Repo2ROCm: An Agentic System for Reproducing Research Code on AMD ROCm GPUs

Gaurav Sarkar[1,†] **Umang Pandey**[2] **Pratik Prabhanjan Brahma**[2]
Zicheng Liu[2] **Emad Barsoum**[2]
[1]IIIT Hyderabad [2]Advanced Micro Devices, Inc. (AMD)
gaurav.sarkar@research.iiit.ac.in
{upandey, pprabhan, Zicheng.Liu, ebarsoum}@amd.com

Abstract

Machine learning practitioners routinely rely on public code repositories accompanying research papers, libraries, and open-source projects to evaluate, baseline, or build upon prior work. Yet getting such repositories to run and reproduce results often involves non-trivial effort, and is especially challenging when the target software and hardware stack differs from the one the original code was developed on. A prominent instance is running repositories originally built for CUDA-enabled hardware on AMD GPUs supporting the ROCm software stack. Here, successful installation alone is not a sufficient signal: a range of subtler issues may cause the code to either fail outright or silently fall back to inefficient execution.

We present **Repo2ROCm**, an agentic system that migrates CUDA-centric ML repositories to ROCm in an automated manner. Repo2ROCm reconstructs a ROCm-compatible environment, adapts hardware-specific assumptions, executes a real workload, and confirms GPU-backed execution emitting auditable artifacts at every step (environment specs, source patches, execution logs, and verification evidence). We evaluate on **ResearchBench**, 60 repositories belonging to recent papers from top AI conferences. While only 18/60 run out of the box, Repo2ROCm reaches verified functional reproduction for 59/60 (3.3× improvement). Against popular general-purpose coding agents on a curated ten-repository diagnostic set (with all agents using the same Claude Sonnet 4 backbone), Repo2ROCm achieves the highest **Portability Fidelity Score (PFS)**, a measure of how faithfully a run preserves the accelerator-critical execution path, real workload, and verified environment, scoring 77.3 vs. 61.4 (Claude Code), 58.7 (Mini-SWE-Agent), and 40.3 (Cursor Agent). Code and artifacts are available at github.com/Gaurav7888/Repo2ROCm.

† Work done when the author was affiliated with AMD.

1 Introduction

For industry AI teams, public ML repositories are often the first artifact used to decide whether a new method is worth adopting, but running those off-the-shelf repositories reliably is rarely just a clone-and-execute exercise. In practice, it often takes quite a bit of manual time and effort to integrate and reproduce. Failures arise at two levels: first, reconstructing the repository's intended software and experimental setup; second, moving that setup onto a different accelerator platform.

Tier 1: Reproducibility challenge. Evaluating a public ML repository begins with a practical question: can the code run reliably in our environment? This is often non-trivial because research repositories frequently have stale setup instructions, conflicting dependency files, version mismatches, implicit entry points, missing external artifacts, and experiment scripts tied to the authors' local environment.

Tier 2: Cross-hardware platform migration. The challenge deepens when the repository assumes one accelerator stack but the target hardware uses another. For CUDA-centric repositories evaluated on AMD ROCm, migration requires selecting a compatible software stack, replacing or rebuilding CUDA-specific dependencies, and adapting source-level hardware assumptions such as device checks, custom kernels, and backend-specific libraries. These changes must preserve the repository's *execution path*: the model, workload, accelerator backend, GPU-critical libraries, kernels, and device placement used at runtime. A repository that runs only after disabling accelerator-critical components, substituting a degraded or inefficient backend, falling back to CPU execution, or reducing the workload may no longer evaluate the method described by the paper, making research-result reproduction unreliable.

EMNLP 2026 Industry Track, pages 2265–2278
October 24-29, 2026 ©2026 Association for Computational Linguistics

Verification is therefore central. Successful installation, imports, command-line help screens, device-availability probes, or toy smoke tests do not establish that the intended workload ran on AMD GPU hardware. We therefore focus on *functional reproduction*: reconstructing a ROCm-compatible environment, executing the repository's intended workload, and verifying that the accelerator-critical execution path is preserved. This target is narrower than full paper-result reproduction, but stronger than installation success.

Why existing agents are insufficient. General-purpose coding agents are effective at making software appear runnable, but ROCm migration requires a stronger standard than task completion. In our setting, an agent must preserve the repository's accelerator-critical execution path and prove that a real workload ran on AMD GPU hardware. Existing agents may make progress through path-changing repairs, such as replacing FlashAttention with SDPA, disabling GPU-critical modules, or reducing workload scale. These repairs can produce a runnable repository, but they do not establish faithful cross-hardware reproduction. The missing ingredient is not just better code editing, but evidence-gated migration: every environment change, dependency substitution, source edit, and success claim must be tied to runtime evidence that the intended workload still executes on the target accelerator.

These challenges motivate an automated system that treats repository migration as an evidence-driven reproducibility task rather than a software-debugging task. Such a system must repair environments and adapt code for ROCm while verifying that the intended workload still follows the accelerator path being evaluated.

Contributions. This paper makes the following contributions:

- **REPO2ROCM:** an automated system for functional reproduction of CUDA-centric research repositories on AMD ROCm with zero manual intervention.
- **Evidence-gated plan-then-execute migration:** a guarded architecture that first plans repository migration statically, then executes changes only when deterministic runtime evidence supports environment changes, source edits, or GPU-backed success claims.
- **RESEARCHBENCH and PFS:** a 60-repository real-world research-code benchmark and Portability Fidelity Score for evaluating faithful cross-hardware execution beyond binary run/fail outcomes.

We do not claim reasoning-and-acting execution, knowledge retrieval, checkpoint-rollback, causal memory, or deterministic verification as individually novel mechanisms. Our contribution is their integration into a *guarded cross-accelerator migration architecture*, in which every irreversible action (switching base images, replacing accelerator libraries, editing critical source paths) and every success declaration must be backed by non-overridable runtime evidence, together with the RESEARCHBENCH benchmark and the fidelity-oriented PFS evaluation methodology.

2 The REPO2ROCM System

REPO2ROCM takes a GitHub repository as input and produces a ROCm-ready execution package: a compatible environment, source patches when needed, execution logs, and evidence that the target workload ran on AMD GPU hardware. Figure 1 gives an end-to-end view of the system, from repository fingerprinting and static planning to guarded execution, verification, and artifact generation.

Inputs, outputs, and operating modes. We use three operating settings throughout the paper:

- **Mode 0:** the out-of-the-box baseline. The repository is cloned and run by following its public instructions, with no intervention from REPO2ROCM.
- **Mode 1:** the main REPO2ROCM setting. REPO2ROCM reconstructs the environment, adapts CUDA-centric assumptions, runs the intended workload, and declares functional reproduction only after verified AMD GPU execution.
- **Mode 2:** a limited metric-checking extension used when a paper claim can be expressed as a command, metric, and tolerance.

Keeping Mode 2 separate is deliberate: functional reproduction and full paper reproduction are different goals, and our main claim is cross-hardware functional reproduction rather than arbitrary paper-result reproduction. A successful REPO2ROCM run emits a reproducible Dockerfile, source patch,

command logs, verification evidence, and the full agent trajectory.

When a repository counts as reproduced. We fix an explicit, repository-level definition of functional reproduction that the deterministic verifier enforces and the agent cannot override. A Mode 1 run is counted as a success only if *all* of the following hold: (i) an official workload of the repository (inference, training, evaluation, or a forward pass) executes to completion; (ii) there is runtime evidence that this workload ran on AMD GPU hardware through the ROCm stack; (iii) the repository's accelerator-critical components (e.g., attention backends, custom kernels, GPU-critical dependencies) are preserved rather than disabled, stubbed, or replaced by a degraded path; and (iv) there is no unreported CPU fallback, backend substitution, disabled module, or toy-scale execution. This criterion is deliberately stronger than successful installation, imports, device-availability probes, or smoke tests, and narrower than full paper-result reproduction (Mode 2).

Repository fingerprinting and static planning. The first stage runs outside Docker and performs lightweight repository reconnaissance. REPO2ROCM clones the repository, identifies its framework stack, configuration files, candidate entry points, CUDA-specific dependencies, and custom GPU kernels, then ranks compatible AMD ROCm base images. The planner emits a compact execution plan containing the likely install path, migration hazards, workload command, and validation checks, avoiding expensive Docker-agent turns on basic discovery. For limited Mode 2 runs, it also records the selected command, target metric, and tolerance when these can be extracted unambiguously from the paper and code.

ROCm base-image ranking. Base-image selection is a high-impact planning decision: an incompatible container can fail before repository-specific repair begins. REPO2ROCM first detects the host accelerator architecture and gathers candidate ROCm images from live Docker metadata and locally cached images. Candidates that target the wrong architecture, predate the detected GPU family, are explicitly excluded after a failed attempt, or are marked as repeatedly failing in the knowledge base are filtered out before scoring.

The remaining images are ranked by matching repository requirements against image inventory tokens. Let T_r be the set of normalized package and framework name tokens extracted from the repository's imports and dependency files (e.g., `requirements.txt`, `environment.yml`, `setup.py`/`pyproject.toml`), and let T_i be the inferred package inventory of candidate image i. Tokens are lowercased and canonicalized to a common form (e.g., distribution and import aliases are unified) before matching. Version constraints are excluded from both T_r and T_i and handled separately by compatibility filters, so this term measures package presence rather than exact version agreement. The main compatibility term is a Jaccard score:

$$J(r, i) = \frac{|T_r \cap T_i|}{|T_r \cup T_i|}.$$

The final score adds bonuses for matching the detected workload framework, being locally cached, explicitly matching the host architecture token, and appearing in recent Docker metadata, with soft penalties for Python-version mismatch, overly large images, or floating tags. The highest-scoring surviving image becomes the Dockerfile base image. If the container crashes during early startup, REPO2ROCM records the failure, excludes that image, and reranks the remaining candidates.

Kernel migration. Repositories with custom CUDA kernels require a separate migration path from ordinary Python dependency repair. When the planner detects `.cu` files or CUDA extension builds, REPO2ROCM invokes a kernel agent that applies CUDA-to-HIP conversion, using tools such as `hipify` where appropriate, and then validates whether the converted extension can build and load under ROCm.

Guarded execution in Docker. The execution stage runs inside a GPU-pinned Docker sandbox. A ReAct-style executor follows the static plan one action at a time: installing dependencies, modifying files, building extensions, running validation commands, and recording results. After successful milestones, the sandbox state is checkpointed; failed actions can roll back to the most recent stable state instead of restarting the repository from scratch. This rollback mechanism is important because ROCm migration is exploratory: several package versions, containers, or source patches may fail before a valid configuration is found. Recovering from a checkpoint lets the agent explore alternatives without corrupting the last known-good environment.

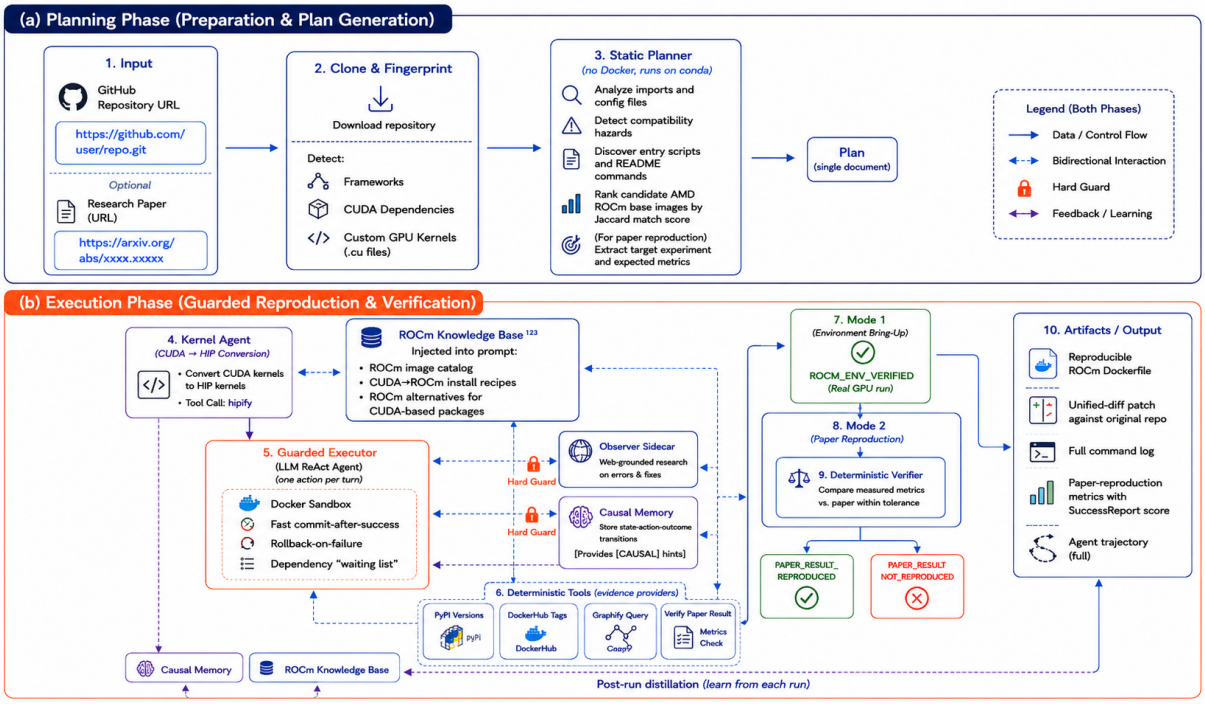

Figure 1: The REPO2ROCM pipeline, shown in two phases. **(a) Planning phase:** the system clones and fingerprints the repository (frameworks, CUDA dependencies, custom `.cu` kernels), then a static planner running outside Docker detects compatibility hazards, discovers entry points, ranks candidate AMD ROCm base images by Jaccard match score, and emits a single plan document. **(b) Execution phase:** a guarded ReAct executor runs the plan one action at a time inside a GPU-pinned Docker sandbox, optionally invoking a kernel agent for CUDA-to-HIP conversion, and is supported by a ROCm knowledge base, deterministic evidence-provider tools, an observer sidecar, and causal memory. Hard guards block irreversible actions that lack supporting runtime evidence. Mode 1 terminates with a verified GPU-backed run (`ROCM_ENV_VERIFIED`); the preliminary Mode 2 additionally compares extracted metrics against a specified target through a deterministic verifier. Post-run distillation feeds causal memory and the ROCm knowledge base for later runs.

Evidence providers and hard guards. REPO2ROCM treats deterministic tools as evidence providers rather than optional helpers. Package and image lookup tools check PyPI and Docker metadata, code-indexing tools such as Graphify let the executor inspect relevant files and symbols without loading the entire repository into the language-model context, and GPU validation tools verify that the workload actually executes on AMD hardware. Hard guards block irreversible actions until supporting evidence is available: for example, the executor cannot switch base images, replace accelerator libraries, modify critical source paths, or declare `ROCM_ENV_VERIFIED` from weak evidence such as an import check, help screen, or CPU-only run. In limited Mode 2 runs, a deterministic verifier compares extracted metrics against the specified target and records a match or mismatch.

2.1 Deterministic Verification

A language model can generate plausible explanations for both successful and unsuccessful runs, making free-form success claims difficult to trust. REPO2ROCM therefore delegates functional-reproduction checks, and optional metric checks in Mode 2, to deterministic verifiers that evaluate execution logs and predefined criteria. The verifier's output is authoritative and cannot be overridden by the agent, keeping outcomes evidence-based, auditable, and resistant to agent gaming.

Examples. First, GPU-backed workload verification checks that the selected container exposes AMD devices through the ROCm-backed runtime and that a repository-level forward, training, inference, or evaluation command runs before emitting `ROCM_ENV_VERIFIED`; an import check alone is insufficient. Second, accelerator-path verification checks that GPU-critical libraries remain active: for example, a FlashAttention migration must

show a ROCm-compatible import and GPU execution path, while an SDPA replacement is recorded as a degraded fallback rather than silent success.

2.2 ROCm Knowledge, Observer, and Causal Memory

ROCm knowledge base. The ROCm knowledge base injects curated migration rules into the executor, including known ROCm package variants, compatible container choices, CUDA-to-ROCm replacement patterns, and common fixes for recurring build failures. This prevents the agent from repeatedly rediscovering the same ROCm-specific solutions through unrestricted search.

Observer sidecar. The observer sidecar is triggered when the main ReAct loop stops making progress for several steps or when its hints become stale. It reviews the recent trajectory and command logs from an outside perspective, can consult external documentation for fresh context, and injects concise repair suggestions that help the executor escape repeated failure loops.

Causal memory. Causal memory records prior ⟨state, action, outcome⟩ transitions and supplies hints when a new repository reaches a similar failure state. It complements the static ROCm knowledge base by capturing run-time repair experience from previous repositories while keeping final success dependent on deterministic runtime evidence.

Post-run knowledge distillation. After each run, REPO2ROCM distills the trajectory into reusable migration knowledge, creating a learning loop across repositories. The distillation step summarizes which actions succeeded, which failed, which ROCm image and package choices were stable, and which errors required rollback or observer intervention. These summaries update the causal memory and ROCm knowledge base used by later runs. The learning loop is intentionally conservative: distilled knowledge can provide hints or rank future choices, but it does not bypass hard guards or deterministic verification.

Artifacts and operational safety. Each run is isolated in its own Docker sandbox with command timeouts, resource limits, and checkpoint-and-rollback recovery. The resulting artifact bundle contains the generated Dockerfile, source-code patch, command logs, verification outputs, and agent trajectory; optional Mode 2 runs also include the metric-comparison report. These artifacts make the migration auditable and allow many repositories to be run independently across a multi-GPU server.

3 Evaluation

Benchmark. We evaluate REPO2ROCM on RESEARCHBENCH, a benchmark of 60 repositories accompanying recent machine-learning and natural-language-processing papers from ICLR, ICML, ACL, and NeurIPS. We build this benchmark because existing coding-agent benchmarks are not designed for research-code setting: they often use mature software repositories with curated issue descriptions, test suites, well documented Readme and maintainable project structure. By contrast, research repositories commonly have incomplete setup instructions, unclear entry points, external datasets or checkpoints, and hardware-specific assumptions. RESEARCHBENCH captures this less curated but industrially common workflow across diverse repository types, including inference-heavy systems, training pipelines, agentic workflows, dataset-processing code, and benchmarking suites, all evaluated for faithful execution on available accelerator hardware.

Metrics. Our primary metric compares Mode 0 out-of-the-box execution against Mode 1 functional reproduction: verified GPU-backed execution on AMD hardware.

For comparisons with general-purpose agents, a binary ran/failed label is too coarse: an agent may make a repository execute by disabling accelerator-critical components, substituting a fallback backend, using a toy workload, or relying on a prebaked environment. We therefore define the *Portability Fidelity Score* (PFS), a continuous 0–100 score for each repository-agent pair:

$$\mathrm{PFS} = \mathrm{Ran} \times (15 + 35F_{\mathrm{path}} + 20A_{\mathrm{work}} + 20A_{\mathrm{env}} + 10D_{\mathrm{proc}}).$$

Here, Ran gives baseline credit for producing runnable evidence, F_{path} measures preservation of accelerator-critical components, A_{work} measures whether the real workload was run, A_{env} measures whether a ROCm environment was built and verified, and D_{proc} measures evidence-gated process discipline. Appendix B gives the full grading rubric. All Mode 1 runs are evaluated using a deterministic functional-reproduction rubric and degradation indicators such as attention fallbacks, environment substitutions, or reduced execution scale.

Experimental setup. All experiments run on a host with eight AMD Instinct MI350X devices (gfx950, 270.6 GB HBM3E memory per device), running ROCm Runtime 1.18 (amdgpu driver 6.14.14). The environment configuration uses one or up to eight devices depending on the use case. The base image is selected per repository by the planner. REPO2ROCM uses Claude Sonnet 4 as its language-model backbone and is capped at 100 turns per run; we impose no fixed token budget.

Agent comparison and fairness. For the comparison in Table 2, all four systems (REPO2ROCM, Claude Code, Cursor Agent, and Mini-SWE-Agent) use the same underlying model, Claude Sonnet 4, so that measured differences reflect system design rather than model capability. Every agent receives the same repository revision, target hardware, task specification, execution budget, fidelity requirements, and access to public ROCm documentation. REPO2ROCM additionally ships a curated ROCm knowledge base (package mappings, compatible container configurations, CUDA-to-ROCm migration patterns, and verified build fixes); this knowledge base is an explicit component of the proposed architecture rather than an external advantage, so the comparison contrasts complete systems.

Cost and latency. A typical REPO2ROCM migration consumes on average 719K tokens (median 608K) and about 18–20 minutes of wall-clock time, using between one and eight MI350X devices depending on the repository. We report token usage and wall-clock time rather than a single GPU-hour figure because the number of devices engaged varies per repository.

3.1 ResearchBench Results

Table 1 summarizes the main RESEARCHBENCH outcome. Out-of-the-box execution succeeds for only 18/60 repositories. The remaining repositories fail due to a mixture of ROCm-specific incompatibilities and common research-code breakages: CUDA-only dependencies, unsupported container environments, hard-coded CUDA assumptions, silent CPU fallback, stale APIs, syntax errors, and small repository bugs that require source-level fixes.

REPO2ROCM increases the functional reproduction rate to 59/60, a 3.3× improvement over the out-of-the-box baseline of 18/60.[1]

Setting	Verified runs	Run rate
Mode 0 (out of the box)	18/60	30.0%
Mode 1 (REPO2ROCM)	59/60	98.3%

Table 1: Summary of RESEARCHBENCH functional reproduction results. A verified run requires GPU-backed execution on AMD ROCm.

Detailed repository-wise outcomes are reported in Appendix D, Table 5.

A qualitative inspection shows that failures often came from path-changing repairs rather than simple crashes: unsupported containers, disabled accelerator-specific components, incorrect API adaptations, and fallback implementations such as SDPA instead of Flash-Attention. These cases motivate evaluating functional reproduction with execution-path fidelity rather than binary run/fail labels.

These failure modes directly motivated the evidence-gated execution model and ROCm-specific knowledge base used by REPO2ROCM.

When repositories exceed available GPU memory, REPO2ROCM applies controlled scaling adjustments, such as reducing batch size or generation length, while preserving the original model and execution path whenever possible.

3.2 Comparison with General-Purpose Agents

We compare REPO2ROCM against two closed-source commercial agents (Cursor Agent and Claude Code Agent) and one open-source agent (Mini-SWE-Agent). The comparison uses a manually curated set of ten repositories, selected *before* observing any agent scores to span diverse migration challenges: image generation, multimodal classification, vision analysis, JAX training, LLM training and editing, recommendation, KV-cache optimization, and protein modeling. We therefore treat it as a diversity-oriented diagnostic comparison rather than a benchmark-wide estimate; complete per-repository results and component-level annotations are released with the artifact bundle (Appendix A). All agents use the same Claude Sonnet 4 backbone under the matched conditions of §3.

Table 2 reports average PFS. REPO2ROCM achieves the highest average score, indicating that

[1] The only failure, DiffusionVeteran, fails under both Mode 0 and Mode 1 (Table 5), not a regression from a working baseline.

Agent	Average PFS
REPO2ROCM	**77.3**
Claude Code Agent	61.4
Mini-SWE-Agent	58.7
Cursor Agent	40.3

Table 2: REPO2ROCM versus general-purpose coding agents on the ten-repository diagnostic set, under the Portability Fidelity Score (PFS). All agents use the same Claude Sonnet 4 backbone.

Configuration	Ran	Run rate	Speedup
baseline	15/20	75%	1.00×
causal-mem	**18/20**	**90%**	1.16×

Table 3: Causal-memory ablation across 20 CUDA-oriented, GPU-dependent repositories. Paired comparison; the memory-disabled baseline retains planning, the ROCm knowledge base, guarded execution, rollback, and deterministic verification.

its gains are not only from making repositories run, but from preserving the intended accelerator path, workload, and ROCm environment. Appendix C analyzes the path-changing repairs made by the general-purpose agents, and Appendix B reports a weight-sensitivity analysis showing that the ordering is preserved under equal component weights and under removal of any single PFS component.

Run-to-run stability. To probe variance, we repeated migrations five times each for three repositories spanning easy, moderate, and difficult cases (AlphaRec, LADDER, and UltraHR-100K). Each repeat started from the same commit in a clean sandbox with identical hardware, model, budget, knowledge-base snapshot, and verifier configuration, with cross-run memory disabled. All $15/15$ runs reproduced the same verification outcome as the primary evaluation. Pooled across the three repositories, PFS was 67.13 ± 21.47 and token usage was $1{,}027{,}801 \pm 945{,}936$; because these statistics are pooled across repositories of differing difficulty, the standard deviation primarily reflects between-repository variation rather than run-to-run noise.

3.3 Mode 2 : Case Study

We use Mode 2 only as a case study after Mode 1 has established a verified ROCm execution environment. **TURBOQUANT** (Zandieh et al., 2026) is the positive example: the repository exposes a clear quantization experiment with machine-checkable metrics, and the verifier found the reported compression, similarity, and match-rate values within tolerance. As a contrasting case, **DKV-CACHE** (Ma et al., 2025) executed its intended workload on AMD GPUs under Mode 1, but its reported hardware-sensitive speedup did not reproduce within tolerance, so Mode 2 correctly classified it as *not reproduced*. Of the two preliminary Mode 2 cases, one matched the reported metrics and one did not, illustrating that result-level verification can distinguish functional execution from scientific-result reproduction without conflating the two.

Negative verification outcomes frequently reflected well-known reproducibility challenges rather than failures of ROCm execution. Although repositories could often be built and executed successfully, independent result validation was hindered by missing datasets, unavailable checkpoints, proprietary or undocumented preprocessing steps, gated model access, external service dependencies, and incomplete evaluation instructions. Such issues are common in contemporary machine learning research and illustrate that releasing source code alone is often insufficient for enabling full reproduction of reported results.

3.4 Impact of Causal Memory

We ablate causal memory on 20 CUDA-oriented, GPU-dependent repositories, selected before running the ablation to span custom-kernel workloads and the migration-failure types observed in §3 rather than for the mere presence of `.cu` files. The comparison is paired: the full system is compared against an otherwise identical configuration with causal memory disabled, using the same repositories, budgets, and evaluation conditions. Crucially, this memory-disabled baseline is *not* the out-of-the-box Mode 0 setting: it retains static planning, the ROCm knowledge base, guarded execution, rollback, and deterministic verification, so the ablation isolates the contribution of causal memory alone. As shown in Table 3, causal memory improves the functional-run rate from 75% $(15/20)$ to 90% $(18/20)$, a $1.2\times$ increase. On repositories completed by both configurations, it yields a $1.16\times$ relative speedup; counting rescued failures and timeouts as baseline wasted effort increases the effective speedup to roughly $1.42\times$.

4 Related Work

Agentic software engineering. Recent systems and benchmarks such as SWE-BENCH (Jimenez et al., 2024) and SWE-AGENT (Yang et al., 2024) study whether language-model agents can modify repositories and resolve software issues through execution feedback. These works demonstrate the value of iterative code editing and validation, building on reasoning-and-acting patterns such as ReAct (Yao et al., 2023) and memory-based agent improvement such as Reflexion (Shinn et al., 2023). REPO2ROCM shares the use of iterative execution, but targets a different failure point: before code-level tasks can be evaluated, many research repositories must first be made runnable on the target accelerator. Our setting also lacks the curated tests and issue descriptions common in software-engineering benchmarks, so success must be grounded in workload and GPU-execution evidence.

Environment reconstruction. Environment reconstruction systems such as REPO2RUN (Hu et al., 2025) automate dependency resolution and Docker environment construction for code repositories. REPO2ROCM builds on this line of work but specializes it to heterogeneous GPU migration. In our setting, choosing a container and installing packages is not enough: the system must select ROCm-compatible images, adapt CUDA-centric dependencies and source assumptions, and verify that the intended workload executes on AMD GPU hardware without silent CPU fallback or path-changing substitutions.

GPU backend portability. The practical difficulty of cross-hardware reproduction is amplified by accelerator-specific software stacks. AMD ROCm (AMD, 2025) provides a CUDA-adjacent ecosystem, but high-performance ML repositories often depend on backend-specific components such as FlashAttention (Dao et al., 2022; Dao, 2023) or serving stacks such as vLLM (Kwon et al., 2023). These components may require different builds, kernels, or fallback behavior across hardware platforms. REPO2ROCM treats such backend-specific knowledge as part of the migration process, combining ROCm-specific guidance with evidence-gated verification of the executed path.

Limitations

Several limitations remain. First, this work focuses on functional reproduction: constructing a ROCm-compatible environment and verifying that the intended workload runs on AMD GPU hardware. We tested Mode 2 only as a limited metric-checking extension on simple cases where the target command, metric, and tolerance were explicit. Making REPO2ROCM robust for broader result reproduction is the next phase of the project, requiring better experiment extraction, metric parsing, multi-metric verification, and handling of long-running or data-dependent workflows.

Second, the evaluation is limited to RESEARCH-BENCH, AMD ROCm, and the hardware/software stack described in §3. The benchmark is constructed from recent research papers, so it may not capture the additional dependency drift, broken links, and archival issues that appear in repositories from papers published four or five years earlier. Extending the benchmark to older research repositories is part of our future evaluation pipeline. Although the failure modes are common in cross-hardware ML migration, results may differ for other ROCm versions, accelerator generations, target backends, or internal industry repositories.

Third, REPO2ROCM cannot resolve all external research-artifact dependencies. Some repositories require datasets or checkpoints behind request forms, author email, Google Drive permissions, expired links, or downloads that exceed practical time limits. In such cases, REPO2ROCM can document the blocker and preserve the migration artifacts, but functional or metric-level verification may remain impossible without access to the missing artifacts.

Further, REPO2ROCM relies on a curated ROCm compatibility knowledge base that must be maintained as container images, package variants, and framework support evolve. Although live package and image lookups reduce staleness, high-quality migration rules require periodic updates.

Finally, our evaluation methodology has two scope limits. PFS path-fidelity grades are produced by a single language-model judge (Claude Opus 4.6) under a frozen rubric applied identically to all systems (Appendix B); while this keeps grading consistent and agent-agnostic, we do not yet report inter-annotator agreement, and an independent human re-grading with judge–human agreement (weighted Cohen's κ) is left to future work. In addition, the agent comparison uses a manually

curated ten-repository diagnostic set rather than the full benchmark, so the reported PFS gaps are diagnostic rather than benchmark-wide estimates.

Ethical Considerations

REPO2ROCM executes third-party research repositories inside containerized GPU environments and therefore inherits the risks associated with running untrusted code. Deployments should incorporate appropriate isolation mechanisms, resource controls, and security policies to mitigate potential misuse or unintended side effects. Furthermore, patches generated by the system should be reviewed by human operators before being incorporated into production codebases.

The system is designed to support transparent and verifiable reproducibility studies. Reproduction outcomes are determined by deterministic evaluation procedures and evidence collected during execution rather than by model-generated claims. This design reduces the risk of fabricated or unverifiable reproduction results while preserving an auditable record of system behavior.

All experiments reported in this work operate on publicly available repositories and accompanying research artifacts. The system does not require the collection of personal information and is not intended for processing sensitive user data.

References

AMD. 2025. AMD ROCm software platform documentation. https://rocm.docs.amd.com. Accessed 2026.

Tri Dao. 2023. FlashAttention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Ruida Hu, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. 2025. Repo2Run: Automated building executable environment for code repository at scale. *arXiv preprint arXiv:2502.13681*.

Houcheng Jiang, Junfeng Fang, Ningyu Zhang, Mingyang Wan, Guojun Ma, Xiang Wang, Xiangnan He, and Tat-Seng Chua. 2025. AnyEdit: Edit any knowledge encoded in language models. In *International Conference on Machine Learning (ICML)*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*.

Jiyeon Kim, Hyunji Lee, Hyowon Cho, Joel Jang, Hyeonbin Hwang, Seungpil Won, Youbin Ahn, Dohaeng Lee, and Minjoon Seo. 2025. Knowledge entropy decay during language model pretraining hinders new knowledge acquisition. In *International Conference on Learning Representations (ICLR)*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*.

Chenao Li, Shuo Yan, and Enyan Dai. 2025. UniZyme: A unified protein cleavage site predictor enhanced with enzyme active-site knowledge. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Haofei Lu, Dongqi Han, Yifei Shen, and Dongsheng Li. 2025. What makes a good diffusion planner for decision making? In *International Conference on Learning Representations (ICLR)*.

Xinyin Ma, Runpeng Yu, Gongfan Fang, and Xinchao Wang. 2025. dKV-Cache: The cache for diffusion language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Leheng Sheng, An Zhang, Yi Zhang, Yuxin Chen, Xiang Wang, and Tat-Seng Chua. 2025. Language representations can be what recommenders need: Findings and potentials. In *International Conference on Learning Representations (ICLR)*.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Ao Wang, Hui Chen, Jiaxin Li, Jianchao Tan, Kefeng Zhang, Xunliang Cai, Zijia Lin, Jungong Han, and Guiguang Ding. 2025. PrefixKV: Adaptive prefix KV cache is what vision instruction-following models need for efficient generation. In *Advances in Neural Information Processing Systems (NeurIPS)*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. 2026. TurboQuant: Online vector quantization with near-optimal distortion rate. In *International Conference on Learning Representations (ICLR).*

A Artifact Layout

Each REPO2ROCM run writes an artifact bundle under `verification/<repo>/`. The bundle is intended to make the migration auditable and reproducible:

- `Dockerfile`: the ROCm-ready environment selected and constructed by the system.
- `patch/`: unified diffs containing source changes made to adapt the repository.
- `inner_commands.json` and `outer_commands.json`: command logs from inside and outside the Docker sandbox.
- `plan.txt`: the static planner output, including detected hazards, selected base image, candidate workload, and validation steps.
- `trajectory.jsonl`: the step-by-step agent trajectory used for audit and post-run knowledge distillation.
- `metric_check.json`: the optional Mode 2 metric-comparison report, present only for limited result-checking runs.

B Portability Fidelity Score Details

PFS is computed per repository-agent pair:

$$\mathrm{PFS} = \mathrm{Ran} \times (15 + 35F_{\mathrm{path}} + 20A_{\mathrm{work}} + 20A_{\mathrm{env}} + 10D_{\mathrm{proc}}).$$

$\mathrm{Ran} \in \{0, 1\}$ indicates whether the run completed and produced runnable evidence. If the run does not execute, PFS is zero. If it runs but satisfies none of the fidelity criteria, it receives the 15-point running baseline. All other subcomponents are normalized to $[0, 1]$, so the maximum score is 100.

Path fidelity. For path fidelity, each repository has K accelerator-critical components detected from the repository itself, such as attention backends, GPU-critical dependencies, or custom kernels. The detector is repository-specific rather than a fixed library list. Each component receives grade g_i: 1.0 for a preserved ROCm-native or equivalent path, 0.5 for a degraded but functional path, and 0.0 for a disabled, stubbed, removed, or signature-altered component. Examples of preserved paths include AMD Triton FlashAttention, ROCm builds of GPU-critical dependencies, or HIP-ported kernels. Examples of degraded paths include SDPA replacing FlashAttention or CPU fallback for a GPU dependency. Examples of broken-but-masked paths include commented-out modules, disabled kernels, or calls rewritten with an incorrect signature. We compute

$$F_{\mathrm{path}} = \frac{1}{K} \sum_{i=1}^{K} g_i.$$

This keeps path fidelity bounded and independent of the number of checked components. Adding support for a new dependency such as `gsplat`, `bitsandbytes`, or a custom kernel only requires adding a detector that maps the component to one of the three grades.

Workload authenticity. Workload authenticity measures whether the run executed the paper's real workload. It penalizes model substitution, dataset scale-down, toy dimensions, smoke tests, import checks, `-help`, or device-availability checks without a real forward, train, or evaluation step on GPU.

Environment authenticity. Environment authenticity measures whether the agent selected, built, and verified a ROCm environment for the repository. Runs that inherit an unverified pre-baked container score lower than runs that select a base image, build the environment, and show genuine AMD GPU execution evidence.

Process discipline. Process discipline rewards evidence lookup before irreversible changes and efficient evidence-gated execution. Agents receive higher credit when they consult package, image, code-indexing, or GPU-verification tools before modifying dependencies, patching accelerator paths, or declaring success.

Grading protocol. Path-fidelity grades g_i are assigned by a single language-model judge (Claude Opus 4.6) applied to the artifact bundle (Dockerfile, patches, command logs, GPU-execution evidence, and the repository-specific accelerator-critical components), using a frozen rubric, prompt, and temperature, and the same configuration for every system so that grading is consistent and agent-agnostic. Because a single automated judge was used, we do not report inter-annotator agreement here; an

Repository	P	W	E	D
REPO2ROCM				
TarFlow	31.5	7	18	7
LADDER	31.0	2	20	6
MM-ArgFallacy	26.0	2	20	6
SuperCollapse	18.0	5	8	9
UltraHR-100K	0.0	2	18	4
Claude Code				
TarFlow	26.25	3	15	7
LADDER	29.16	1	15	5
MM-ArgFallacy	23.35	0	14	5
SuperCollapse	17.50	0	12	5
UltraHR-100K	11.66	1	14	6

Table 4: Per-component PFS point contributions on the five component-annotated comparison repositories: P (path fidelity, out of 35), W (workload authenticity, out of 20), E (environment authenticity, out of 20), and D (process discipline, out of 10). For a run that executed, $\mathrm{PFS} = 15+\mathrm{P}+\mathrm{W}+\mathrm{E}+\mathrm{D}$. Mean PFS: REPO2ROCM 63.1 vs. Claude Code 57.2.

independent human re-grading with judge–human agreement (weighted Cohen's κ) is left to future work (see Limitations).

Weight sensitivity. The PFS weights encode failure severity: path fidelity carries the largest weight because a silent backend substitution can invalidate an otherwise successful run; workload and environment authenticity are direct task requirements; and process discipline carries the smallest weight as secondary auditability evidence. The 15-point floor credits basic runnable evidence. To test whether the conclusions depend on these specific weights, we recomputed scores on the five comparison repositories for which we have component-level annotations, using both the original weights and equal component weights. Table 4 lists the per-component point contributions. Under the original weights REPO2ROCM averages 63.1 versus 57.2 for Claude Code, and under equal component weights 63.2 versus 55.9; removing any single component does not change the aggregate ordering. On UltraHR-100K, Claude Code scores higher than REPO2ROCM (47.7 versus 39.0), which we report transparently.

C Qualitative Analysis of General-Agent Failures

Several general-purpose agent runs reached an apparently runnable state by changing the repository's execution path. The most common pattern was backend substitution: when a ROCm-compatible FlashAttention path was not installed, agents replaced FlashAttention with PyTorch SDPA in repositories such as KNOWLEDGE-ENTROPY and PREFIXKV. This allowed execution to proceed, but it changed the accelerator-critical computation being evaluated and often reduced efficient use of the target GPU.

We also observed signature and module-level changes that masked failures rather than preserving the intended behavior. In PREFIXKV, Mini-SWE-Agent called a library function with an incorrect signature, producing a runnable path that no longer matched the original implementation. In UNIZYME, a module used by the original repository was left disabled so the run could complete. These cases illustrate why a binary run/fail metric is insufficient: the repository may run, but the measured path differs from the one the paper intended.

REPO2ROCM treats these cases differently. Its guards require evidence before accepting backend substitutions, source edits, or success claims, and PFS penalizes degraded, disabled, or signature-altered accelerator-critical components.

D Benchmark Repository Details

Table 1 summarizes the benchmark outcomes in the main paper. Table 5 gives the repository-wise Mode 0 and Mode 1 outcomes.

Repository	Mode 0	Mode 1	Repository	Mode 0	Mode 1	Repository	Mode 0	Mode 1
GCTM	✗	✓	LayerDAG	✗	✓	HARDMath	✓	✓
mm-argfallacy	✗	✓	EARTH	✗	✓	ELABORATION	✓	✓
understanding_mcqa	✓	✓	Knowledge-Entropy	✗	✓	VTI	✗	✓
WildBench	✗	✓	convolutional_diffusion	✗	✓	modelmap	✓	✓
DiffusionVeteran	✗	✗	AlphaRec	✗	✓	TimeEmb	✗	✓
Self-Certainty	✗	✓	pcx	✓	✓	CEB	✗	✓
Ladder	✗	✓	paperbanana	✓	✓	ACL25-CoPE	✗	✓
TOP_ERL_ICLR25	✓	✓	SeqSNN	✓	✓	PrefixKV	✗	✓
DataEnvGym	✗	✓	MCNC	✗	✓	CLEME	✓	✓
m-rewardbench	✗	✓	Risk-Sensitive-CMDP	✓	✓	FR-Spec	✗	✓
askqe	✓	✓	HarmoniCa	✗	✓	moa	✗	✓
HealthGPT	✗	✓	topoloss	✓	✓	SemEval2025-EAMT	✗	✓
CATCH	✗	✓	llm_reproducibility	✗	✓	ORMind	✗	✓
CSL-Mem	✗	✓	torch_brain	✗	✓	AIRMVC	✗	✓
TemporalHead	✗	✓	IoA	✓	✓	Stacey	✗	✓
NLPromptEval	✗	✓	c-seo-bench	✓	✓	BRIGHT	✗	✓
luckmatters	✗	✓	LCPO	✓	✓	CoMRes	✗	✓
AdaKV	✗	✓	gated_attention	✓	✓	Effort-AIGI-Detection	✗	✓
VerbalizED	✗	✓	jl-metric	✓	✓	SelfElicit	✗	✓
HeadKV	✗	✓	bookcoref	✓	✓	NeurIPS2025-KFF	✗	✓
Totals:	**Mode 0 (out of the box)** $18/60$			**REPO2ROCM (Mode 1)** $59/60$				

Table 5: Per-repository functional reproduction across the 60-repository benchmark. ✓ = verified GPU-backed execution; ✗ = failed. Venues span ICLR, ICML, ACL, and NeurIPS.

For auditability, the repositories used in the benchmark and agent-comparison experiments are listed below with their paper metadata and code links.

- **GCTM** (ICLR 2025): Generalized Consistency Trajectory Models for Image Manipulation. Code: link.
- **LayerDAG** (ICLR 2025): LayerDAG: Diffusion Model for Directed Acyclic Graph Generation. Code: link.
- **HARDMath** (ICLR 2025): HARDMath: A Benchmark Dataset for Challenging Problems. Code: link.
- **mm-argfallacy** (ACL 2025): Leveraging Context for Multimodal Fallacy Classification. Code: link.
- **EARTH** (ICML 2025): EARTH: Epidemiology-Aware Neural ODE. Code: link.
- **ELABORATION** (ACL 2025): ELABORATION: Benchmark on Human-LLM Competitive Programming. Code: link.
- **understanding_mcqa** (ICLR 2025): Answer, Assemble, Ace: Understanding How LMs Answer MCQs. Code: link.
- **Knowledge-Entropy** (ICLR 2025): Knowledge Entropy Decay Hinders New Knowledge Acquisition. Code: link.
- **VTI** (ICLR 2025): Reducing Hallucinations via Latent Space Steering. Code: link.
- **WildBench** (ICLR 2025): WildBench: Benchmarking LLMs with Tasks from Real Users. Code: link.
- **convolutional_diffusion** (ICML 2025): An analytic theory of creativity in convolutional diffusion models. Code: link.
- **modelmap** (ACL 2025): Embedding language models in probability space via log-likelihood vectors. Code: link.
- **DiffusionVeteran** (ICLR 2025): DiffusionVeteran. Code: link.
- **AlphaRec** (ICLR 2025): Language Representations Can be What Recommenders Need. Code: link.
- **TimeEmb** (NeurIPS 2025): TimeEmb: Lightweight Static-Dynamic Disentanglement. Code: link.
- **Self-Certainty** (NeurIPS 2025): Scalable Best-of-N Selection for Large Language Models Via Self-Certainty. Code: link.
- **pcx** (ICLR 2025): Benchmarking Predictive Coding Networks – Made Simple. Code: link.
- **CEB** (ICLR 2025): CEB: Compositional Evaluation Benchmark for Fairness in LLMs. Code: link.
- **Ladder** (ACL 2025): LADDER: Language-Driven Slice Discovery and Error Rectification. Code: link.
- **paperbanana** (arXiv 2026): PaperBanana: Automating Academic Illustration for AI Scientists. Code: link.
- **ACL25-CoPE** (ACL 2025): Modeling Uncertainty in Composed Image Retrieval via Probabilistic Embeddings. Code: link.
- **TOP_ERL_ICLR25** (ICLR 2025): TOP-ERL: Transformer-based Off-Policy Episodic Reinforcement Learning. Code: link.
- **SeqSNN** (NeurIPS 2025): Toward Relative Positional Encoding in Spiking Transformers. Code: link.
- **PrefixKV** (NeurIPS 2025): PrefixKV: Adaptive Prefix KV Cache Is What Vision Instruction Models Need. Code: link.
- **DataEnvGym** (ICLR 2025): DataEnvGym: Data Generation Agents in Teacher Environments. Code: link.
- **MCNC** (ICLR 2025): MCNC: Manifold-Constrained Reparameterization for Neural Compression. Code: link.
- **CLEME** (ACL 2025): CLEME2.0: Towards Interpretable Evaluation. Code: link.
- **m-rewardbench** (ACL 2025): M-RewardBench: Evaluating Reward Models in Multilingual Settings. Code: link.
- **Risk-Sensitive-CMDP** (ICML 2025): Online Learning in Risk Sensitive constrained MDP. Code: link.
- **FR-Spec** (ACL 2025): FR-Spec: Sparse Feature Representations. Code: link.
- **askqe** (ACL 2025): AskQE: Question Answering as Automatic Evaluation for Machine Translation. Code: link.
- **HarmoniCa** (ICML 2025): HarmoniCa: Harmonizing Training and Inference for Better Feature Caching. Code: link.

- **moa** (ICLR 2025): Mixture-of-Agents Enhances Large Language Model Capabilities. Code: link.
- **HealthGPT** (ICML 2025): HealthGPT: Medical Large Vision-Language Model via H-LoRA. Code: link.
- **topoloss** (ICLR 2025): TopoNets: Vision and language models with brain-like topography. Code: link.
- **SemEval2025-EAMT** (ACL 2025): UAlberta at SemEval-2025 Task 2: Prompting for Entity-Aware Translation. Code: link.
- **CATCH** (ICLR 2025): CATCH: Channel-Aware Multivariate Time Series Anomaly Detection. Code: link.
- **llm_reproducibility** (NeurIPS 2025): LLM Reproducibility. Code: link.
- **ORMind** (ACL 2025): ORMind: A Cognitive-Inspired End-to-End Reasoning Framework for OR. Code: link.
- **CSL-Mem** (ICML 2025): CSL-Mem: Towards Memorization Estimation: Fast, Formal and Free. Code: link.
- **torch_brain** (ICLR 2025): Multi-session, multi-task neural decoding from distinct cell-types. Code: link.
- **AIRMVC** (ICML 2025): Automatically Identify and Rectify: Robust Deep Contrastive Multi-view Clustering. Code: link.
- **TemporalHead** (ACL 2025): Does Time Have Its Place? Temporal Heads in Language Models. Code: link.
- **IoA** (ICLR 2025): Internet of Agents: Weaving a Web of Heterogeneous Agents. Code: link.
- **Stacey** (ICML 2025): Stacey: Promoting Stochastic Steepest Descent. Code: link.
- **NLPromptEval** (ACL 2025): What Makes a Good Natural Language Prompt?. Code: link.
- **c-seo-bench** (NeurIPS 2025): C-SEO Bench: Does Conversational SEO Work?. Code: link.
- **BRIGHT** (ICLR 2025): BRIGHT: A Realistic Benchmark for Reasoning-Intensive Retrieval. Code: link.
- **luckmatters** (NeurIPS 2025): Composing Global Solutions to Reasoning Tasks Via Algebraic Objects. Code: link.
- **LCPO** (ICLR 2025): Online Reinforcement Learning in Non-Stationary Environments. Code: link.
- **CoMRes** (ICLR 2025): CoMRes: Semi-Supervised Time Series Forecasting. Code: link.
- **AdaKV** (NeurIPS 2025): Ada-KV: Optimizing KV Cache Eviction By Adaptive Budget Allocation. Code: link.
- **gated_attention** (NeurIPS 2025): Gated Attention for Large Language Models. Code: link.
- **Effort-AIGI-Detection** (ICML 2025): Orthogonal Subspace Decomposition for Generalizable AI-Generated Image Detection. Code: link.
- **VerbalizED** (ACL 2025): Evaluating Design Decisions for Dual Encoder-based Entity Disambiguation. Code: link.
- **jl-metric** (ICLR 2025): Quality Measures for Dynamic Graph Generative Models. Code: link.
- **SelfElicit** (ACL 2025): SelfElicit: Your Language Model Secretly Knows Where is the Relevant Evidence!. Code: link.
- **HeadKV** (ICLR 2025): Not All Heads Matter: A Head-Level KV Cache Compression Method. Code: link.
- **bookcoref** (ACL 2025): BookCoref: Coreference Resolution at Book Scale. Code: link.
- **NeurIPS2025-KFF** (NeurIPS 2025): Class-aware Domain Knowledge Fusion for Continual Test-Time Adaptation. Code: link.
- **TarFlow** (ICML 2025): TarFlow: Normalizing Flows are Capable Generative Models. Code: link.
- **supercollapse** (ICML 2025): Scaling Collapse Reveals Universal Dynamics. Code: link.
- **UltraHR-100k** (NeurIPS 2025): UltraHR-100K: Enhancing UHR Image Synthesis. Code: link.
- **AnyEdit** (ICML 2025): AnyEdit: Edit Any Knowledge Encoded in Language Models. Code: link.
- **UniZyme** (NeurIPS 2025): UniZyme: A Unified Protein Cleavage Site Predictor. Code: link.