TRACECONE: EXACT DEPENDENCY REUSE FOR LOCALIZED REPOSITORY REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Repository-repair agents create diagnostic findings, patch fragments, and test outcomes whose validity must survive later failures. TraceCone makes this validity question an explicit execution operation. Each generated record carries the complete identity of every executable input it consumed; a failed contract queries those identities in two directions, recovering verified diagnostic context from incoming dependencies while invalidating the outgoing descendant closure. This gives learned executable state a precise preservation rule: records outside the closure remain reusable, while affected generated records are regenerated and affected checks are re-executed. We formalize the condition under which this operation is exact, distinguish it from content retrieval and learned cone prediction, and specify an evaluation contract that fixes units, endpoints, pairing, and budget accounting before any empirical aggregate is reported. The resulting contribution is a validity-maintenance interface for repository repair: persistent records become reusable state only when a system can identify the consumed inputs that justify preserving them.

1 INTRODUCTION

Repository repair is an incremental decision process. An agent may establish a diagnosis, run a targeted test, construct a patch fragment, and reuse the resulting state in several later edits. A later contract failure should identify the records whose executable inputs may have changed. The quality of that decision determines whether prior verified work is preserved and whether a fixed repair budget is concentrated on the affected portion of a task.

TraceCone represents this decision directly. It treats agent state as typed executable records linked by the complete identities of the records they consume. A failing contract exposes two distinct queries. An incoming query retrieves verified inputs for diagnosis. An outgoing query computes the inclusive descendant closure whose state may depend on the failed record. The executor freezes records outside that closure, regenerates mutable records inside it, and re-executes affected deterministic checks. The representation therefore turns a contract outcome into an explicit validity-maintenance operation.

This framing targets a gap between strong stateful repair systems and exact dependency maintenance. Repository agents already combine navigation, shell interaction, tests, and patch generation (Yang et al., 2024; Zhang et al., 2024; Wang et al., 2024). Knowledge distillation, graph reasoning, and addressable memory improve model compression, search, or state capacity (Hinton et al., 2015; Besta et al., 2024; Sukhbaatar et al., 2015; Graves et al., 2016). Incremental build systems, provenance, and truth-maintenance systems identify affected deterministic artifacts (Acar, 2005; Mokhov et al., 2018; Cheney et al., 2009; Kleer, 1986). TraceCone combines these perspectives for learned generation: content and next actions remain model-generated, while preservation is governed by recorded executable dependencies and contract outcomes.

The paper makes three contributions.

- It defines contract-triggered preservation semantics for generated repository-repair records, separating diagnostic ancestry from downstream invalidation.
- It gives a conditional exactness result: when every consumed executable input is represented and merges preserve those inputs, recomputation is confined to changed records and their descendants.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

- It provides a matched evaluation contract that distinguishes task success from clean-fix, fixes the unit of analysis and aggregation, and specifies how comparison controls obtain or do not obtain executable dependency identity.

2 TRACECONE

2.1 EXECUTABLE RECORD STATE

For an issue x, TraceCone maintains a directed acyclic record graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. A record v has a type $\tau(v) \in \{\texttt{claim}, \texttt{operation}, \texttt{observation}, \texttt{test}, \texttt{merge}\}$, structured content $c(v)$, a contract $\kappa(v)$, and an execution-input identity $I(v)$. The identity is an ordered, typed tuple containing the record's contract schema, the identities and contract outcomes of every record read while producing or executing v, and every repository object or tool result that the record consumes. The executor appends an edge (u, v) for each record identity in $I(v)$ that originates at u. Thus an edge means that executing v consumed executable state produced by u; it is not a similarity relation or an inferred after-the-fact association.

This representation creates a concrete interface between generated content and state maintenance. A claim record can consume an observation and a test outcome; an operation can consume a claim and a repository object; a test can consume an operation and a test specification. The record graph stores dependencies actually used by the executor, not a global program graph.

2.2 EXECUTION CONTRACTS AND EDGE EMISSION

An execution-input identity is more than a content hash. For each record, the executor retains a canonical, ordered read set that identifies the contract schema, each upstream record version and observed contract outcome, and every non-record input used by the execution. Non-record inputs include a repository revision or object identity, a tool invocation together with its returned value, and an evaluator or test specification. Canonicalization must preserve type and order whenever either can affect execution. A hash may be used as an index into this structured identity, but it is not itself the semantic evidence for a dependency.

The contract $\kappa(v)$ specifies the conditions under which the executor may retain v: its required inputs, the check to execute, and the outcome that marks the record valid. A record becomes eligible for reuse only after its read set, contract outcome, and emitted outgoing-edge list are committed together. In particular, an executor may not first reuse a record and only later infer the dependencies that would justify that reuse. This commit rule makes a missing edge a violation of the representation invariant, rather than an ambiguity resolved by a retrieval score.

The graph records execution dependence, not every semantic relation in an issue. Two records can discuss the same file without being adjacent. Conversely, a record that consumes a tool result or an earlier test outcome has an edge even if their rendered text has no lexical overlap. This distinction is essential when a repair changes a small object whose effect propagates through generated intermediate state rather than through a visible source-file relation.

2.3 VERSIONING NON-DAG EXECUTOR STATE

An executor can hold mutable state whose storage topology is not a DAG: a retrieval index can point to itself through summaries, a workspace can be revisited, and a controller can retain a recurrent hidden state. TraceCone does not require that this ambient state be erased or globally restructured. Instead, it represents a committed execution as an acyclic graph of record *versions*. A record version is created after the inputs it consumed have been fixed, so its emitted edges point to versions that were available at that execution step. Reading a mutable object after it changes produces a new object version and hence a distinct input identity.

This version discipline separates the operational graph from an executor's storage graph. A persistent non-DAG controller may retain arbitrary typed records and still use its own learned score to decide what to revisit. It becomes a TraceCone executor only when a record eligible for preservation exposes every consumed version as an emitted input edge. The distinction prevents a back-reference in a

memory store from being mistaken for an executable dependency, while allowing a repair system to retain rich state outside the validity-maintenance interface.

The transaction boundary also defines the treatment of concurrent work. A transaction reads a committed snapshot of the records outside its closure. A concurrent worker that produces a new input version does not silently alter that snapshot; its result is committed as a later version, with a contract outcome and emitted edges of its own. A subsequent transaction can then include that version in its triggering set if it changes the inputs of a previously reusable record. The construction does not prescribe a scheduler, but it requires the scheduler to make the identity boundary visible whenever reuse is claimed.

2.4 CONTRACT-TRIGGERED PRESERVATION

Let $S \subseteq \mathcal{V}$ be the records whose contracts fail after an execution step. TraceCone uses the inclusive outgoing closure

$$\mathcal{C}(S) = S \cup \{v \in \mathcal{V} : \exists s \in S,\ s \rightsquigarrow v\},$$

where $s \rightsquigarrow v$ follows outgoing record-dependency edges. Records outside $\mathcal{C}(S)$ retain their stored contents and contract status. Mutable generation records within the closure are regenerated in topological order, and deterministic checks within the closure are re-executed. Incoming edges are used only for diagnostic retrieval of verified inputs; they do not define the invalidation set.

2.5 REPAIR TRANSACTION

A repair transaction has four phases. First, the executor records the failed contract and inserts into S every record whose execution input or contract outcome has changed. Second, it traverses emitted outgoing edges to materialize $\mathcal{C}(S)$. Third, it freezes the committed state outside the closure and invalidates mutable state inside it. Finally, it regenerates invalidated records in a topological order, executing each contract once its recorded inputs are available. A newly failed contract starts a subsequent transaction; it does not retrospectively enlarge a completed transaction without recording a new triggering set.

This ordering separates two decisions that are often conflated in agent traces. Diagnostic retrieval asks which already verified inputs a worker should inspect before acting. Preservation asks which stored records remain licensed for reuse after an input changes. The former can follow incoming ancestry, search an index, or solicit a learned proposal. The latter is a validity decision and therefore follows the represented outgoing dependency relation. A learned system may still choose the next action inside the closure; the closure only determines which prior records cannot be treated as unchanged.

Freezing is also a contract discipline rather than a cache-eviction policy. A frozen record may be read by later work, but its contents and recorded outcome cannot be silently overwritten during the transaction. If a later execution needs a different input identity, it creates a new record version and an edge from the changed input. This permits the executor to distinguish genuine preservation from a replacement that merely reused the same storage location.

2.6 CONDITIONAL PRESERVATION GUARANTEE

Exactness has a narrow operational meaning: exact traversal of the graph induced by complete execution-input identities. It does not mean equality between independent stochastic generations or recovery from an omitted dependency. A merge is admissible only when its candidate records have equal type, structured content, contract schema, and full execution-input identity. A merge that drops one consumed repository object, tool result, or contract outcome is a new record, not a compression that preserves the original dependency relation.

Proposition 1 (represented-input preservation). Let S contain every record whose execution input or contract outcome changed. Assume that every consumed record identity emits its corresponding edge, replay is deterministic for a fixed execution-input identity, and merges preserve the complete identities of the records they represent. Then every $v \notin \mathcal{C}(S)$ has the same execution-input identity after the transaction as before it. Deterministic replay would consequently give v the same contract outcome.

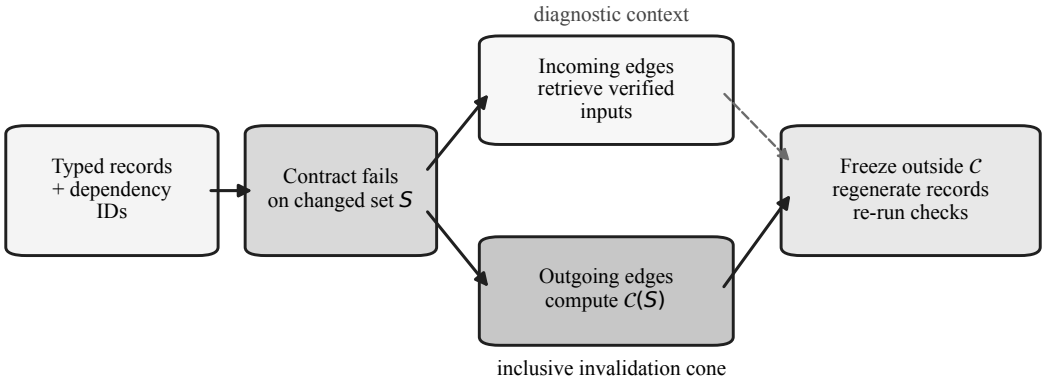


Figure 1: TraceCone preservation semantics. A failed `test` record reads incoming dependencies for verified diagnostic context and follows outgoing dependencies to form the inclusive invalidation closure. Records outside the closure remain reusable; affected generated records are regenerated and affected deterministic checks are re-executed.

To see this, suppose that $I(v)$ changed for a record outside the closure. A changed non-record input would place v itself in S. Otherwise, some changed record u was consumed while producing or executing v. Complete emission creates (u, v). Following changed consumed records backwards reaches a member of S, yielding an outgoing path from S to v, contradicting $v \notin \mathcal{C}(S)$. Input-preserving merges cannot remove that path, and deterministic replay gives the stated contract consequence. The result is a preservation condition over the represented execution graph; it makes no statement about unrecorded dependencies.

The assumptions expose the operational boundary of the claim. Contract checks may be learned, probabilistic, or incomplete, but their output is reusable only to the extent that the inputs consumed by their execution are represented. Similarly, a nondeterministic generation can remain part of the repair process, but an executor must either record the randomness and decoding configuration as inputs or decline to use the proposition to justify reuse across a change. The guarantee is thus a criterion for building a conservative state interface, not a claim that generation itself becomes deterministic.

2.7 Control boundary

TraceCone separates dependency identity from general state capacity. Table 1 gives the boundary. A flat retrieval system can reuse text without executable dependency identity. A persistent non-DAG executor can retain typed records and re-execute contracts without materialized record-to-record edges. A learned cone predictor may propose a repair region, but its proposal is not an exact dependency query. A content-hash control may reconstruct a candidate relation from record content; it does not receive the emitted dependency identities that TraceCone uses for invalidation. The table is a method specification, not an empirical ranking.

For a matched intervention, every family receives the same issue, executor budget, record types, contract schemas, and decoding schedule. TraceCone alone receives the emitted execution-input identity and follows its outgoing edges. The flat-retrieval and learned-cone families may retain content or propose candidate regions, but neither treats that proposal as an emitted exact edge. The persistent non-DAG executor retains state and can re-execute a contract without obtaining record-to-record identity. The content-hash reconstruction family can derive a new candidate relation from retained text, but its relation is not substituted for the executor's committed input identity. These conditions make the rows separable interventions instead of alternate names for a common edge oracle.

2.8 Schematic trace

Consider a schematic repair sequence with an observation o, a diagnostic claim d, a patch operation p, a test t, a merge record m, and an unrelated prior record q. Suppose that d consumes o; p consumes

Table 1: Matched control boundary. Execution-input identity denotes the complete emitted identity used by an executed record, not textual similarity, retrieval score, or a post-hoc program analysis.

Control family	Retained state	Repair-region source	Preservation operation
Flat retrieval	Serialized prior outputs	Retrieval over content	Regenerate selected text or a suffix
Persistent non-DAG executor	Typed records and contract outcomes	Learned repair scoring	Re-execute selected records
Learned cone predictor	Persistent records with predicted candidates	Predictor output	Regenerate the proposed candidate set
Content-hash reconstruction	Persistent records with content hashes	Equality over reconstructed content signatures	Reconstruct and regenerate a candidate set
TraceCone	Typed records with emitted input identities	Outgoing edges from failed records	Preserve the complement; regenerate the descendant closure

d; t consumes p; and m consumes d together with the versioned patch input. If the observation or the contract outcome for d changes, the outgoing closure contains d, p, t, and m, while q remains frozen. Incoming ancestry can retrieve o as diagnostic context, but it does not add o to the invalidation set unless its own contract or identity has changed.

The sequence illustrates why a closure is not simply a suffix of an agent transcript. The merge m may be created after the test, yet depend on d through a separate branch. Regenerating only a chronological suffix could retain m despite its changed diagnostic input. Conversely, regenerating every stored record discards q even though no represented input to q changed. The graph makes both decisions inspectable at the level of execution inputs. This is a schematic specification of the operation, not a benchmark trace or an empirical result.

If replaying p produces a distinct patch version, the executor records a new p' and re-executes downstream contracts that consume p. A successful replay does not license reuse of the old test merely because its text appears similar; the executor compares the complete recorded identity. If the new test fails, it becomes a new member of a triggering set for the next transaction. The resulting sequence supports repeated local repair while preserving a clear record of which state was retained under which contract.

3 Evaluation Contract

3.1 Population, unit, and endpoints

The evaluation unit is one issue–run pair (x, r), where x belongs to a versioned benchmark population P and r names one complete training and decoding run. Task success is the binary indicator that the benchmark's standard evaluation harness accepts the final patch for (x, r). A clean-fix is task success together with two additional conditions: all evaluator-defined regression checks pass, and every generated record changed during repair lies inside the executor's recorded invalidation closure for the failure that triggered that repair. A reported run-level rate must be

$$\frac{1}{|P|} \sum_{x \in P} y(x, r)$$

for the named binary endpoint y. The population, harness revision, issue list, and run identifier are part of the result record. This definition makes task success and clean-fix distinct while keeping their denominators explicit.

3.2 Aggregation, pairing, and budgets

For learned methods, each retained aggregate is the arithmetic mean of one run-level rate for each listed training seed; its standard error is the sample standard deviation of those rates divided by the square root of the number of listed seeds. Fixed external systems use separately labelled complete

decode runs and are not pooled with training-seed uncertainty. Pairwise comparisons match methods on the same issue and run schedule. The statistical unit is the issue; a two-sided paired permutation test samples signs of nonzero issue-level differences, and a paired bootstrap resamples issues. Any reported interval or p value must name this unit, the run schedule, the resampling count, and the tail rule.

Every budget record separates hard caps from realized values. Verifier calls, repair attempts, actions, and wall-clock time are measured per issue–run pair. A capped metric cannot exceed its stated cap; setup or teardown is reported separately rather than folded into a capped runtime.

3.3 Matched intervention protocol

The intervention point is a contract failure observed during an otherwise identical issue–run schedule. Before the first run, an evaluation records the benchmark revision and issue list, the fixed executor configuration, the decoding configuration, every training seed or external-system run identifier, and the contract that induces a repair transaction. The same evaluator version and endpoint definition apply to every compared method. These fields prevent a change in benchmark membership, harness revision, or run schedule from being misdescribed as an effect of dependency identity.

For a paired comparison, the ledger stores one binary endpoint outcome per issue for each member of a matched pair. A tie, win, or loss is derived only from these paired outcomes after the population and endpoint have been fixed. Strata are admissible only when every issue has an explicit stratum membership and the stratum denominator is retained with the outcome. A reported overall effect and a reported stratum effect must therefore be reproducible from the same issue-level table rather than from separately rounded summaries.

The protocol also distinguishes a hard resource limit from a measured resource use. The executor declares caps for actions, repair attempts, verifier calls, and elapsed time before a run. The ledger then records the realized value of each field for each issue–run pair. A time spent provisioning an environment or cleaning it up can be useful operational information, but it must be labelled separately from a capped repair runtime. The comparison is consequently meaningful even when different systems have different internal decompositions of their work.

3.4 Benchmark identity

A benchmark population such as SWE-bench is treated as a versioned benchmark with an explicit issue list (Jimenez et al., 2024). Derived candidate universes, high/low-reuse slices, post-freeze constructions, and cross-benchmark transfer claims require their own versioned inclusion manifest and outcome ledger before they are used as evidence. The evaluation contract therefore ties every retained aggregate to one declared population instead of using a benchmark name as a proxy for an unspecified cohort.

3.5 Result admissibility

The contract has a simple admission rule for tables, figures, and prose claims. A result record must identify the benchmark revision and issue list; the issue–run key; the binary endpoint definition and denominator; any pairing key and stratum membership; the realized and capped budget fields; and the aggregation, uncertainty, and testing rule. A visualization and its cited prose must be generated from this same record set. The rule makes the displayed object auditable without requiring a reviewer to infer a denominator from rounded percentages or reconstruct a seed cohort from a caption.

This requirement applies equally to positive and null comparisons. If an aggregate is a mean over training seeds, the listed seed cohort determines both the mean and its standard error. If a baseline is a fixed external system, its complete decode runs are labelled separately rather than being treated as training seeds. If a percentage is reported for a run, it is the mean of the stated binary issue outcomes over the declared population. These are definitions of what an admissible report must contain; they do not substitute a protocol for missing outcome records.

For a clean-fix endpoint, the result record additionally identifies the evaluator-defined regression checks and the closure used for the relevant repair transaction. This makes it possible to distinguish a patch accepted by a benchmark harness from a patch whose changed generated state remained inside

the recorded repair region. The two endpoints can coincide on an issue, but they answer different questions and must retain their own binary outcomes before either is aggregated.

4 Connections to Prior Systems

Repository-repair agents motivate the operational setting. SWE-agent describes agent-computer interfaces for automated software engineering, AutoCodeRover targets autonomous program improvement, and OpenHands provides a platform for generalist AI software developers (Yang et al., 2024; Zhang et al., 2024; Wang et al., 2024). These systems demonstrate why a repair process accumulates heterogeneous state across navigation, commands, tests, and patches. TraceCone focuses on a complementary question: which previously generated record remains justified when a later contract exposes a changed execution input.

The representation also differs from methods that expand the capacity or organization of learned state. Knowledge distillation compresses information into a deployable model (Hinton et al., 2015); Graph of Thoughts represents generated units and their dependencies as a graph (Besta et al., 2024); and memory networks and differentiable neural computers provide addressable external memory (Sukhbaatar et al., 2015; Graves et al., 2016). Each perspective can inform an executor's content generation or retrieval. TraceCone's additional constraint is that a preservation decision is licensed by identities emitted during execution, rather than by a graph inferred from generated content or by the existence of stored memory alone.

Finally, the formal boundary draws on incremental computation, build systems, provenance, and truth maintenance. Self-adjusting computation tracks dependencies to propagate changes (Acar, 2005); build-system models describe dependencies and rebuild scheduling (Mokhov et al., 2018); database provenance studies why, how, and where data arise (Cheney et al., 2009); and assumption-based truth maintenance maintains consequences under changing assumptions (Kleer, 1986). TraceCone adapts the shared dependency-maintenance intuition to generated repair records. Its claim is conditional because learned content and next actions can vary, while the recorded input relation provides the object over which preservation is decided.

5 Operational Consequences

5.1 Record lifecycle

The representation gives every reusable record a small but consequential lifecycle. A record is pending while its execution is incomplete. It becomes committed only after the executor stores its canonical input identity, contract outcome, and emitted edges together. A committed record is either retained for a subsequent transaction, frozen while an unrelated closure is regenerated, or invalidated when it belongs to the closure of a changed record. Regeneration produces a later version rather than modifying a retained record in place. This lifecycle makes a repair trace interpretable even when the same worker revisits a file, a tool, or a memory location several times.

The distinction between invalidation and deletion matters for diagnosis. An invalidated record remains part of the prior execution history, so a reviewer or executor can inspect which input version caused it to lose reuse eligibility. A regenerated version can then identify its new consumed inputs and contract result. This history does not require keeping every generated token as executable state: it requires keeping the structured identity and contract evidence for records whose preservation is consequential. An implementation may compact rendered content or externalize bulky objects, provided that compaction does not discard an input identity needed to evaluate a later reuse decision.

5.2 Fail-closed preservation

An executor can be conservative without recomputing an entire issue. When it cannot establish that a consumed input is unchanged, it treats the corresponding record as invalidated and follows its emitted descendants. This fail-closed rule does not convert a learned candidate region into a proof of irrelevance. A learned predictor can prioritize which invalidated record to regenerate first, and retrieval can provide useful diagnostic context, but neither replaces the closure when the system asserts that a record was preserved.

The rule also clarifies the role of contract failures. A failed test is not automatically evidence that every preceding action was wrong; it is evidence that the test record and any descendant state require a new execution under the recorded inputs. The incoming ancestry remains useful because it tells the executor which prior observations and claims were consumed in producing the failed state. Keeping diagnostic retrieval separate from invalidation avoids two symmetric errors: discarding verified upstream evidence merely because a downstream check failed, and retaining a downstream merge that depended on an invalidated intermediate claim.

5.3 Auditing a reuse decision

For each transaction, an audit record can state the triggering set, the closure produced by emitted edges, the frozen committed records, the regenerated versions, and the contracts re-executed. These fields permit a reader to ask a precise question about any retained record: which complete execution-input identity remained unchanged, and why is the record outside the transaction's closure? The answer is structural rather than rhetorical. It can be checked against the stored edge relation and the transaction's input-version changes without inferring intent from a natural-language trace.

The same audit record makes disagreement between systems interpretable. If a content-retrieval control regenerates fewer records, the difference may arise from a learned selection rule rather than a claim that the omitted records have unchanged executable inputs. If a non-DAG executor retains more state, the difference may reflect storage capacity rather than an exact preservation operation. A matched comparison can therefore report both endpoint outcomes and the intervention boundary that produced them, while keeping the two kinds of evidence distinct. This is the practical value of treating dependency identity as an explicit interface: it turns state reuse from an informal optimization into a decision whose premises are inspectable.

6 Discussion

TraceCone supplies a design rule for stateful repair: preservation should be licensed by the executable inputs that a record actually consumed. This makes the distinction between storage and validity maintenance concrete. More records, a stronger retrieval mechanism, or a learned repair-region predictor can increase an executor's capacity; they do not by themselves establish which prior records remain justified after a contract failure. Materialized input identity supplies that missing decision interface.

The same distinction clarifies how repository repair relates to incremental computation. In a build system, dependencies and artifacts are typically deterministic. In TraceCone, record content and the next action are model-generated, but the preservation decision is still made over an executed dependency relation. The conditional guarantee isolates the portion of the operation that can be reasoned about independently of model stochasticity.

7 Conclusion

TraceCone reframes reusable repair state as a validity-maintenance problem. By recording complete execution-input identities, a failed contract can recover verified diagnostic context and invalidate exactly the represented descendant closure. The resulting interface preserves justified records and focuses regeneration on records whose inputs may have changed. This operational separation between state capacity and dependency-directed preservation gives repository-repair systems a precise basis for reusing verified work across localized failures.

AI Use Statement

Generative AI was used only to assist with writing, editing, and presenting the manuscript, including its conceptual and formal exposition. It was not used to generate empirical measurements, experimental outcomes, human judgments, or citations for this work. The authors are responsible for the final manuscript.

REFERENCES

Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, 2005. URL https://csd.cmu.edu/academics/doctoral/degrees-conferred/umut-a-acar.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682–17690, 2024. doi: 10.1609/aaai.v38i16.29720. URL https://ojs.aaai.org/index.php/AAAI/article/view/29720.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009. doi: 10.1561/1900000006. URL https://doi.org/10.1561/1900000006.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626): 471–476, 2016. doi: 10.1038/nature20101. URL https://www.nature.com/articles/nature20101.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. URL https://arxiv.org/abs/1503.02531.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986. doi: 10.1016/0004-3702(86)90080-9. URL https://doi.org/10.1016/0004-3702(86)90080-9.

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. Build systems à la carte. *Proceedings of the ACM on Programming Languages*, 2(ICFP):79:1–79:29, 2018. doi: 10.1145/3236774. URL https://doi.org/10.1145/3236774.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *Advances in Neural Information Processing Systems*, volume 28, 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/hash/8fb21ee7a2207526da55a679f0332de2-Abstract.html.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024. URL https://arxiv.org/abs/2407.16741.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024. doi: 10.1145/3650212.3680384. URL https://arxiv.org/abs/2404.05427.