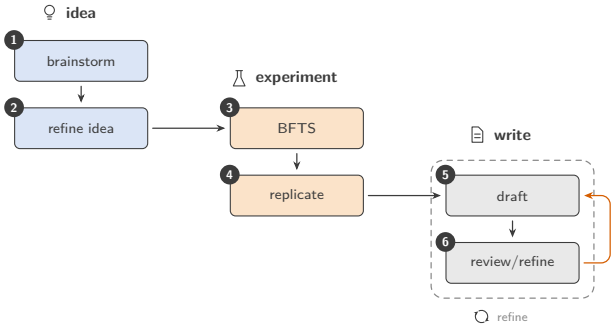

Figure 12: The paper-generation lifecycle as phase-grouped columns: idea, experiment, write (steps 1–6). The dashed enclosure marks the review/refine pair; its back edge is the only non-monotonic transition in the lifecycle (section 4.5).

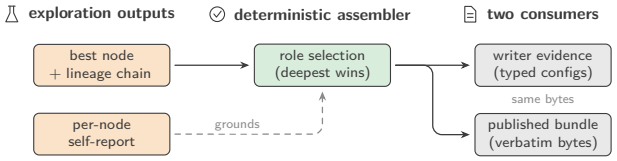

Figure 13: Evidence assembly. The deterministic assembler walks the best node's lineage and, grounded by each node's structured self-report (content hashes, success self-assessment, literal build/run commands, captured hardware), makes *one* role-specific selection that feeds *two* consumers with byte-identical content: the writer's lineage evidence and the published artefact bundle. Each contributing file is chosen by recorded metadata; on a path collision the deepest occurrence wins, and a chain deletion removes the file.

on the live survey and model calls routed through ARI's LLM layer; candidate ideas carry VirSci-style self-assessments (novelty, feasibility, clarity) and are ranked by a novelty-weighted sum. An opt-in *VirSci-live* mode instead drives the vendored VirSci engine itself, unmodified (fig. 18): a frozen Semantic-Scholar-derived snapshot — a paper corpus with a SPECTER2 retrieval index and a freshness-ranked co-author graph — is materialised under the checkpoint, VirSci's own `select_coauthors` forms a scientist team over that graph, and its K-round `generate_idea` deliberation produces the candidates. The multi-agent mechanism ARI builds on is thus the published one, executed against current literature rather than paraphrased; freezing the snapshot keeps the deliberation substrate reproducible, and the live path degrades on any failure to the re-implemented loop, so enabling it never regresses a run.

Two run-level facts are folded into the deliberation on both paths. The platform constraints verified by the probe (section 3.9) are appended to the research topic, so every prompt that consumes the topic plans only measurements the platform can take; and on a derivative run the ancestor checkpoints' idea pool (section 3.7) is injected read-only, so the team refines, extends, or explicitly pivots from what the lineage already proposed instead of re-deriving it. The ranked selection among the candidates is recorded for post-hoc audit (section 3.7), keeping the run's starting point as traceable as every later expansion.

4 Paper Generation Pipeline

4.1 Pipeline overview

Once exploration terminates, the surviving lineage is handed to the paper-generation pipeline, the second half of a run (fig. 12; fig. 3 places it in the run-level stage DAG). A *write* phase compiles the lineage into a draft, and a *review* phase scores that draft against a rubric; the two are coupled by a refine loop (section 4.5). The pipeline persists three outputs: the LaTeX draft, the per-leaf review with its rationale, and the rubric instance used to score the run (kept because the rubric may be generated per paper rather than fixed in advance).

The organising principle of the pipeline is that every quantitative statement in the draft must be traceable to something exploration actually produced. Two mechanisms carry that principle. A deterministic *evidence assembler* (section 4.2) fixes *which* artefacts the writer may draw on, as a pure function of recorded metadata. A *claim–evidence contract* (section 4.3) fixes *what* the paper must say about them: the falsifiable claims declared at idea time, the measurement names that count as evidence for each, and a deterministic gate that re-derives every reported number from the recorded results before the paper can ship. The remaining sections — the rubric formalism (section 4.4), the review/refine loop (section 4.5), and the PaperBench replication path (sections 4.6 and 4.7) — build on these two.

4.2 Evidence assembly

Between exploration and writing sits the evidence assembler (fig. 13). It decides which of the lineage's per-node artefacts the writer may draw on and reshapes them into a single science-facing record. The design keeps this seam auditable — which node a claim traces to is a function of recorded metadata, not of a language model's judgement — so that the contract of section 4.3 and the guardrails of section 4.8 have something concrete to enforce.

Selection is role-specific: one node may contribute differently to three consumers, and a single predicate family decides each. A node enters the *synthesis* set if it carries real measurements (or the evaluator certified its run a success); it enters the *code* set if it introduced or modified a source file relative to its parent; and it enters the *narrative* set if its step succeeded. A node that merely explored a dead end is admitted to none. Each criterion is a pure predicate over recorded metadata, so the same inputs always select the same contributing set.

Source files are gathered by overlaying, in depth order, the files each contributing node touched along

the ancestor chain $\mathrm{lin}(n^\star)$ of the best node $n^\star$. Because a child node executes inside a copy of its parent's working tree, one relative path may occur at several depths; the deepest occurrence wins, and a deletion along the chain removes the file — so the reconstructed set matches the best node's actual working tree. This selection then feeds *two* consumers from the same bytes: the writer's lineage evidence and the published artefact bundle, so the lineage code the paper describes is the code that is released. Selection is chain-only — a node off the best lineage is not published even when its measurements are reported by the synthesis set, and is logged for provenance — and, as a fidelity aid for pseudocode, the writer is additionally shown the raw source of the top-scoring nodes, which is not itself the published set.

The assembler reads code, parameters, and measurements *directly* from each node's working tree rather than from a summary. Where an experiment declared a typed result, the record separates *input parameters* (configuration knobs — problem size, thread count, seeds) from *measured outputs* (throughput, accuracy, latency) and from derived quantities (efficiencies, model-predicted ceilings). The separation is not cosmetic: per-key extrema reported across configurations are reduced deterministically and *only* over measured keys,

$$\mathrm{best}(m) \;=\; \operatorname*{red}_{c \in \mathcal{E}} c[m], \qquad m \notin \mathcal{I}, \tag{7}$$

where red is a maximum or a minimum according to the metric's declared direction, $\mathcal{E}$ is the set of contributing configurations, and $\mathcal{I}$ is the set of declared input keys held out of every reduction. Holding the inputs out is what stops a large input magnitude — a problem size of several million nonzeros, say — from being read as a headline result. The language model is confined to the one task that genuinely needs it: turning verbatim source and logs into prose and pseudocode and narrating methodology. It never supplies the numbers.

Grounding all of this is a per-node structured self-report written when a node completes (`node_report.json` in the checkpoint). It records, by content hash, which files the node added or modified relative to its parent; the node's own success self-assessment together with the concerns the evaluator flagged; the literal build and run commands; and the hardware the measurement ran on. The hashes make the source selection reproducible and let the released bundle carry a reproduction script built from the recorded commands; the captured hardware lets the paper's environment section be written from fact rather than left as "not recorded".

4.3 The claim–evidence contract

The pipeline's centrepiece is the *metric contract*: a run-level declaration of the falsifiable claims the eventual paper must support, each naming the exact measurement names that count as evidence for it (fig. 14).

Algorithm 2 Evidence assembly (exploration → writer input).

Input: lineage nodes $\mathcal{N}$ with per-node reports; metric-direction map
Output: science record $\mathcal{E}$: typed configurations, per-key extrema, methodology narrative

$n^\star \leftarrow \arg\max_{n \in \mathcal{N}} r(n)$ ▷ ties: validated, then deeper
$L \leftarrow \mathrm{lin}(n^\star)$ ▷ root-to-best ancestor chain
$\mathcal{C} \leftarrow \{\, n \in L : \textsc{Contributes}(n) \,\}$ ▷ role predicate (code)
$F \leftarrow \emptyset$ ▷ relative path ↦ (node, depth)
for all $n \in L$ in depth order **do**
  for all path p added/modified by $n \in \mathcal{C}$ **do**
    if $p \notin F \vee \mathrm{depth}(n) \geq \mathrm{depth}(F[p])$ **then**
      $F[p] \leftarrow (n, \mathrm{depth}(n))$ ▷ deepest wins
    end if
  end for
  for all path p deleted by n **do**
    remove p from F ▷ deletion overlays it away
  end for
end for
load the bytes of every $p \in F$ verbatim, under a size budget
for all measured key $m \notin \mathcal{I}$ **do**
  $\mathrm{best}(m) \leftarrow \mathrm{red}_c\, c[m]$ ▷ eq. (7); deterministic
end for
$\textsc{Narrate}(F)$ ▷ LLM: verbatim source/logs → prose, pseudocode
return $\mathcal{E}$

The claims are owned by the selected idea, so the implementing agent cannot drop an inconvenient one; the contract is persisted in the checkpoint at idea time and threads through the entire run. During exploration, nodes emit measurements under the contract's names; the same names steer expansion toward still-uncovered claims and seed the lineage chaining of section 3.8, so evidence that is *computed* from earlier measurements can be produced along a single lineage. During writing, the deterministic gate described below matches claims to evidence by exact name. A positive or a negative outcome both satisfy a claim; what the contract makes mandatory is the evidence to judge it.

The design descends from Story2Proposal [8], which converts a finished "research story" — a narrative plus its experimental evidence — into a manuscript by threading a persistent contract through a fixed pipeline of architect, writer, refiner, and renderer agents. Its shared contract registers the manuscript's visual artefacts under canonical labels, binds them to section-level obligations, and carries document-wide validation rules; evaluation agents for reasoning verification, data fidelity, and visual coherence observe intermediate artefacts and update the contract during generation, and the renderer closes with deterministic structural validation. What that contract guarantees is structural and visual integrity — unique labels, resolvable cross-references, narrative–visual alignment — and its authors are explicit about the boundary: the framework "enforces structural and visual obligations rather than generating missing scientific evidence," and a manuscript can remain "structurally valid while

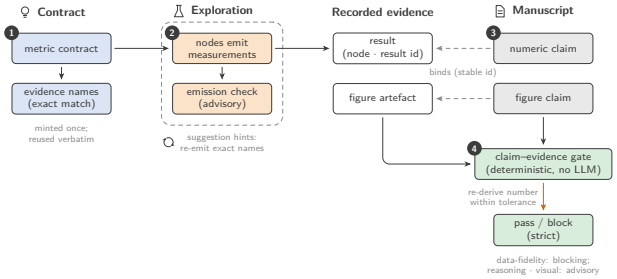

Figure 14: ARI's execution-grounded realisation of the Story2Proposal claim–evidence contract [8]. (1) The run-level metric contract declares falsifiable claims and the exact measurement names that count as evidence; once a claims-bearing contract is persisted it is reused verbatim. (2) Exploration nodes emit measurements under those names; an advisory point-of-emission check flags still-uncovered claims and offers suggestion-only name hints for re-emission — nothing is auto-bound. (3) Manuscript claims — numeric assertions and figure references — bind to the recorded evidence by stable node, result, and figure identifiers. (4) A deterministic claim–evidence gate (no language model) re-derives each reported number from that evidence within a tolerance; the data-fidelity axis blocks in strict mode, while the reasoning and visual-coherence axes are advisory.

containing less rigorous reasoning" [8].

ARI keeps the contract idea and the three evaluation axes but re-grounds the contract in execution. The declared object is not the manuscript's visual structure; it is the run's evidentiary obligations, declared before any experiment runs and bound to measurements that executed experiments actually emitted. Data fidelity is correspondingly re-implemented: instead of a language-model evaluation agent, a deterministic gate re-derives every reported number from the recorded evidence, and this axis blocks in strict mode while reasoning and visual coherence remain advisory. Story2Proposal validates how given evidence is written up; ARI also controls how the evidence comes to exist, which is what makes a blocking check sound.

Concretely, the writer emits an in-text anchor alongside each reported number: a comment-line declaration naming the claim the number supports, the metric, the derivation formula, and the operand configurations it was computed from. A deterministic linker reconciles anchors against the assembled science record, and the gate then recomputes each declared number from the recorded measurements using the named formula and compares it within a declared tolerance (a small relative tolerance by default). The declaration is forward-only — no reverse search guesses what a number might mean — so a wrong declaration fails as a mismatch rather than being silently repaired. Emission is also checked at the point of production: when a node emits results, the presence checks the final gate will later run are mirrored immediately, while the node can still re-emit; the warnings name still-uncovered claims and may carry conservative, suggestion-only lexical hints toward the required names. Nothing is auto-bound, and the gate itself never consults the hint logic.

Experience with real runs hardened four points of this seam, each a structural rule rather than a prompt fix:

- **The contract is minted once.** Once a claims-bearing contract has been persisted, every later call returns the persisted contract verbatim instead of regenerating it; a re-derivation request becomes a harmless re-read of the obligations. A language model's naming is not referentially stable — asked twice, it names the same quantity differently — and in a real run a mid-run regeneration silently renamed the evidence vocabulary, hiding measurements that sibling nodes had already emitted from the exact-match gate: claims reported as missing evidence had in fact been measured under the earlier names.
- **Parsing meets the writer where it is.** The gate's numeric verification parses scientific notation in both its common forms (exponent suffixes and $m \times 10^k$ literals) as a single value, and normalises math delimiters and spacing macros away so that a number and its unit are adjacent when a value–unit pair is extracted. Writer declaration quirks that recur in practice — synonymous formula names for the same derivation, operand references written with a redundant labelling prefix, metric names carrying LaTeX escapes, one anchor identifier shared by several mentions (disambiguated per mention) — are normalised at parse time rather than reported as mismatches. The rationale is twofold: the gate's verdicts should measure the paper's numbers, not its surface syntax; and because anchor lines may not be edited during refine, a declaration that dead-ends at parse time could never be repaired downstream.
- **Review findings reach the refiner whole.** Every warning raised by the semantic review — a complementary language-model pass that flags overclaims and unsupported causal language — is forwarded to the refine step (section 4.5) as an advisory revision entry, not a truncated subset; a warning without a forwarded entry would never reach the refiner, and its count could never decrease. The post-refine resolved count is a raw delta between the two review passes, so a refine that introduces new findings shows up as a negative delta — a regression — rather than being clamped away.
- **Blocking stays reserved for objective falsehoods.** Gate findings come in two tiers. Policy-gated findings — a reported number that fails recomputation, an operand that cannot be resolved, a numeric assertion whose cited evidence is absent — block the final check under the strict policy and are reported without blocking otherwise. Objective falsehoods — a violated universal or declared invariant, a failed or missing declared

correctness check, a placeholder constant where a measured ceiling is required, a metric that cannot be recomputed from its own raw operands, a declared claim with no supporting measurement anywhere in the run — block the final check under *any* policy, because each is deterministically false or unsound. The subjective findings of the semantic review are advisory by design, steering the refiner without ever gating publication on a matter of judgement.

4.4 Rubric formalism

We model the *rubric* R as a finite tree whose leaves carry tuples $(c_i, w_i, \text{judge}_i)$. Each leaf i has a category descriptor c_i, a non-negative weight w_i with $\sum_i w_i = 1$, and a *judge function* $\text{judge}_i : P \to [0, 1]$ that returns a graded score. The paper score is the convex combination

$$\text{score}(P) \;=\; \sum_i w_i \,\text{judge}_i(P). \tag{8}$$

This tree-structured rubric is shared with PaperBench [9], where binary leaf grades propagate up the tree as weighted averages; ARI reuses the upstream package through a vendored copy (section 4.6) so that its grade and aggregation semantics track upstream verbatim. Two judge families coexist: an LLM-backed judge for qualitative leaves, and a deterministic judge for leaves with a binary or numeric verifier (e.g. "is the cost reported in absolute dollars?"). Both satisfy the same evaluator interface, so the search engine (section 3) and the paper pipeline reduce axes to a scalar through one seam.

A rubric is generated in one of two *modes*. In *agent-benchmark* mode the tree decomposes the paper by scientific contribution and each leaf asks whether a candidate submission's executed output reproduces the paper — the original PaperBench framing. In *paper-audit* mode the tree's top-level children are a fixed set of reproducibility axes and each leaf grades the descriptive completeness of the paper text itself — a framing inversion aimed at HPC reproducibility audits, where the question is not "can an agent reproduce this paper?" but "does this paper describe enough to be reproducible?". Venue knowledge (which axes, how they are phrased) is supplied by a per-venue template rather than baked into the engine, so ARI core stays domain-agnostic (P4) and a new venue is a data change, not a code change.

Lifting the paper-audit score out of a degenerate regime — where an empty submission drives every submission-oriented leaf to "no" — required rephrasing the leaf questions to interrogate the *paper* rather than a (missing) submission, feeding the judge the paper's figures alongside its text so vision-capable models can use them, and admitting optional artifact-description and artifact-evaluation appendices as extra input. These adjustments live entirely in ARI's wrapper layer; the vendored judge is untouched, which keeps scores comparable across upstream versions. We deliberately do *not* synthesise a reproduction log from the paper's own reported numbers: doing so would short-circuit the executed-submission stage and inflate the score in a way not comparable to leaderboard runs.

4.5 Review/refine loop

The refine cycle iteratively rewrites P_t to P_{t+1} under the review feedback $J(P_t)$:

$$P_{t+1} \;=\; \text{Refine}\big(P_t, J(P_t)\big). \tag{9}$$

The cycle stops when the reviewer accepts the draft — modelled as the *convergence criterion* $\text{score}(P_{t+1}) - \text{score}(P_t) < \epsilon$ — or when a small per-section revision cap is reached. Which branch fires more often is an empirical question section 5 is meant to answer at panel scale; we make no quantitative claim here.

The feedback $J(P_t)$ is a merged revision stream with its sources kept distinct: the independent venue review (the rubric instance of section 4.4), the evidence-grounded semantic review forwarded whole (section 4.3), and mechanical revision entries derived from gate findings. The merge is structural — no language model synthesises the streams — so the refiner sees each finding in its original terms, and the independence of the venue review is preserved alongside the evidence-grounded checks.

The refine step is intentionally non-monotonic per axis: fixing a clarity issue may slightly degrade a numeric axis. This is the source of the back edge in fig. 12, and it is why the convergence criterion is defined on the aggregate score rather than per-axis — together with a per-axis floor (section 4.8) that prevents the aggregate from hiding a collapse on any single axis.

4.6 PaperBench integration

The replication path is *not* a re-implementation of PaperBench [9]: the upstream package is vendored unchanged so that ARI tracks its task and rubric semantics verbatim, and ARI contributes only a thin wrapper around it. The wrapper exposes the protocol-faithful loop as three composable stages with a shared vocabulary (fig. 15):

1. *rollout* — a replication agent works from the paper towards a submission (code, and optionally a reproduction script);
2. *reproduce* — the submission is executed in an isolated container, on the local machine or dispatched to a cluster;
3. *grade* — the vendored judge scores the submission against the rubric tree.

Two design choices keep this faithful to the benchmark. First, grading is left to the vendored judge; ARI's adapters only reshape its output and aggregate

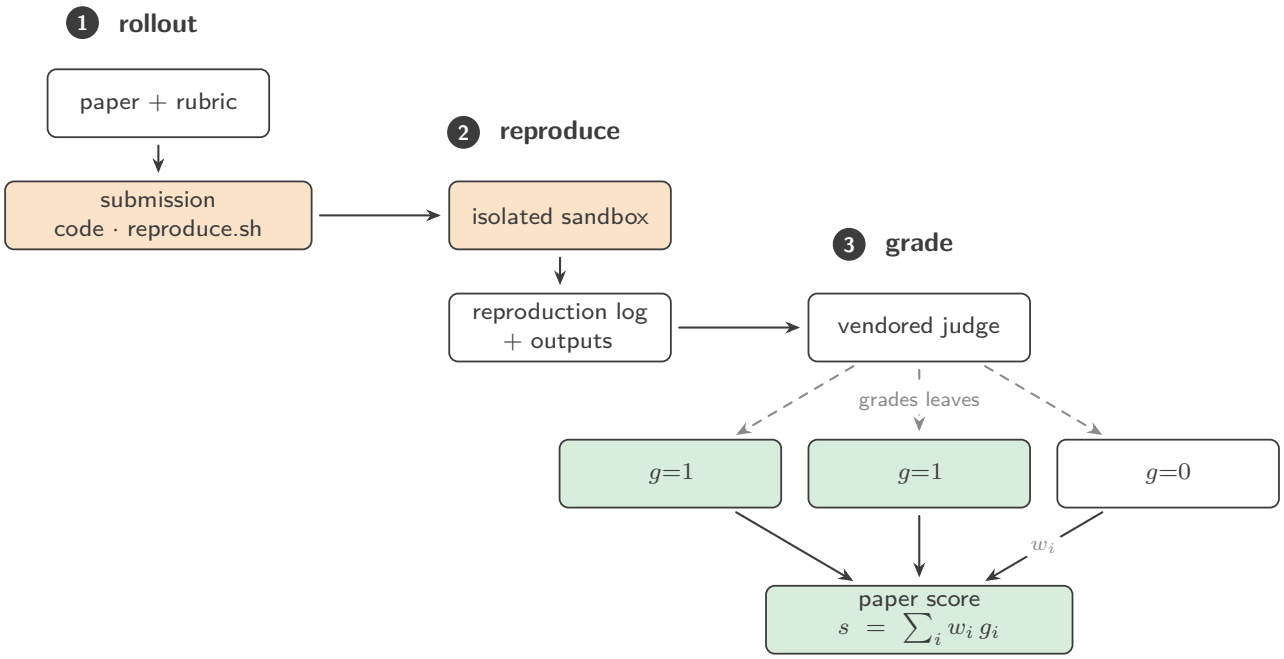

Figure 15: The PaperBench replication path as three composable stages. The agent *rollout* turns the paper and its rubric into a submission (code and a reproduction script); *reproduce* executes that submission in an isolated sandbox, capturing the reproduction log and its outputs; *grade* scores the submission against the *weighted rubric tree*, where the vendored judge assigns each leaf a grade $g \in \{0, 1\}$ and the paper score is the weight-aggregated roll-up $s = \sum_i w_i g_i$ (eq. (8)).

repeated runs into a single score, so upgrading PaperBench is a vendor swap with no change to how leaf grades are combined — the same envelope feeds eq. (8), with $\text{judge}_i \in \{0, 1\}$ for binary leaves. Second, the wrapper *fails loud*: when a benchmark-fair run cannot be served (no container runtime, no scheduler, no GPU resource registration) it raises rather than silently degrading to a weaker host, because a silent downgrade would make the resulting score incomparable to a leaderboard entry. Legacy silent-fallback behaviour is available only behind an explicit opt-in.

A second concern is keeping the replication agent honest about what the host actually provides. Left to the upstream prompt, an agent tends to scaffold a reproduction script that calls binaries the host lacks, and to default to Python regardless of the paper's real language. ARI counters both by priming the agent to probe the environment before it scaffolds, and by injecting a per-host notes block — generated from a live probe of the available toolchain, GPUs, and network reachability — so that what the agent reads matches what the host can do.

4.7 Domain breadth: the execution-profile contract

PaperBench upstream assumes a single container with one GPU. To cover papers whose methodology is fundamentally parallel — multi-rank MPI kernels, multi-node training, cluster-specific module environments — ARI extends the rubric with an optional *execution profile* (fig. 16). The profile is domain-agnostic: it names a *parallel-execution shape* (single-CPU, single- or multi-GPU, MPI, MPI+GPU) together with the paper's scale envelope and any scheduler hints, rather than anything about the paper's scientific field. When a paper is single-machine the profile is omitted and the pipeline degenerates to the original single-node invocation: the contract is additive, not breaking.

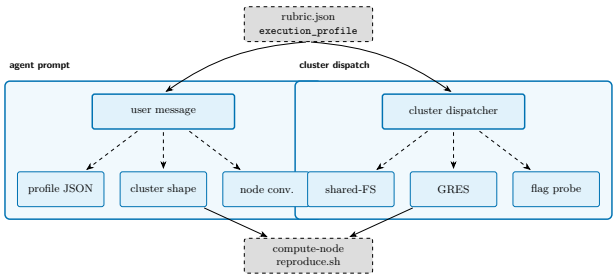

Figure 16: The execution-profile contract threads from a single rubric artefact into two consumers: the replication agent's prompt (left) and the cluster dispatcher (right). Runtime probes adapt the dispatched resource request to the local scheduler, so one rubric runs faithfully on heterogeneous clusters without being modified.

The profile feeds two consumers (fig. 16). It is appended to the agent's prompt as a compact "compute-node execution conventions" block — shared-filesystem discipline, a preference for the scheduler's launcher over a bare MPI launcher, environment activation, and wall-clock wrapping — and it parameterises the cluster dispatch (node, GPU, memory, and placement requests). Because clusters disagree on which scheduler flags they accept, the dispatcher probes the local scheduler at run time and drops flags it does not support, so the same rubric dispatches faith-

fully on a multi-GPU cluster, on a sandbox partition without GPU registration, or on a single workstation. A regression guard enforces the inverse: ARI running inside an unrelated cluster allocation must never leak cluster- or MPI-specific prose into the prompt for a non-HPC paper. Only this wrapper layer changes; the vendored benchmark and ARI core are untouched across the extension.

4.8 Failure modes and guardrails

Four failure modes are common enough to be named. None is hypothetical: the closest published predecessor documents generated drafts that misreport the experimental hardware and hallucinate entire result tables despite prompt-level instructions to use only real results [2]. ARI's position is that each failure mode needs a structural — not merely advisory — mitigation:

- **LLM-only support**: the paper asserts a claim that no node in the lineage substantiates. The mitigation constrains the writer to a pre-selected set of lineage artefacts, so every numerical or causal claim is traceable to a node that produced it; the draft cannot cite evidence that was not generated. The companion constraint is on the memory side: the verified context (section 3.10) admits a result as a quantitative claim only when it is artefact-grounded and has not failed a reproduction attempt (eq. (6)), so a reproduced result is preferred and an unsupported reflection cannot become a headline number.
- **Citation hallucination**: the paper cites a nonexistent work. The mitigation is structural — the writer may not invent citation keys, and every bibliography entry is verified against an external index before it is admitted (section 6).
- **Refine drift**: the aggregate score climbs while the paper loses coherence. The mitigation is a per-axis floor: if any axis falls below a threshold the refine is aborted, so a gain on one axis cannot mask a collapse on another.
- **Number drift**: a reported quantity does not match the recorded result it summarises. Because every numeric claim names the node and measurement it rests on (section 4.3), the deterministic claim–evidence gate re-derives each from the recorded results and compares it within a tolerance (fig. 14); a value that fails to match the recomputed result is flagged, and the two-tier blocking policy of section 4.3 decides whether the finding rejects the draft at the final check or is reported for repair.

The first guard also bounds context: the writer is given the selected lineage artefacts and the top-scoring nodes' source, under a size budget, so the draft stays within the model's context window even for long runs.

5 Preliminary Observations

This chapter reports what the running system establishes at v0.9.0. The evidence is of three kinds: an end-to-end generation run whose released paper passed every verification gate (section 5.1); properties that hold by construction and are confirmed in operation (section 5.2); and an earlier reproduction exercise carried out on the then-current v0.8.0 pipeline (section 5.3). All of it is preliminary: the end-to-end observations are single runs rather than a controlled benchmark, and the panel-scale study they call for is deferred (section 5.4). Quantitative figures elsewhere in this report are illustrative, generated from fixed-seed sample data so that the document build is reproducible.

5.1 Case study: a gate-verified sample paper

One end-to-end result is demonstrated and inspectable: the released sample paper, an eight-page study produced autonomously by a real run and reproduced in full in section C. It studies compressed sparse row (CSR) sparse-dense matrix multiplication (SpMM), selecting a store policy across right-hand-side (RHS) widths under the guidance of a performance model (the paper's "loopline" model) fitted from the run's own measurements. We trace this run end to end (fig. 17) because it is an existence proof for the verification machinery this report describes.

Claim structure. The run's metric contract — minted once at idea time and returned verbatim for the rest of the run (section 4.3) — declared twelve falsifiable claims. Nine are *independent probes* that a single node measures directly: correctness against a dense reference, the store-policy comparison, an irregular-gather and a locality-destruction probe, a page-footprint (translation lookaside buffer, TLB) stress probe, a compute-rate probe, and bandwidth microbenchmarks. The remaining three form a *computed-evidence chain* — fit the performance model's parameters, cross-validate the fit on held-out probe measurements, then select a configuration from the fitted model — where each claim consumes the output of the previous one rather than measuring something fresh.

Lineage closure. That chain is the case this run exists to demonstrate. A computed claim cannot be evidenced by a node that has no inputs, and three earlier runs confirmed the failure mode: nodes directed to fit the model from sibling probes re-ran a single probe instead and the chain claims stayed uncovered. Here the chain closed because a child node inherited its parent probe node's working directory and computed the fit, the cross-validation, and the selection from the inherited measurements — lineage chaining (section 3.8) made "compute from the data you already hold" the locally available move. The

Reported as grounded	Correctness: max absolute FP32 error 6.79×10^{-6} versus an independent dense reference; all twelve declared claims carry emitted evidence; the computed-evidence chain executed.
Explicitly hedged	Store-policy crossover values, double-precision error, measured bandwidth ceilings, and the performance model's numerical predictions — stated as methodology and marked not artefact-grounded in this revision.

Table 1: The released sample paper asserts only gate-grounded numbers and hedges the rest in its own text — the deterministic gate and the review–refine loop scoping the paper to what the run can support.

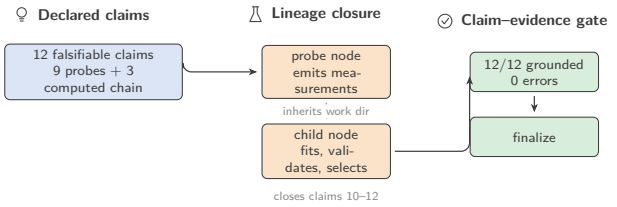

Figure 17: The case-study run as a worked example of the claim–evidence contract. Twelve declared claims split into nine independent probes and a three-claim computed chain; the chain is closed by lineage — a child node inherits its parent probe node's working directory and computes the fit, cross-validation, and selection from the inherited measurements — after which the deterministic gate finds every claim grounded and the paper is finalised.

final claim–evidence gate then found all twelve claims grounded and reported zero errors, so the paper was finalised.

What the gate verified, and what the paper hedged. The result's value is the *verified* property, not any headline number: every quantitative claim the released paper asserts is bound to a recorded artefact and re-derived by the deterministic gate, and the paper hedges explicitly wherever a statement would outrun its artefact grounding (table 1). The paper restricts its strong quantitative claims to the correctness metric — a maximum absolute single-precision (FP32) error of 6.79×10^{-6} against an independent dense reference — and presents the store-policy crossover, the bandwidth ceilings, and the model's numerical predictions as methodology rather than grounded outcomes. This honest scoping is the gate's effect, reinforced by the semantic review: that review flagged five overclaims in the draft and the refine pass resolved all five, with the resolved count recorded as a raw before/after delta (section 4.3) so that a refine introducing new overclaims would surface as a regression.

We read this result narrowly. The property is worth stating because earlier autonomous pipelines lack it: in the closest published predecessor, hallucination control is prompt-level, the automated review of a finished draft is advisory rather than blocking, generated papers were observed to hallucinate experimental details, and the authors list automatic verification of reported results as future work [2]. A single verified run demonstrates that the contract, lineage chaining, gate, and review–refine loop compose end to end; it does not establish a distribution, which is exactly the controlled study deferred below.

5.2 Properties holding by construction

Three further properties hold by construction and are confirmed in operation. Cost accounting is exact: a dollar cost is attached to every model invocation and accumulated, per run, into the checkpoint's results record. Exploration termination is not budget-only: the stagnation detector (section 3.5) provides a content-based exit and has been observed to fire before the step and cost budgets ($T_{\max}$ / $C_{\max}$) are exhausted. And the execution-profile pathway behaves as specified (section 4.7): rubric generation emits a multi-rank Message Passing Interface (MPI) execution profile, with the paper's reported scale envelope, for a high-performance-computing (HPC) paper and omits it for a single-machine paper, while a multi-flag cluster dispatch is accepted by the local scheduler once the runtime probes drop the flags it does not support.

5.3 An earlier reproduction exercise (v0.8.0)

The system's first operational end-to-end exercise predates v0.9.0: the then-current v0.8.0 pipeline was run against a published lossy-compression target, a GPU error-bounded scientific compressor, through the reproduction path of section 4.6. The degree of reproduction achieved on this single target was partial but substantive. The agent fetched a real slice of the paper's cited simulation dataset rather than substituting synthetic data; compiled native CUDA interpolation kernels and ran them on the GPU; implemented Lorenzo and multi-level interpolation predictors with per-level autotuning; and swept a range of error bounds, recovering reconstruction qualities from roughly 50 to 90 dB peak signal-to-noise ratio (PSNR) with the corresponding compression ratios. Against the fine-grained rubric the run cleared on the order of a tenth of the weighted leaves, for two reasons rooted in the agent's behaviour rather than the pipeline's: the graded metric came from a CPU predictor — the GPU kernels were built and executed, but the agent omitted their predicted quality from its reported results — and the agent terminated voluntarily after a small fraction of its time budget, leaving the remaining datasets and the proprietary lossless stage unattempted. Both behaviours match failure modes documented for replication agents at large, where most models end runs early claiming completion and score far higher on writing code than on executing it and matching the paper's results [9]. The exercise predates the verification machinery of section 5.1 and is reported unchanged, as the historical baseline.

5.4 Limitations

The evaluation remains preliminary. Neither result above is a controlled benchmark: the generation run is a single trajectory, and the reproduction exercise covers a single target paper. A controlled evaluation — median run cost, per-seed exploration curves, and per-category rubric distributions over a frozen checkpoint, together with full-rollout replication scores across a panel of contrasting target papers — is left to future work. The same study should settle the empirical questions this report leaves open, such as which termination cause (section 3.5) fires most often in practice.

6 Related Work

We position ARI against four threads of prior art.

6.1 Tree search for code

AIDE [5] frames machine-learning engineering as code optimisation — a solution is a script scored by a stateless objective — and casts trial-and-error as search over a tree of scripts joined by draft, debug, and improve edges. On MLE-bench [6], 75 offline Kaggle competitions graded against real medal thresholds, this scaffold dominates: the same model earns medals in 8.7% of competitions under AIDE against 0.8% and 4.4% under two general-purpose scaffolds, so scaffold design outweighs raw model capability. ARI inherits the solution-tree skeleton but replaces AIDE's hard-coded greedy policy — flagged by its own authors as a local-optima risk — with an LLM-driven selector (section 3.3) that weighs `has_real_data`, full `metrics`, depth, and a soft `diversity_bonus` jointly, and adds the cross-run lineage of section 3.7. And where both works receive their objective with the task — real research, MLE-bench's authors concede, "may not even have a clear problem statement" — ARI mints its own: the metric contract (section 4.3) declares before search what counts as evidence.

AlphaCode [10] is the canonical generate-and-filter result: up to millions of sampled programs per problem, filtered on the task's example tests and clustered to at most ten submissions, reaching an average ranking in the top 54.3% of participants over ten simulated Codeforces contests. The recipe presupposes a cheap, task-supplied verifier; an ARI candidate is an executed experiment on cluster resources with no given verifier, so breadth gives way to best-first steering of a small frontier, and the cheap filter to deterministic checks on the run's own evidence.

6.2 Autonomous research agents

The AI Scientist family [2, 3] closes a similar loop end-to-end (idea → experiment → paper) at a reported cost below $15 per paper, with an automated reviewer of near-human balanced accuracy on real conference submissions. But the reviewer is *advisory* — it ranks

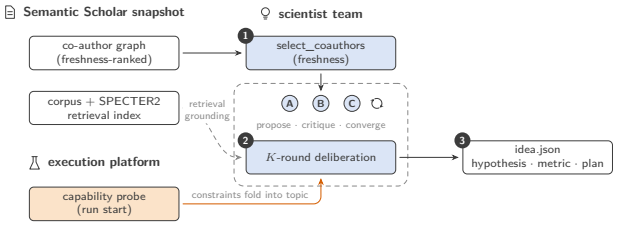

Figure 18: VirSci-style idea generation as ARI drives it. `select_coauthors` forms a heterogeneous scientist team from a frozen Semantic Scholar snapshot (corpus with SPECTER2 retrieval index, freshness-ranked co-author graph); the team runs a K-round deliberation grounded by retrieval against the snapshot, and a run-start capability probe folds the platform's measured constraints into the deliberation topic, so the converged `idea.json` promises only measurements the platform can take.

finished PDFs, and hallucination control is prompt-level — and the authors, cataloguing hallucinated hardware, fabricated ablation tables, and a negative result narrated as an improvement, name automatic verification of results as future work. ARI is the contrapositive: experiments branch under BFTS instead of one linear trajectory per idea, the metric is a run-level artefact frozen at first mint (section 4.3), and verification blocks — a deterministic claim–evidence gate, not a reviewer score, decides what may be published (section 4.8).

Agent Laboratory [4] treats the agent as a collaborator rather than a driver: it requires a human-provided research idea, mutates a flat two-program pool under an LLM reward model scoring plan adherence, and gates papers on simulated reviewers. Its most instructive result is negative: those reviewers overestimate human reviewers by 2.3 points on a ten-point scale — the clearest quantification of why holistic LLM-judge gating is insufficient. ARI's semantic review therefore steers but never blocks, blocking is reserved for the deterministic gate, and every node persists in a checkpoint-scoped tree rather than a transient pool.

VirSci [7] covers the stage these systems treat most thinly: a scientist team grounded in a digital twin of a research community, whose multi-agent deliberation improves on a single-agent baseline by 13.8% and 44.1% in alignment with and potential impact on contemporary research. It stops at an abstract scored by embedding-proximity novelty proxies, and its authors call for stricter downstream validation. ARI takes both halves of that division of labour: it drives VirSci's original deliberation engine directly and unmodified (fig. 18) — freshness-based `select_coauthors` team formation and the multi-agent `generate_idea` loop — over a live Semantic Scholar snapshot, so the mechanism it builds on is the published one executed against current literature; and it supplies the deferred validation downstream, where executed evidence under the claim–evidence gate replaces proxy novelty scores.

6.3 Paper evaluation

PaperBench [9] pioneered the rubric-as-tree formalism: each of its 20 target papers carries an author-co-developed rubric — 8,316 gradable leaves in total — whose leaf grades are pass/fail and aggregate by weighted average, with a clean-start reproduction phase so hard-coded results cannot masquerade as replication. ARI vendors the benchmark unchanged (section 4.6), adopts the same per-leaf grade and aggregate weight (eq. (8)), and adds a per-axis floor used as a refine guard (section 4.8). Its execution gap — the strongest agent scores 43.3% on code development but 4.5% on execution and zero on matching results — motivates moving verification inside generation rather than grading after the fact; and its hand-built rubrics (weeks per paper, with the original authors) leave automated rubric construction as future work, the branch ARI takes: rubrics are generated per paper (section 4.4), so handcrafted-rubric reproduction and leaderboard-comparable scores are out of scope by design.

Story2Proposal [8] governs structured manuscript generation through a persistent shared contract — section structure, a registry of visual artifacts, document-wide validation rules — with writer, refiner, and renderer agents under a generate–evaluate–adapt loop scored on reasoning verification, data fidelity, and visual coherence. The gain holds across four LLM backbones, but the evidence is taken as given, and its failure-mode analysis concedes a manuscript can remain "structurally valid while containing less rigorous reasoning": the contract binds structure, not evidentiary validity. ARI carries the principle and its axes into an execution-grounded pipeline (section 4.2): the contract extends the experiment record with claims and numeric assertions referencing stable node, result, and figure identifiers; data fidelity becomes the deterministic gate that re-derives each reported number from recorded results; reasoning and visual coherence become a non-blocking evidence-grounded review beside the independent manuscript review. Executing the experiments itself, ARI extends the manuscript-level contract to execution-grounded provenance.

6.4 Foundations

The per-node agent loop (section 3.6) follows the Reason-and-Act (ReAct) pattern [1], whose precise finding is a trade: interleaving thoughts with grounded actions cuts hallucination sharply while raising the reasoning-error rate. The refine cycle descends from Self-Refine [11] and Reflexion [12], each with a caveat ARI takes seriously: Self-Refine's gains vanish where the model cannot detect its own errors, and Reflexion's ablations show grounded evaluation — not memory alone — drives its improvement. ARI therefore never lets a model grade itself on matters of fact: refinement is steered by rubric scores and gate verdicts computed outside the writer, and a result becomes a quantitative claim only when artefact-grounded (section 3.10). The scratchpad in *Think* descends from chain-of-thought prompting [13], whose gains are reported as emergent with model scale — its authors never claim the model reasons. None of these proposes state beyond a single trajectory; the checkpoint-scoped state that makes a run reproducible is ARI's contribution.

An earlier draft cited a further research-agent benchmark; the reference was removed when no candidate source passed the bibliography verification rule of section 4.8, so coverage here is restricted to entries verified against the Semantic Scholar index.

7 Discussion and Limitations

7.1 Determinism vs novelty

P2, the determinism principle, is the binding constraint of the system. It constrains not whether the research memory (section 3.10) may use an embedding model but *where* the resulting non-determinism may act: the memory makes no generative language-model calls, and its embedding-ranked archival search — dependent on a remote service, hence not byte-reproducible — stays off the reproducible path as an optional recall aid the loop never pulls, unable by itself to ground a paper claim. The memory reads the loop does depend on — the working context injected at each node, the verified context that grounds the paper — are deterministic, ancestor-scoped folds of per-node reports. Every reproducibility win closes such a door to non-determinism; we accept the cost because the checkpoint, not the run, is the contract with downstream users.

7.2 Instruction-following at structural seams

The recurring lesson of v0.9.0 is that instruction-following degrades at structural seams — an index to parse, a name to reuse, a number to declare — and that the durable remedy is parser-side absorption, not prompting. An unparseable selection index falls back to a deterministic ranking (section 3.3); a model asked twice names the same quantity differently, so the metric contract is minted once and replayed verbatim; the writer's recurring numeric-declaration quirks are normalised at parse time (section 4.3). The catalogue of absorbed quirks is reactive, grown from real runs, so an unanticipated variant first surfaces as a gate mismatch — conservatively so: it is flagged, an unresolved mismatch blocks the final check, and nothing is silently accepted.

The same discipline fixes what may block. The gate certifies that every quantitative claim matches a recorded artefact, while the semantic review's subjective findings steer the refiner (section 4.5) but never gate publication — unlike the wholly post-hoc review of [2] — because a subjective verdict on the blocking path would be an unauditable veto. A verified paper can therefore still be a dull one.

7.3 Scaling limits

The single-machine assumption underlying the BFTS run-loop is the biggest scaling constraint: a single `ThreadPoolExecutor` (section 3.2) is the sole worker pool, capped at four workers. Multi-host expansion would require extending the explicit-lineage discipline (P4; section 2.5) to merges, not just ancestor walks — the same gap that bounds computed evidence, whose chains (section 3.8) are scheduled today by name hints along a single lineage rather than by an explicit scheduler over the evidence dependency graph. A separate constraint is the LLM grader, empirically a dominant cost item alongside the explorer; caching its output by (paper-hash, rubric-hash) would help and has not yet shipped.

7.4 Outlook

The widest gap is evaluation breadth: section 5 rests on one reproduction exercise (run on the then-current v0.8.0 pipeline) and one verified generated paper — evidence that the machinery composes, not a distribution. Future work is threefold: a controlled study over a frozen checkpoint and a panel of contrasting target papers, in the protocol spirit of [6, 9]; variance-aware aggregation over repeated rollouts and grader passes in place of single-scalar roll-ups, the judges themselves being noisy; and broader domains — widening the evidence kinds the contract can name beyond scalar measurements, as the execution-profile contract (section 4.7) widened the computations ARI can host, toward fields whose primary evidence is not a single metric.

A Notation Table

This appendix collects every symbol used in the body of the report, with its meaning.

Symbol	Meaning
$\mathcal{N}$	set of nodes in the search tree
$\mathcal{T}$	search tree $(\mathcal{N}, E)$
$n \in \mathcal{N}$	individual node
$\mathrm{ch}(n)$	children of n
$\mathrm{pa}(n)$	parent of n
$\mathrm{depth}(n)$	depth of n
$s(n)$	node state $\in \{\mathrm{open}, \mathrm{eval}, \mathrm{done}, \mathrm{failed}\}$
$r(n)$	scalar reward of n (its `_scientific_score` $\in [0, 1]$)
$\mathbf{e}(n)$	per-axis evaluation vector
ϕ	axis $\to$ scalar aggregator
$\mathrm{div}(n)$	diversity bonus (eq. (4))
$\mathrm{fb}(n)$	deterministic fallback ranking (eq. (5))
$T_{\max}, C_{\max}$	step / cost budgets
R	rubric tree
c_i	rubric leaf i's category
w_i	rubric leaf i's weight
judge_i	rubric leaf i's judge
P	paper draft
$\mathrm{score}(P)$	paper aggregate score (eq. (8))
Refine	refine operator (eq. (9))
ckpt	checkpoint directory
$\mathrm{lin}(n)$	lineage chain ending at n

B Runtime LLM Prompts

This appendix lists the LLM prompts that ARI uses at runtime, reproduced verbatim from `ari-core/ari/prompts/` at v0.9.0. Each listing is byte-for-byte identical to the source on disk. Because the exact bytes are what reach the model, the prompts are reproduced as-is in every language version of this report and are not translated.

Orchestrator

B.1 BFTS expansion

```
You are expanding a BFTS research tree node.

{goal_line}Parent node id={parent_id_short}, depth={parent_depth}, status={parent_status}
{depth_note}{budget_note}Parent metrics: {parent_metrics_json}
Parent summary: {parent_summary}
{sci_note}{idea_block}{parent_report_block}
{siblings_block}{ancestors_block}{existing_block}{diversity_block}Propose exactly ONE child research direction
that is the most scientifically valuable next step. For the "label" field, strongly prefer one of the canonical
labels [draft, improve, debug, ablation, validation]; only invent a custom label (e.g. "replication",
"generalization") when none of the five fits meaningfully - the custom string will be preserved as raw_label.
Base your choice on the experimental context above, not on a fixed template.

Label selection guidance:
- 'debug': parent FAILED or has no real data - diagnose and fix it.
- 'improve': parent succeeded and you want to push its metrics higher by tuning parameters, flags, or
algorithms.
- 'ablation': isolate which component drives the parent's gains by removing or varying ONE component. State
explicitly what is removed/varied and what delta vs. the parent metrics you expect.
- 'validation': rigorously verify the parent's claims (different seeds, edge cases, stress tests,
expected-degradation checks).
- 'draft': start a fresh implementation from scratch to introduce a fundamentally NEW perspective (use this
instead of inventing a new label like 'replication' or 'generalization').

Reply ONLY with a JSON array containing exactly one element: [{"label": "<one of:
draft|improve|debug|ablation|validation>", "direction": "..."}]
Example: [{"label": "validation", "direction": "<one-sentence plan>"}]
```

B.2 BFTS expansion-time selection

```
You are selecting which completed research node to expand next in a BFTS tree.

Experiment goal: {experiment_goal}

Completed nodes awaiting expansion:
{candidates}

Select the single most promising node to expand. Prefer nodes with high scientific_score, strong metrics, and
unexplored directions. Avoid nodes that have already been retried many times or are at excessive depth.
Reply with ONLY the index number (0-based).
```

B.3 BFTS selection

```
You are selecting the most promising node to explore next in a research tree.

Experiment goal: {experiment_goal}

Relevant past memories:
{memory_context}

Candidates:
{candidates}

Selection criteria:
- Nodes with has_real_data=True and strong metrics are high-value
- Consider all metrics holistically (multi-objective)
- Deeper nodes with excellent results are worth continuing
- A small diversity_bonus is awarded to underrepresented exploration types; treat it as a soft tiebreaker, not
a primary signal
Reply with ONLY the index number (0-based) of the best node.
```

B.4 Lineage continuation decision

```
You are a research orchestrator. Given the current state of an exploratory research run, decide the next
lineage-level action. Possible actions:
  continue       - current idea is still productive; keep exploring
  switch_to_idea - current idea has stagnated; switch to an alternative
  fanout         - current idea succeeded; explore an alternative in parallel
  terminate      - research thread is exhausted; stop
Choose the LEAST disruptive action that fits the evidence. Prefer continue unless there is a concrete reason to
escalate. When choosing switch_to_idea or fanout, set target_idea_index to a value that appears in the
alternatives pool. When choosing switch_to_idea, set disable_generate_ideas=true (the child should run with the
chosen idea pinned, not regenerate). For fanout you may set it false so the child can also explore additional
novel directions.
Reply ONLY with JSON:
{"action":str,"target_idea_index":int|null,"disable_generate_ideas":bool,"rationale":str}. No markdown.
```

B.5 Root idea selector

```
You are a research orchestrator picking the ROOT idea for a run from a VirSci-generated pool. VirSci has
already scored each idea by novelty/feasibility/clarity, but you have additional context (venue rubric,
ancestor research thread, run notes). Your job is to pick the idea most likely to produce a strong submission
for this venue, considering all signals.

Rules:
  - chosen_index MUST be an integer that exists in the pool.
  - Default to VirSci's choice (index 0) UNLESS another idea is     clearly stronger given the additional
  context.
  - One sentence rationale describing your reasoning.
Reply ONLY in JSON: {"chosen_index": int, "rationale": str}. No markdown.
```

Agent loop

B.6 ReAct agent system prompt

```
You are a research agent. You MUST use tools to execute experiments. Do NOT write plans or text descriptions -
call a tool immediately.

AVAILABLE TOOLS:
```

```
{tool_desc}

RULES:
- Your FIRST action must be a tool call. Never output a text plan.
- If `make_metric_spec` tool is available and this is a new experiment (not a continuation), call it early to
self-determine evaluation criteria.
- NEVER fabricate numeric values - only report values from actual tool outputs
- AFTER your final measurement run, when `emit_results` is available, call it once to record a typed split
between INPUT parameters (matrix size, thread count, seeds - knobs you ran on) and MEASUREMENTS (throughput,
accuracy, latency - what you measured). This lets downstream stages distinguish "what we ran on" from "what we
measured" so a best-of reduction never picks an input size as the result. Do NOT include input parameters in
`measurements` and do NOT include measured outputs in `params`.
- When all experiments are done, return JSON: {{"status": "success", "metrics": {{...}}, "summary": "..."}}
- Do NOT call gap_analysis or generate_hypothesis
- Ensure your experiment is reproducible: capture whatever information would be needed for an independent
researcher to reproduce your results and verify your findings{memory_rules}{extra}
```

Evaluator

B.7 Peer-review evaluator

```
You are a research data extractor AND a scientific peer reviewer.
Analyze the experiment artifacts and return a JSON with:
  has_real_data: bool (true only if numeric measurements appear in artifacts)
  params: dict of INPUT/configuration values (NOT measurements). Examples: matrix size, thread count, seed.
  measurements: dict of MEASURED quantities (the experiment's output). Examples: GFLOP_per_s, accuracy,
  latency.
  metrics: dict - flat union of params and measurements (back-compat).
  reason: str (one sentence describing what was measured)
{axes_block}
  comparison_found: bool (true if results involve comparison with existing approaches)

When scoring claim_implementation_alignment, look for concrete preconditions or contracts the plan / model
states (e.g. 'eliminates RFO traffic', 'preserves equivariance', 'requires sorted input') and cross-reference
them against the artifacts. Score low when the implementation contradicts a stated assumption, even if the
kernel runs without errors.
Return ONLY valid JSON, no markdown fences.
```

B.8 Metric extraction

```
You are a research data extractor AND a scientific peer reviewer.
Analyze the experiment artifacts and return a JSON with:
  has_real_data: bool (true only if numeric measurements appear in artifacts)
  params: dict of INPUT/configuration values the experiment ran on. These are knobs, not outcomes. Examples:
  matrix dimensions (M, K, nnz), thread count, batch size, ISA flags, seeds. They are NOT candidates for a
  best-of reduction.
  measurements: dict of MEASURED quantities - what a peer reviewer would treat as the experiment's result.
  Examples: GFLOP_per_s, GB_per_s, latency_s, accuracy. ARE candidates for best-of.
  metrics: dict - for back-compat, the union of params and measurements as a single flat dict (e.g.
  {{"GFLOP_per_s": 754.8, "M": 120000}}). Downstream consumers may read either the typed split above or this
  flat view. NEVER include the same key in both views with different values.
  reason: str (one sentence describing what was measured)
  axis_scores: dict with EXACTLY these five keys, each a float in 0.0-1.0:
    - measurement_validity: are numeric measurements present and methodologically sound?
    - comparative_rigor: are results compared against baselines or prior work?
    - novelty: does the work advance beyond existing approaches?
    - reproducibility: is there enough detail for someone else to reproduce the result?
    - clarity_of_contribution: is the scientific claim stated clearly and specifically?
    Score 0.0 on an axis when that dimension is absent or fatally weak.     Score toward 1.0 as the dimension
    approaches publishable quality.     These axes are combined via weighted harmonic mean, so a low score on
    ANY axis     drags the overall score down - differentiate axes deliberately.
  axis_rationales: dict with the same five keys; each value is one sentence     explaining the score for that
  axis.
  comparison_found: bool (true if results involve comparison with existing approaches)
Return ONLY valid JSON, no markdown fences.
```

Pipeline

B.9 Keyword librarian

```
You are a research librarian. Given experiment summaries, produce a SHORT broad academic search query (3-6
words) for Semantic Scholar. Use GENERAL terms (e.g. 'algorithm optimization', not 'custom 4-stage pipelined
batch-norm fused kernel'). Avoid domain-specific jargon, acronyms, or overly narrow terms. Return ONLY the
query string, no explanation.
```

Wizard (Viz)

B.10 Wizard goal chat

```
You are an assistant helping a researcher set up an ARI experiment. ARI automatically writes, runs, benchmarks
code, then produces a paper. Ask focused questions ONE AT A TIME to understand: (1) what to optimize or
investigate, (2) how to measure success (metric name and direction), (3) platform/hardware constraints, (4) any
baseline to compare against. Be concise. After 3-5 exchanges and sufficient info, output EXACTLY: ---READY---
followed by a complete experiment.md with sections: ## Research Goal, ## Evaluation Metric, ## Constraints. Do
NOT output the MD before getting at least 2 user responses.
```

B.11 Wizard config generator

```
You are helping a researcher set up an automated experiment. Convert the following research goal into a concise
experiment.md file with a ## Research Goal section. Keep it to 3-5 sentences maximum. Do not add code or
technical details. Research goal: {goal}
```

C The Generated Sample Paper

This appendix reproduces, in full and unedited, the eight-page paper that the system generated autonomously in the case-study run of section 5.1. It is included as an artefact: every claim it asserts passed the deterministic claim–evidence gate, and the text hedges explicitly wherever a statement would outrun its recorded evidence. Hardware is described at the instruction-set level only; the single product name that appears does so inside a cited paper's title, not as a description of the run environment.

CSR Sparse-Dense Matrix Multiplication on CPUs with RHS-Width Robustness and a Loopline-Guided Performance Model

Autonomous Research Infrastructure

June 12, 2026

Abstract

Sparse-dense matrix multiplication (SpMM) in CSR format is a core kernel in scientific computing and modern sparse learning pipelines, yet its performance on CPUs is highly sensitive to the width k of the dense right-hand side (RHS) matrix. We present a CSR SpMM implementation with a cache-aware kernel structure and a store-policy selector intended to switch between regular temporal stores and non-temporal stores as k varies. We validate correctness against an independent dense FP32 reference, achieving a maximum absolute error of 6.78958×10^{-6}.

1 Introduction

Sparse-dense matrix multiplication (SpMM), $C \leftarrow AB$ with sparse A and dense B, is widely used in simulation codes, signal processing, and learning workloads. In CSR SpMM, the irregular access pattern induced by the sparse structure interacts with cache hierarchies and store buffers; a particularly important practical issue is that performance can change drastically when the RHS width k changes (e.g., small k behaves like SpMV, while larger k exposes more reuse in B). This sensitivity complicates performance portability and makes it difficult to choose optimizations such as non-temporal stores and blocking strategies.

This paper targets a CPU implementation of CSR SpMM with a cache-aware kernel structure and a store-policy selector intended to adapt to varying k, and it outlines a measurement methodology intended to support future performance modeling (e.g., identifying bandwidth- vs. compute-limited regimes). Our work is motivated by the broader importance of sparse matrix kernels Dorrance et al. [2014], Alappat et al. [2020] and by recent interest in high-performance SpMM on modern Arm CPUs Lei et al. [2025].

Contributions.

- We implement a CSR SpMM kernel and a store-policy selector intended to choose between regular temporal stores and non-temporal stores as a function of k (Figure 1 illustrates the design goal).
- We provide a reproducible measurement methodology that includes correctness validation against an independent dense FP32 reference, plus optional microbenchmarks (bandwidth probes and a compute-inflation probe) intended to support future performance modeling.
- We provide an experimental setup for collecting artifacts on aarch64 and x86_64 to support future characterization and model validation Alappat et al. [2020], Dufek et al. [2021].

2 Related Work

Sparse matrix kernels have a long history of architecture-aware optimization, where performance is often limited by memory bandwidth and irregular access rather than peak FLOPs. Foundational studies of sparse kernels emphasize scalable implementations and the role of memory systems Dorrance et al. [2014]. On recent aarch64 HPC systems, performance modeling for streaming kernels and sparse operations highlights the importance of measuring attainable bandwidth ceilings and accounting for architectural details beyond nominal peak values Alappat et al. [2020]. Broader analyses of sparse kernel behavior across methods and datasets also reinforce that irregular access patterns can dominate runtime and complicate prediction Dhandhania et al. [2021].

For sparse matrix-matrix multiplication, modern work explores format choices and heterogeneity-aware strategies Cheng et al. [2023], Namjoo et al. [2026]. While much of this literature focuses on SpGEMM or GPU/heterogeneous settings, the underlying challenge of selecting appropriate kernels and data layouts remains central on CPUs, particularly as k changes. Recent work on Arm architectures argues that careful microarchitectural tuning is required for high SpMM performance Lei et al. [2025]. Our work complements these efforts by focusing on implementing mechanisms intended to improve k-robustness in CSR SpMM (notably a store-policy selector), and by providing a lightweight measurement methodology intended to support future model development and validation.

3 Methodology

We compute $C \in \mathbb{R}^{M \times k}$ from a CSR matrix $A \in \mathbb{R}^{M \times K}$ and a dense RHS $B \in \mathbb{R}^{K \times k}$:

$$C_{i,:} \leftarrow \sum_{p=\texttt{rowptr}[i]}^{\texttt{rowptr}[i+1]-1} \texttt{val}[p] \cdot B_{\texttt{col}[p],:}. \tag{1}$$

The implementation is parallelized over CSR rows using OpenMP with a fixed thread count.

3.1 CSR SpMM kernel

The baseline kernel follows the standard CSR SpMM structure: for each row i, iterate over the nonzeros, gather the corresponding RHS row of B, and perform a scaled vector add into $C_{i,:}$. The critical design choice for k-robustness is how the kernel initializes and stores C, because C is dense and potentially large; depending on k, write-allocate and cache pollution can dominate.

We consider two store policies:

- **Regular temporal stores**: standard writes to C, benefiting from cache when reuse exists.
- **Non-temporal stores**: streaming stores to reduce cache pollution and avoid write-allocate traffic for large k.

A selector chooses the policy based on an empirical tuning procedure over k (Section 3.4); in this revision we describe the mechanism, but do not claim artifact-grounded crossover values.

3.2 Correctness validation

Correctness is validated by comparing the CSR SpMM output against an independent dense reference implementation (dense A constructed from CSR) in FP32. FP64 validation is an optional

check when enabled, but FP64 error results are not reported as artifact-grounded outcomes in this revision. The reported metric is the maximum absolute elementwise error:

$$\epsilon_{\max} = \max_{i,j} \left| C_{ij}^{(\text{spmm})} - C_{ij}^{(\text{ref})} \right|.$$

3.3 Microbenchmarks and ceilings

To parameterize the performance model, we measure:

- **DRAM memcpy bandwidth** and related in-code memory ceilings (used as bandwidth ceilings).
- **STREAM triad bandwidth** as an additional indicator of sustainable bandwidth.
- **Compute inflation probe**: an artificial increase in arithmetic intensity by adding a fixed number of fused multiply-add operations (FMAs) per element of B. This yields an estimate of an effective compute rate f_{eff} (FLOP/s) that can be used as a compute ceiling.

This style of combining attainable bandwidth measurements with compute ceilings follows the spirit of roofline/loopline modeling Alappat et al. [2020], Dufek et al. [2021].

3.4 Store-policy selector

For a fixed sparse matrix A and varying k, we benchmark both store policies and identify an empirical crossover k at which non-temporal stores become faster than regular stores. The selector then chooses:

$$\texttt{policy}(k) = \begin{cases} \texttt{regular} & k < k^{\cdot} \\ \texttt{non-temporal} & k \geq k^{\cdot} \end{cases}$$

Figure 1 visualizes the speedup trend and the crossover behavior.

3.5 Reproducible pseudocode

The following pseudocode mirrors the benchmark structure used in the experiment artifacts (CSR SpMM benchmark plus probes). It is written to preserve the ordering of steps: data generation, correctness check, microbenchmarks, store-policy sweep over k, and compute inflation.

Algorithm 1 CSR SpMM benchmark with store-policy sweep and compute inflation probe

Inputs: thread count T; CSR dimensions (M, K); `nnz_per_row` or `nnz`; RHS widths $k \in \{8, 16, 32, 64, 128\}$; warmup w; iterations r; seeds
Set `OMP_NUM_THREADS` $\leftarrow T$
Generate CSR matrix A with fixed `nnz_per_row`: for each row i, sample `nnz_per_row` column indices and values using the given seed; build `rowptr`, `col`, `val`
Generate dense RHS $B \in \mathbb{R}^{K \times k}$ with pseudo-random values (seeded)
Allocate $C \in \mathbb{R}^{M \times k}$
Correctness check:
Build dense reference A_{dense} from CSR (or compute reference directly) and compute $C_{\text{ref}} \leftarrow A_{\text{dense}} B$
Compute $\epsilon_{\max} \leftarrow \max |C - C_{\text{ref}}|$ for FP32; optionally repeat in FP64 when enabled (FP64 error is not reported as a grounded result in this revision)
Bandwidth microbenchmarks:
Measure DRAM `memcpy` bandwidth and (optionally) `memset` bandwidth on arrays sized to exceed cache
Measure STREAM triad bandwidth
Store-policy timing sweep:
for each k in $\{8, 16, 32, 64, 128\}$ **do**
    Regenerate/resize B and C for this k
    Time CSR SpMM with **regular temporal stores**: run w warmup iterations, then average over r iterations
    Time CSR SpMM with **non-temporal stores**: run w warmup iterations, then average over r iterations
end for
Determine empirical crossover k where non-temporal becomes faster; record speedup curve
Compute inflation probe:
Choose list of added FMAs per B element (e.g., $[0, 2, 4, 8]$ or $[0, 4, 8, 16]$)
for each α in added-FMA list **do**
    Run CSR SpMM kernel variant that adds α FMAs per B element inside the innermost accumulation loop
    Record runtime $t(\alpha)$
end for
Fit effective compute rate f_{eff} from the slope of $t(\alpha)$ vs. added work

4 Experiments and Results

4.1 Setup

Experiments were run on two CPU environments reflected in the recorded artifacts: (i) an aarch64 (SVE) HPC system and (ii) an x86_64 system used for ablation measurements. The main benchmark uses OpenMP with a fixed thread count.

Main benchmark configuration (headline). For the headline CSR SpMM measurement, we use:

- Threads: $T = 48$.

- Sparse matrix: $M = N = 2048$, $K = 64$, `nnz_per_row`$= 32$, CSR format.
- RHS: dense, with RHS width $k = 64$ in this configuration.
- Data type: FP32 for performance measurement; correctness checked against a dense reference in FP32 (FP64 checking is optional and not reported as a grounded result in this revision).

Validation configuration (model/probe suite). A separate validation suite averages results across seeds $\{1, 2, 3\}$ and includes: DRAM bandwidth microbenchmarks, a store-policy sweep over $k \in \{8, 16, 32, 64, 128\}$, and a compute-inflation probe with $\alpha = 8$ added FMAs per B element. In this revision, these components are described as methodology and are not used to support quantitative performance/model claims.

Ablation configuration (x86_64). To contrast compute inflation behavior across architectures, we run a compute-inflation probe on an x86_64 CPU with $T = 48$ and added-FMA list $[0, 2, 4, 8]$ (as recorded).

4.2 Results

We report only artifact-backed quantitative results. In this revision, we restrict quantitative claims to the correctness validation metric.

Correctness. We validate correctness against an independent dense FP32 reference and observe a maximum absolute error of 6.78958×10^{-6}.

Headline SpMM throughput and FLOP rate. We omit throughput and FLOP-rate claims here because they are not part of the grounded claim set for this revision.

Validation-suite end-to-end performance. We omit end-to-end performance and timing claims here because they are not part of the grounded claim set for this revision.

Store-policy crossover with increasing RHS width. Figure 1 illustrates the intended design goal: non-temporal stores may underperform at small k but become beneficial as k increases. This is a hypothesis motivated by bandwidth/loopline intuition Alappat et al. [2020], not an artifact-grounded result in this revision. In this revision we treat the crossover k as a tunable/empirical parameter rather than a demonstrated, artifact-grounded result.

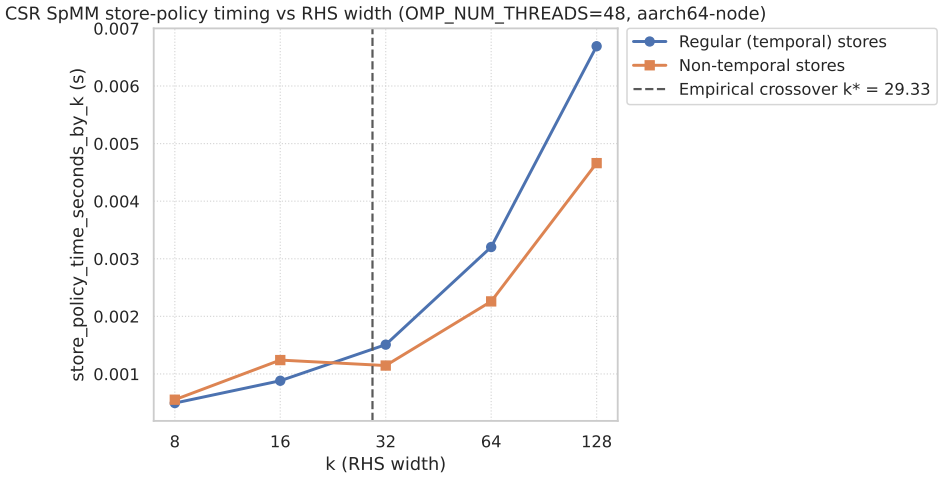

Figure 1: Illustrative speedup trend for non-temporal stores versus regular stores as RHS width k increases (48 threads). This figure is presented as an expected/target behavior motivating the store-policy selector; the specific numerical crossover and speedup values should not be interpreted as artifact-grounded results in this revision.

Bandwidth ceilings and their role. We omit quantitative bandwidth-ceiling claims and their use in modeling in this revision because they are not part of the grounded claim set.

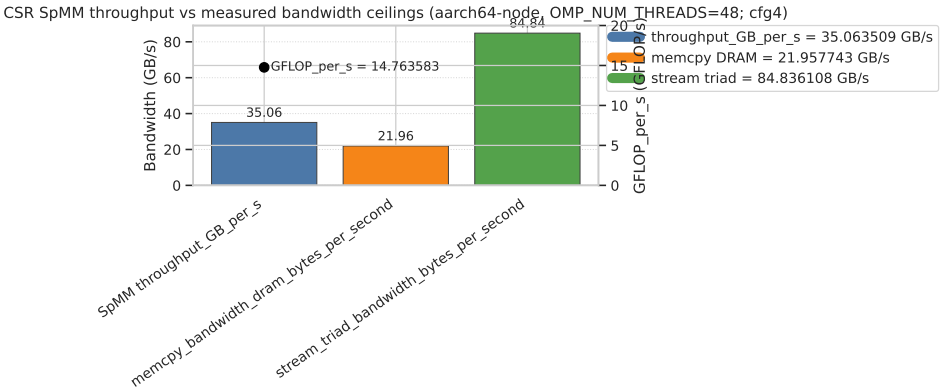

Figure 2: Measured DRAM `memcpy` bandwidth (GB/s) across configurations. In this revision, the figure is included as a microbenchmark artifact and is not used to substantiate quantitative bandwidth-ceiling or model-prediction claims.

Compute inflation and effective compute ceiling. Figure 3 illustrates the compute-inflation probe conceptually. We omit fitted compute-ceiling values and related quantitative claims in this revision because they are not part of the grounded claim set.

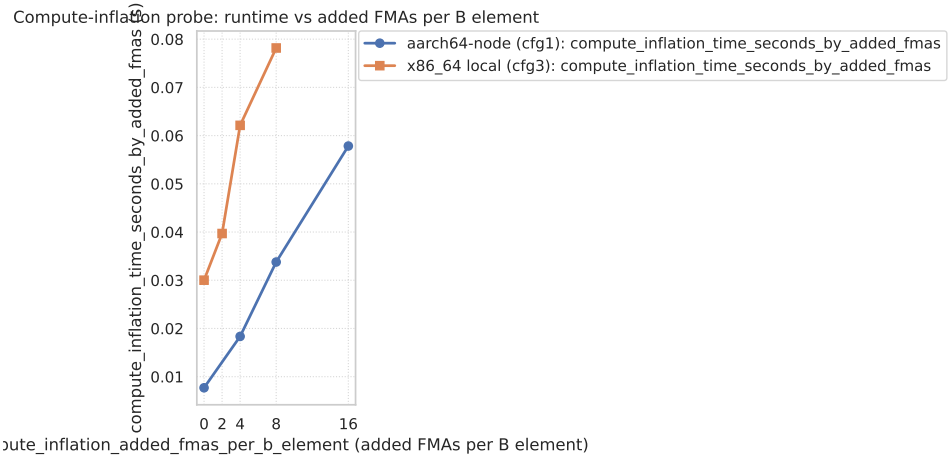

Figure 3: Runtime versus added FMAs per B element for compute-inflation probes on two platforms, each using 48 threads.

5 Conclusion

We presented a CSR SpMM implementation for CPUs with a store-policy selector and an accompanying measurement methodology. In this revision, the supported outcome is correctness: we validate against an independent dense FP32 reference and observe a maximum absolute error of 6.78958×10^{-6}. Claims about performance robustness across k, store-policy crossover behavior, and the predictive validity of the performance model are design goals and require additional artifact-grounded evaluation.

Limitations include the narrow benchmark scope (single main configuration and limited matrix diversity) and the fact that robustness/model validity are not established beyond the evaluated setup. Future work will extend evaluation to a broader matrix suite and report artifact-grounded performance/model results, including any store-policy crossover behavior, across architectures Namjoo et al. [2026].

References

C. Alappat, Jan Laukemann, T. Gruber, G. Hager, G. Wellein, N. Meyer, and T. Wettig. Performance modeling of streaming kernels and sparse matrix-vector multiplication on a64fx. *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 1–7, 2020.

Helin Cheng, Wenxuan Li, Yuechen Lu, and Weifeng Liu. *HASpGEMM: Heterogeneity-Aware Sparse General Matrix-Matrix Multiplication on Modern Asymmetric Multicore Processors*. 2023.

Sunidhi Dhandhania, A. Deodhar, Konstantin Pogorelov, Swarnendu Biswas, and J. Langguth. *Explaining the Performance of Supervised and Semi-Supervised Methods for Automated Sparse Matrix Format Selection*. 2021.

R. Dorrance, Fengbo Ren, and D. Markovic. *A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs*. 2014.

A. S. Dufek, J. Deslippe, P. Lin, Charlene Yang, B. Cook, and Jonathan Madsen. An extended roofline performance model with pci-e and network ceilings. *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 30–39, 2021.

Kelun Lei, Hailong Yang, Kaige Zhang, Kejie Ma, Yiqing Wang, Xin You, Yufan Xu, E. Quintana-Ortí, Zhongzhi Luan, Yi Liu, and Depei Qian. Low-cost yet high-performant sparse matrix-matrix multiplication on arm sme architectures. 2025.

A. Namjoo, Sanjali Yadav, Helya Hosseini, and Bahar Asgari. Situla: Studying the interplay of sparse formats and cpu/gpu libraries. *2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 567–577, 2026.

References

[1] Shunyu Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. arXiv: 2210.03629 [cs.CL]. URL: https://arxiv.org/abs/2210.03629.

[2] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. *The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery*. 2024. arXiv: 2408.06292 [cs.AI]. URL: https://arxiv.org/abs/2408.06292.

[3] Chris Lu et al. "Towards End-to-End Automation of AI Research". In: *Nature* 651.8107 (2026), pp. 914–919. DOI: 10.1038/s41586-026-10265-5. URL: https://doi.org/10.1038/s41586-026-10265-5.

[4] Samuel Schmidgall et al. *Agent Laboratory: Using LLM Agents as Research Assistants*. 2025. DOI: 10.18653/v1/2025.findings-emnlp.320. arXiv: 2501.04227 [cs.AI]. URL: https://arxiv.org/abs/2501.04227.

[5] Zhengyao Jiang et al. *AIDE: AI-Driven Exploration in the Space of Code*. 2025. arXiv: 2502.13138 [cs.AI]. URL: https://arxiv.org/abs/2502.13138.

[6] Jun Shern Chan et al. *MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering*. 2024. DOI: 10.48550/arXiv.2410.07095. arXiv: 2410.07095 [cs.LG]. URL: https://arxiv.org/abs/2410.07095.

[7] Haoyang Su et al. *Many Heads Are Better Than One: Improved Scientific Idea Generation by a LLM-Based Multi-Agent System*. 2024. DOI: 10.18653/v1/2025.acl-long.1368. arXiv: 2410.09403 [cs.CL]. URL: https://arxiv.org/abs/2410.09403.

[8] Zhuoyang Qian et al. *Story2Proposal: A Scaffold for Structured Scientific Paper Writing*. 2026. arXiv: 2603.27065 [cs.CL]. URL: https://arxiv.org/abs/2603.27065.

[9] Giulio Starace et al. *PaperBench: Evaluating AI's Ability to Replicate AI Research*. 2025. DOI: 10.48550/arXiv.2504.01848. arXiv: 2504.01848 [cs.AI]. URL: https://arxiv.org/abs/2504.01848.

[10] Yujia Li et al. *Competition-Level Code Generation with AlphaCode*. 2022. DOI: 10.1126/science.abq1158. arXiv: 2203.07814 [cs.PL]. URL: https://arxiv.org/abs/2203.07814.

[11] Aman Madaan et al. *Self-Refine: Iterative Refinement with Self-Feedback*. 2023. DOI: 10.48550/arXiv.2303.17651. arXiv: 2303.17651 [cs.CL]. URL: https://arxiv.org/abs/2303.17651.

[12] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. *Reflexion: Language Agents with Verbal Reinforcement Learning*. 2023. arXiv: 2303.11366 [cs.AI]. URL: https://arxiv.org/abs/2303.11366.

[13] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2022. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.