

Runnable Commit Untangling for Coding Agents

Jinfeng Jiang^{1,4}, Dongsun Kim^{1,†}, Dayi Lin², Zhou Yang^{3,4,5,†}

¹Korea University, ²University of Waterloo, ³University of Alberta,

⁴Alberta Machine Intelligence Institute (Amii), ⁵Canada CIFAR AI Chair

[†]Corresponding authors

Coding agents produce *large, tangled patches* that mix multiple development purposes, making the code hard to review and maintain. Commit untangling offers the promise of organizing such large patches into *untangled, manageable commits*. This paper emphasizes two important limitations in existing commit untangling studies. First, they do not consider that untangled commits are ordered and should leave the code *runnable*. In practice, maintainers are unlikely to accept commits that prevent the code from running. Second, existing studies claim that commit untangling helps software maintenance. However, they conduct syntactic comparisons between the untangled commits and developers’ original commits without directly showing the claimed maintenance benefits. To address these gaps, this paper makes two novel contributions: ① RUC-TANGLE, the first agentic method that untangles commits while keeping the code runnable after each commit; and ② TANGLE-EVAL, the first evaluation framework that quantifies how untangled, manageable commit histories help coding agents repair bugs. We compare RUC-TANGLE against four untangling methods on 131 agent-generated patches. All histories produced by RUC-TANGLE are runnable, while baselines produce 20.6%–37.4% unrunnable commit histories. We further collect 453 agent-generated patches that introduce regressions (i.e., causing previously passing tests to fail) and ask two other coding agents to repair regressions. Augmenting agent context with RUC-TANGLE-produced histories yields 5.2% absolute improvement in pass@1. We also analyze agent trajectories to learn how they use untangled commits to navigate and fix bugs. Our findings demonstrate the value of adopting established software engineering practices in the era of coding agents, which broaden the future research agenda: how can agents actively use software history to make better development decisions?

Date: October 8, 2026

Correspondence: darkrsw@korea.ac.kr; zy25@ualberta.ca



1 Introduction

“Change the programs in small steps, so if you make a mistake, it is easy to find where the bug is.”

— Martin Fowler [8]

Coding agents are increasingly used in software development, but their output still requires review and maintenance [37]. Google reported that AI generated 75% of its new code in April 2026, with engineers reviewing and approving the output [28]. Agent-written patches are often *tangled*, containing changes that serve multiple development purposes, such as implementing a feature and refactoring existing code [36, 13]. These patches are also substantially larger: on SWE-bench Verified, agent-written patches are twice as large as human-written patches [24]. When different purposes are mixed in a large patch, reviewers must distinguish the intended behavior changes from edits that should preserve existing behavior. If the patch introduces a bug, developers must also identify which of these changes caused it. Organizing these changes so that they can be inspected separately is therefore important for reviewing and maintaining agent-generated code.

Commit untangling aims to organize a tangled patch into smaller commits, each containing changes that serve a single development purpose [30, 27, 19]. Prior work argues that separating changes by purpose makes

them easier to understand and review [30, 19]. These studies also motivate untangling through its potential benefits for software maintenance. For example, separating unrelated changes can help developers narrow down which changes introduced a bug [27]. It can also let developers revert an unwanted feature without discarding an unrelated bug fix included in the same patch [30, 42]. However, this paper emphasizes *two key limitations in how existing studies construct and evaluate untangled commits*:

Limitation 1: Ignoring Runnability. Existing untangling methods [30, 27, 42] do not check a practical and fundamental requirement for commit histories: the code should remain *runnable* after each commit. Applying a sequence of commits to a repository produces a project version after each step. We refer to the versions between the initial and final versions as *intermediate versions*. In modern development, code review and continuous integration (CI) checks are performed before changes are accepted [9]. A commit that prevents the software from running is likely to be rejected. For example, the Linux kernel guidelines [23] explicitly require the kernel to build and run properly after every patch in a series, so that developers can inspect the history to locate bugs. Thus, we emphasize that *runnability is a prerequisite of untangled histories*. Yet existing studies [30, 27, 42] do not enforce this requirement in method design and evaluation, limiting practical use.

Limitation 2: Unvalidated Maintenance Benefits. Existing studies claim that commit untangling helps developers review changes and locate bugs, but their automated evaluations do not directly test these practical benefits [27, 19, 42]. A widely used evaluation setting combines several developer-written commits from repository history into a single, synthetic tangled patch [27, 19]. Researchers then ask an untangling method to split this patch and measure how accurately it recovers the original commits. A higher score therefore indicates closer agreement with the developers’ original grouping decisions. Atomizer’s additional evaluation on expert-annotated real commits also measures agreement with the reference commits [42]. However, *existing evaluation does not establish the practical benefits of generated commits*, especially in the era of AI coding.

These practical limitations motivate us to make two novel contributions: ① RUCTANGLE, the first agentic method that untangles patches while keeping the code runnable; and ② TANGLEEVAL, the first evaluation framework that quantifies how untangled commits help coding agents.

RUCTANGLE (Runnable Commit UnTangle) is an agentic commit untangling method. Given a tangled patch and the initial project version, an agent groups the changes by development purpose and proposes an ordered sequence of commits. RUCTANGLE applies these commits one by one and checks whether the code remains runnable. When a check fails, it first uses the error message to adjust the commit order. If failures persist, it asks the agent to revise the grouping and order. If these revisions still fail, it merges adjacent commits to remove versions that cannot run. It repeats the checks after each adjustment while preserving the final code produced by the original patch. We compare RUCTANGLE with four baselines on 131 patches produced by a Claude Opus 4.6 agent across 100 Python repositories in SWE-CI [5]. RUCTANGLE produces a history for every patch, with the code after every commit passing the runnability checks. By comparison, 20.6%–37.4% of the histories returned by the baselines contain at least one version that fails runnability checks.

TANGLEEVAL lets coding agents repair regressions in tangled, agent-generated patches. We collect 453 patches generated by six agents from SWE-CI [5], each causing previously passing tests to fail. We ask two other coding agents, powered by Qwen3-Coder-Next and Nemotron-3.5-Lightning-30B-A3B (using mini-swe-agent [33] as the harness), to repair these regressions. For each patch, the repair agents start from the same failing code. We compare repair without untangled commits against repair with histories generated by RUCTANGLE, Armchr [1], or Atomizer [42]. Because RUCTANGLE keeps the code runnable after each commit, we can run the failing tests with `git bisect` to identify a candidate commit that introduced the regression. We add this result to the repair agent’s context. Compared with repair without untangled commits, RUCTANGLE raises pass@1 from 26.9% to 31.3% for Qwen and from 30.8% to 36.0% for Nemotron. The average relative improvement across the two models is 16.6%. Armchr and Atomizer also improve repair performance, but yield smaller average relative gains of 4.5% and 6.1%, respectively.

To better understand how untangled commits help coding agents repair bugs, we conduct case studies comparing agent trajectories with and without these commits. We find that agents use commit diffs and messages to identify behavior that a repair should preserve. However, in one attempt, we also observe agents repeatedly reconsidering a correct repair when it conflicts with a misleading commit message.



To summarize, we make the following contributions:

- **Research Gap** (Section 2): We identify two practical limitations of existing commit untangling research: existing commit untangling methods do not check whether the code remains runnable after each commit, and current evaluations neither consider real-world development constraints nor demonstrate the practical benefits of untangling for subsequent software development.
- **Approach** (Section 3): We propose RUCTANGLE, the first commit untangling method that produces runnable commits by checking runnability after each commit and uses execution feedback to revise the results.
- **Evaluation** (Sections 4 and 5): We propose TANGLEEVAL, the first evaluation framework that measures how untangled commit history helps coding agents repair bugs.
- **Implication** (Section 6): We provide the first empirical evidence that untangled histories can improve coding agents’ bug-fixing performance. This finding motivates using rich histories in software development.

2 Motivation and Background

2.1 Commit Untangling and Runnability

Developers use commits to record changes to a project [4]. A patch is *tangled* when it combines changes serving different development purposes, such as fixing a bug and performing an unrelated refactoring [11]. *Commit untangling* aims to separate these changes into smaller commits, each focused on one purpose [27, 30, 19, 42]. This separation allows developers to inspect related changes together. In a controlled study, Tao and Kim found that participants answered review questions more accurately when a tangled change was presented as separate groups of related changes [34].

The resulting commits should also keep the code *runnable* so that developers can use the history for testing and debugging. Applying the commits in order produces a project version after each step; the versions between the initial and final versions are the *intermediate versions*. The Linux kernel guidelines explicitly require the kernel to build and run properly after every patch in a series, because tools such as `git bisect` may inspect the code at any of these points [22]. Runnability is distinct from functional correctness: code may run and still fail a test. Checking only the completed patch does not establish whether the code remains runnable after each earlier commit.

Figure 1 illustrates this problem with a simplified patch. The patch adjusts the logging level through Change A and switches the default backend through Changes B and C. Change B replaces the `local` entry in the backend dictionary with `sqlite`, while Change C updates `DEFAULT_BACKEND` from `local` to `sqlite`. When `bootstrap.py` is imported, it selects the backend by evaluating `BACKENDS[DEFAULT_BACKEND]`. The selected name must therefore exist in the dictionary.

Suppose each change forms a separate commit. After A and B have been applied, the dictionary contains only `sqlite`, but the default remains `local`. Any test that imports `bootstrap.py` then fails to load with a `KeyError`. Reversing B and C does not solve the problem: applying C first selects `sqlite` while the dictionary still contains only `local`. Grouping B and C into one commit avoids both failures, while A can remain in a separate commit. The runnability check exposes this mismatch, allowing the untangling method to revise its grouping. The example highlights that runnability is a fundamental requirement in commit untangling.

2.2 Task Formulation

Let V_0 denote the project version before a patch P , and let $V_\star = \text{apply}(V_0, P)$ denote the version after applying it. Our task is to reorganize the changes in P into a commit history $C = \langle c_1, \dots, c_k \rangle$. Each commit contains at least one change, and every change in P belongs to exactly one commit. The commits must be applicable in sequence, producing

$$V_i = \text{apply}(V_{i-1}, c_i), \quad 1 \leq i \leq k.$$

The final version must satisfy $V_k = V_\star$. Thus, untangling changes how the patch is organized without changing its final code. We aim to produce smaller commits organized by development purpose, while

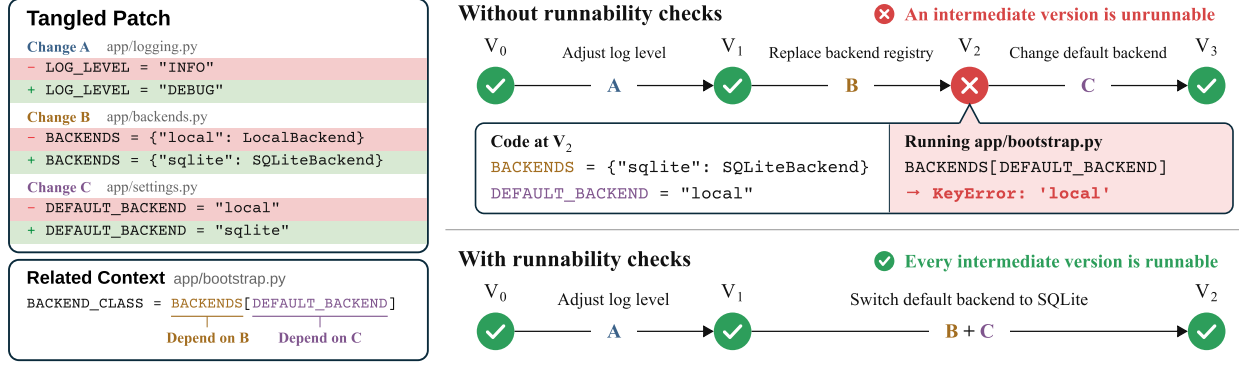


Figure 1. A simplified patch with two development purposes: adjusting the logging level (A) and switching the default backend from local to sqlite (B and C). Change B replaces the entry in the backend dictionary, and Change C updates the default selection. Because `bootstrap.py` evaluates `BACKENDS[DEFAULT_BACKEND]` at import time, separating B and C causes a `KeyError` between these commits. Keeping A separate and grouping B with C avoids this mismatch.

keeping together changes that cannot be separated without breaking runnability.

For the Python projects in our study, we assess runnability through test collection. A version passes the runnability checks when `pytest` can discover and load its tests in the project environment. Let $R(V) = 1$ when version V passes these checks, and $R(V) = 0$ otherwise. *Runnable commit untangling* additionally requires the resulting commit history to be *runnable*:

$$R(V_i) = 1, \quad 1 \leq i \leq k,$$

that is, the code passes the runnability checks after every commit, including the last one. If $V_\star$ fails these checks, preserving its code and satisfying this condition are incompatible.

We do not require functional tests to pass. A runnable version may still contain a regression, including one introduced by the original patch. Untangling preserves that patch; repairing its errors is a separate task. A history containing only one commit may satisfy the requirements above, so we evaluate how well the patch is split as well as whether the resulting versions are runnable.

2.3 Evaluating Untangled Commits

Tangled patches in practice often lack a reference showing how their changes should be grouped (i.e., the ground truth of untangled commits). A common evaluation therefore combines several developer-written commits into one tangled patch and records which original commit each change came from [27, 19]. Methods are then scored on how closely their groups reproduce the original grouping [14, 42]. Although the metrics differ in which code elements they count, they share this focus on agreement with reference groups. The original commits provide a reference for comparison, but do not necessarily represent the only reasonable way to split the patch.

These grouping scores do not evaluate the runnability. The consequences of a grouping error depend on which changes are separated. In Figure 1, separating B and C makes `bootstrap.py` fail to load after the first of these two changes, regardless of their order. Grouping scores also do not assess commit order: rearranging the same groups leaves their scores unchanged, even if the new order imports a module before the commit that introduces it. Finally, agreement with reference groups does not directly measure whether the history helps developers review changes or repair bugs. These uses require evaluation beyond grouping accuracy.

TANGLEVAL evaluates both the generated histories and their use in regression repair. For each history, it checks whether the commits can be applied in order from V_0 , reproduce $V_\star$, and leave every resulting version runnable. We also report commit counts to show how finely each patch is split. For the repair evaluation, we use agent-generated patches that cause previously passing tests to fail. For each patch, repair agents start from the same failing code and tests under the same repair budget. We compare repair without untangled

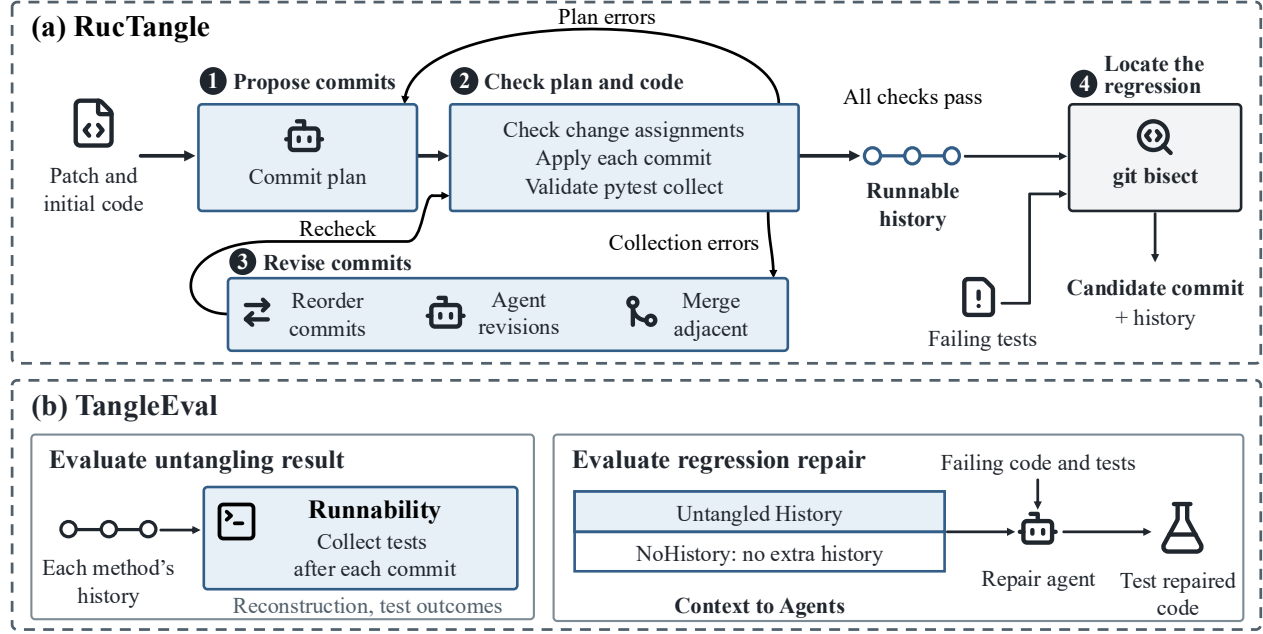


Figure 2. RUCtangle and its use in TangleEval. (a) RUCtangle checks and revises agent-proposed ordered commits to keep each version runnable, then uses `git bisect` to identify a candidate commit for regression repair. (b) TangleEval evaluates untangling results and agents’ performance in regression repair with and without untangled histories.

commits against repair with histories produced by different methods. We measure how often agents fix the failing tests without causing other previously passing tests to fail.

3 Methodology

This section describes how RUCtangle constructs commit histories that ensure runnability, and how TangleEval evaluates these histories and their use in regression repair.

3.1 RUCtangle: Runnable Commit Untangling

Given a patch P and the initial project version V_0 , RUCtangle asks an agent to group the changes and propose an order for the commits. It then applies the commits in order and checks the runnability of each resulting intermediate version. Failures guide adjustments to the grouping and order, as shown in Figure 2. Throughout this process, the final code must remain equal to V_* . The procedure below assumes that V_* passes the runnability checks defined in Section 2.2; otherwise, preserving its code and keeping every version runnable are incompatible.

Step 1: Proposing Commits. RUCtangle first divides P into code changes for the agent to group. Whole-file additions and deletions are initially treated as single changes. The agent examines these changes and the project code to propose a *commit plan*: an ordered list of commits, each containing a group of changes and a message describing its intended purpose. The agent groups changes by development purpose and considers their dependencies when choosing the order. An initial change may contain changes serving several purposes, as can happen when a newly added file contains several features. The agent can use the `split` tool to replace such a change with smaller changes that preserve the same edits. It then submits the completed plan through the `submit` tool.

Step 2: Checking the Plan and Code. After the agent submits a commit plan, RUCtangle validates it in two stages. First, it performs structural validation to ensure that the plan is well formed, that every change is assigned to exactly one commit, and that each commit contains at least one change. Formatting or assignment

errors are returned to the agent for correction and resubmission. Next, starting from V_0 , RUCANGLE applies the planned commits in order to construct intermediate versions, and checks their runnability following the definition in Section 2.2.

Step 3: Revising the Commits. When a runnability check fails, RUCANGLE first tries reordering the commits. For example, an error may indicate that a required module is introduced only by a later commit; moving that commit earlier may resolve the failure without changing the commit groups. If failures remain, RUCANGLE starts a new agent session with the current plan and the validation error. The agent revises the plan by regrouping or reordering changes, and further splitting changes when necessary. RUCANGLE allows up to two such revision sessions. If an intermediate version V_i remains unrunnable after two agent revisions, RUCANGLE removes the corresponding commit boundary by merging c_i with c_{i+1} . If necessary, it progressively merges subsequent commits until the version after the merged commit becomes runnable. This fallback removes the unrunnable intermediate version while preserving all changes, at the cost of producing fewer commits. After each revision, RUCANGLE revalidates the resulting plan and intermediate versions using the checks in Step 2, and accepts the plan only when all checks pass.

Step 4: Locating the Regression. This step uses the runnable history to provide additional information for regression repair. When P introduces a regression, we denote $V_\star$ as V_{fail} , and let T_{fail} denote the tests that pass at V_0 and fail at V_{fail} . After constructing the runnable history, RUCANGLE runs the tests in T_{fail} on versions selected by `git bisect`. The test outcomes guide the search for a candidate commit that introduced the regression. RUCANGLE provides the generated history and the identified candidate, including its commit message, as context for the repair agent.

3.2 TANGLEVAL: Evaluating Untangled Commit Histories

3.2.1 Evaluating Untangling Results

To evaluate whether the untangling results are suitable for software development, TANGLEVAL selects large, agent-generated patches that contain changes serving multiple development purposes as the inputs for untangling, and evaluates the resulting commit histories using the following metrics:

Patch reconstruction rate. A valid commit history should reconstruct the original patch: its commits can be applied sequentially from V_0 , and the resulting final version matches $V_\star$. This rate is the proportion of input patches for which a method produces a valid history.

Runnability. TANGLEVAL checks whether the version produced after each commit passes the runnability checks defined in Section 2.2. For a history C containing k commits, the runnability rate $\text{Run}(C)$ is the proportion of the resulting versions $V_1, \dots, V_k$ that pass these checks. TANGLEVAL reports the mean runnability rate across all untangling results, together with the proportion of histories for which $\text{Run}(C) = 1$. The latter measures how often a method produces a history in which every intermediate version is runnable.

Intermediate test failures. When commits are applied sequentially, an intermediate commit may temporarily break behavior that has already been established by preceding commits. A desirable untangling should minimize such temporary regressions. TANGLEVAL therefore examines whether applying a commit causes a previously passing test to fail. Let T contain the tests that pass in the final version $V_\star$. For each untangled history, $\text{ITF}(C)$ is the proportion of tests in T that experience a pass-to-fail transition after applying a commit. A lower $\text{ITF}(C)$ indicates fewer temporary regressions.

Commit count. The preceding metrics should be interpreted together with the degree of patch splitting, because keeping the entire patch in a single commit can reconstruct the final code without introducing any intermediate versions. TANGLEVAL therefore reports the mean and median number of commits, together with the proportion of histories containing only one commit. These measurements characterize how finely each method splits the input patch and serve as auxiliary metrics for interpreting the preceding results.

Distribution of test gains. Commit untangling aims to generate commits that each serve a distinct development purpose. However, directly evaluating development purposes at scale is difficult in an automated evaluation.

We therefore use the distribution of functional gains across commits as a proxy, with passing tests providing observable evidence of the corresponding behavior. If functional gains are concentrated in only one commit, the generated history may provide limited decomposition of the patch’s functionality; conversely, gains distributed across multiple commits indicate that tested behavior is established progressively across the history. Let T_{F2P} contain the tests in T that fail at V_0 . These fail-to-pass (F2P) tests [15] provide evidence of functional gains between the initial and final versions. For each F2P test, TANGLEVAL assigns the test to the commit at its last transition from failing to passing in the history. The test passes after this commit and at every subsequent version through $V_\star$. Let q_i be the proportion of these tests assigned to commit i . TANGLEVAL reports this distribution using the largest share assigned to one commit, $\text{MTP}(C) = \max_i q_i$, and its normalized entropy:

$$\text{TPE}(C) = -\frac{1}{\log k} \sum_{i: q_i > 0} q_i \log q_i, \quad k > 1. \quad (1)$$

We define $\text{TPE}(C) = 0$ when $k = 1$. A larger MTP indicates a greater concentration of test gains in one commit, while a larger TPE indicates a more even distribution across commits.

3.2.2 Evaluating Regression Repair

To evaluate whether untangled commit histories can help coding agents repair regressions, TANGLEVAL constructs repair tasks from agent-generated patches that introduce regressions. For each task, V_0 is the version before the patch, and V_{fail} is the resulting version that contains the regression. The set T_{fail} contains tests that pass at V_0 but fail at V_{fail} , representing the regression to be repaired. TANGLEVAL reconstructs the Git history from V_0 to V_{fail} using the commits produced by the evaluated untangling method and makes this history available to the repair agent. For RUC TANGLE, whose reconstructed history contains runnable intermediate versions, it additionally runs the tests in T_{fail} on the versions selected by `git bisect` and provides the resulting bisect information as additional context to the repair agent. The agent may then generate a *repair patch*, which is applied to V_{fail} to obtain the repaired version V' . TANGLEVAL evaluates the agent’s performance by running T_{fail} as well as T on V' . Because agent behavior can vary across independent repair attempts, TANGLEVAL supports repeated attempts.

TANGLEVAL evaluates regression repairs using three metrics. An agent achieves a *partial repair* if at least one test in T_{fail} passes at V' , a *full repair* if all tests in T_{fail} pass, and a *successful repair* if, in addition, every test in T passes. TANGLEVAL reports `pass@1`, which is the proportion of attempts that achieve a *successful repair*. With multiple attempts per patch (e.g., three), TANGLEVAL also reports `pass@3`, which is the proportion of repair patches with at least one *successful repair*.

To characterize repair progress beyond repair success, TANGLEVAL also reports the proportion of attempts that achieve *partial repair* and *full repair* to provide a more comprehensive view of the agent’s performance. Meanwhile, agents usually repair regressions through a process that involves localization, editing, and testing. TANGLEVAL records the repair patch generation rate as the proportion of attempts whose final version differs from V_{fail} . This rate serves as a coarse indicator of whether the agent progresses beyond regression localization to implementing a candidate repair.

4 Experimental Setup

4.1 Datasets

4.1.1 SWE-CI

SWE-CI [5] evaluates how coding agents extend and maintain existing software over multiple development iterations. Each task starts from an earlier version of a repository and uses tests from a later version of the same project to specify the expected behavior. The agents must modify the earlier code until it passes these tests, which may require adding new functionality or updating existing behavior. In each iteration, the benchmark runs the tests on the current code. An architect agent uses the test failures and the code to identify missing or incorrect behavior and write development requirements. A programmer agent receives these requirements and modifies the repository accordingly. The updated code is tested again, and this cycle continues until the tests pass or the iteration limit is reached. We use the published release containing 226 tasks with container images [31], together with the released agent runs [32]. These runs record the code and test outcomes after each development iteration. By comparing the initial code with later recorded versions, we extract the cumulative patches used in our evaluation.

4.1.2 Dataset Construction

We construct two datasets for different evaluation purposes. For RQ1, we examine whether splitting a completed patch preserves the final code and leaves runnable versions after individual commits. For RQ2, we select patches with regressions and evaluate whether untangled commits help another agent repair them.

Before comparing code versions, we remove a fixed set of files created during execution, such as `pytest` caches and coverage reports. We retain project code, tests, configuration files, and generated source files. We also run tests three times under the task’s container configuration to identify consistently passing tests.

For RQ1, we use released Claude Opus 4.6 runs whose final records report completion of the SWE-CI task. The input patch contains the changes from the initial version, V_0 , to the last recorded version, $V_\star$. When runs share the same initial code and execution environment, we keep only one run, giving priority to the earlier release batch. We retain tests that pass in all three runs at $V_\star$ and require at least one test that does not pass at V_0 but passes at $V_\star$. This provides observable changes in test outcomes for evaluating the generated commit sequence. After filtering, the dataset contains 131 patches from 100 Python repositories.

For RQ2, we select code versions in which an agent’s changes cause previously passing tests to fail. We use runs from Claude Opus 4.6, GLM-5.1, DeepSeek-V4-Pro, Kimi-K2.6, Qwen3.7-Max, and GPT-5.5. Within each run, we track tests that pass in the initial version and record their first failing iteration and first subsequent recovery iteration. We retain failing versions with tests that pass immediately before the modification, fail after it, and later recover in the same run. After recovery, these tests have no further recorded failures and pass in the run’s final version, providing evidence that the corresponding regressions are repairable.

For each retained failing version, V_{fail} , we extract the cumulative patch from the initial version, V_0 , to V_{fail} as the untangling input. The repair targets T_{fail} contain all tests that pass at V_0 but fail at V_{fail} . We also retain the set T of tests that pass at V_{fail} to detect new regressions introduced by a repair. The resulting dataset contains 453 patches from 108 tasks in 90 repositories. A task can contribute multiple patches through different generating models or development iterations. Each generating model contributes between 28 patches (GPT-5.5) and 97 patches (Kimi-K2.6).

Figure 3 shows the sizes of the patches in both datasets. Both have a median of seven modified files per patch. The median numbers of hunks are 21 for RQ1 and 24 for RQ2.

To reduce evaluation cost, four repair models (Section 4.3.2) and the commit-merging analysis (Section 6.1) use the same 150-patch sample. We draw this sample from 434 candidates for which RUCANGLE produces a history with at least four commits and completes the bisect step. The four-commit requirement ensures that merging commits to halve their number still leaves at least two commits. Sampling is stratified by repository and the number of commits produced by RUCANGLE, so that the sample approximates the composition of the eligible candidates. All 150 selected patches are included in the reported 453-patch repair dataset.

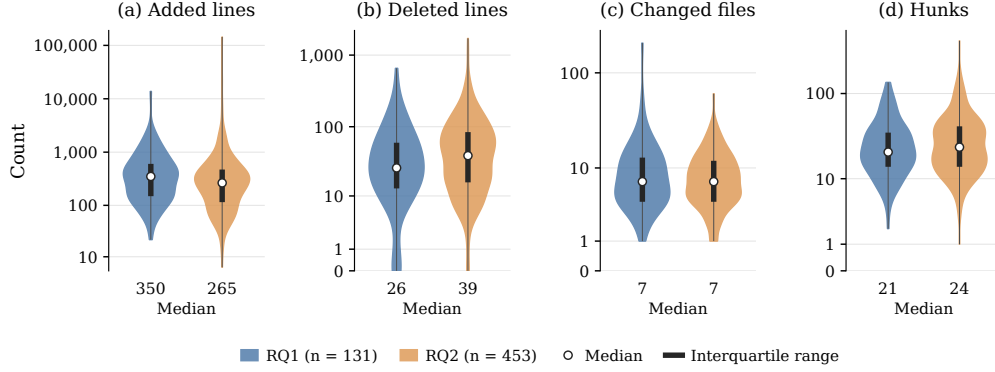


Figure 3. Distributions of modified files, diff hunks, added lines, and deleted lines in the RQ1 and RQ2 datasets. The median numbers of added and deleted lines are 350 and 26 for RQ1, and 265 and 39 for RQ2.

4.2 Compared Methods and Repair Conditions

4.2.1 Commit Untangling Methods

We compare five commit untangling methods. FileSplit and HunkSplit are simple structural baselines that partition a patch by file and by diff hunk, respectively. Armchr [1] combines program-structure and dependency analysis with LLM-assisted untangling. Atomizer [42] constructs dependency-aware minimal change subgraphs, infers change intent, and uses multiple LLM agents to iteratively refine groups by intent. RUCANGLE uses an untangling agent to inspect the repository and propose an ordered commit plan, then uses runnability checks to guide revisions to the grouping and order (Section 3.1).

To anchor the scale of these metrics, we additionally report NoUNTANGLE, a reference condition that keeps the full development patch as a single commit in the untangling results evaluation.

FileSplit, HunkSplit, and Atomizer produce unordered groups. We order these groups according to their dependencies. When no dependency determines the next group, we select the first group that can be applied to the current repository version.

We include Atomizer, a state-of-the-art commit untangling approach that outperforms prior methods on synthetic C# and Java benchmarks [42]. We use a Python adaptation of Atomizer’s change-graph pipeline. Our implementation uses normalized BAAI/bge-base-en-v1.5 embeddings [39] and merges groups whose cosine similarity exceeds 0.6.

4.2.2 Regression Repair Conditions

We evaluate whether providing coding agents with untangled histories helps them diagnose and repair regressions. The main comparison includes repair without untangled commits and repair with commits generated by other baseline untangling methods: Armchr [1] and Atomizer [42]. FileSplit and HunkSplit are excluded because they neither group changes by development purpose nor produce commit messages.

The regression repair evaluation compares four conditions: NoHistory, ARMCHR, ATOMIZER, and RUCANGLE. NoHistory provides no untangled commits. For the other three, TANGLEVAL provides the untangling result of each method as the context to support the repair agent.

4.3 Implementation

4.3.1 Agents and Execution Environment

The untangling agent of RUCANGLE and the repair harness are built on mini-swe-agent [33] (version 2.4.6). RUCANGLE’s untangling agent, Armchr, and Atomizer all use DeepSeek-V4-Flash through the OpenRouter

API. The two main models in regression repair evaluation, Qwen3-Coder-Next [3] (FP8) and Nemotron-3.5-Lightning-30B-A3B [25] (BF16), are served locally with vLLM [16] on four and two NVIDIA L40S 48 GB GPUs, respectively, with a context window of 262,144 tokens. We apply untangled commits, check runnability, and run `git bisect` and tests in the task’s SWE-CI container image, using a separate container for each version produced by an untangled commit or each repair attempt.

Each RUCTANGLE untangling session and each repair attempt is limited to 80 agent steps. RUCTANGLE uses one initial planning session and up to two revision sessions when validation fails. The repair agent may inspect and modify the repository, execute tests, and query the supplied history through Git operations such as `git log`, `git show`, `git diff`, and `git blame`. Operations that would change the checked-out revision, such as `git checkout`, are blocked. This keeps the base revision fixed at V_{fail} for every attempt. All agents use temperature 0 and top-p 1. Because repair outputs can still vary across runs on the shared serving backend, the main experiment uses three repair attempts per patch and condition.

4.3.2 Additional Models

To assess whether the observed benefit extends beyond the two main models, we evaluate MiMo-V2.6-Flash, Qwen3.7-Flash, DeepSeek-V4-Flash, and GPT-6-Luna under NoHISTORY and RUCTANGLE, with one attempt per patch and condition on the shared 150-patch sample. We access these models through the OpenRouter API and run GPT-6-Luna with reasoning effort set to none. All other settings follow the main experiment.

4.4 Trajectory Analysis

Untangled histories present code changes together with commit messages describing their intended purposes. These records can help coding agents identify changes relevant to a regression and determine which behavior a repair needs to preserve. Understanding how agents interpret and act on this information can inform the design of agents that actively use software history in subsequent development. We therefore investigate how coding agents interact with untangled histories.

We conduct case studies of trajectories from the regression-repair evaluation. We select patches for which the NoHISTORY and RUCTANGLE conditions produce different repair outcomes for the same model and attempt index. For each selected patch, we compare the recorded trajectories, tracing code changes the agent inspects, how it interprets commit messages in relation to code behavior, and its subsequent repair decisions and outcomes. When examining agents’ use of commit messages, we also compare repeated attempts for the same patch, model, and condition to understand how agents interpret the same messages differently.

5 Results Analysis

5.1 RQ1. How well do different untangling methods split AI-generated patches?

We evaluate methods on 131 agent-generated patches. Table 1 reports whether their outputs reconstruct the original patches, remain runnable after each commit, and introduce temporary test failures. It also describes the number of commits and the distribution of test gains. NoUNTANGLE keeps each complete patch in a single commit and serves as a reference for interpreting results.

We observe that *reconstructing the original patch does not ensure that the code remains runnable after each commit*. All five methods reconstruct at least 96.2% of the input patches. However, only 62.6%–79.4% of the reconstructed histories from the four baselines pass the runnability checks after every commit. For example, FILESPLIT and HUNKSPLIT reconstruct all 131 patches, but their proportions of fully runnable histories are 72.5% and 62.6%, respectively. RUCTANGLE meets both requirements on all 131 patches. These results show that preserving the final code does not by itself make the generated history suitable for execution at each commit.

RUCTANGLE achieves full runnability while splitting nearly all patches into multiple commits. NoUNTANGLE achieves full runnability and no intermediate test failures by retaining each patch as one commit. This reference shows why execution results must be interpreted alongside the amount of splitting. RUCTANGLE produces a median of eight commits per patch, and only 0.8% of its histories contain a single commit. Armchr

Table 1. Untangling results on 131 patches. Metric definitions are given in Section 3.2.1. Bold marks the best values among the five untangling methods, except for the commit count metrics. Lower values are better for mean Intermediate test failures (ITF) and MTP. NoUNTANGLE is a reference condition.

Method	Patch reconstruction	Commit count			Runnability		ITF		Test gains	
		Mean k	Med. k	Single commit (%)	Mean rate	All versions (%)	Mean rate	No failures (%)	MTP	TPE
NoUNTANGLE	131 (100%)	1.00	1	100.0	1.000	100.0	0.000	100.0	1.000	0.000
FileSplit	131 (100%)	11.18	7	2.3	0.866	72.5	0.209	57.7	0.693	0.400
HunkSplit	131 (100%)	29.71	21	0.0	0.830	62.6	0.324	30.2	0.610	0.361
Armchr	126 (96.2%)	6.54	5	7.1	0.898	79.4	0.164	59.2	0.739	0.394
Atomizer	126 (96.2%)	7.52	6	3.2	0.912	73.8	0.207	51.2	0.699	0.419
RUCANGLE	131 (100%)	8.64	8	0.8	1.000	100.0	0.017	82.3	0.552	0.562

and Atomizer produce medians of five and six commits, with single-commit histories accounting for 7.1% and 3.2% of their outputs, respectively. FILESPLIT creates one commit per changed file, while HUNKSPLIT creates one per diff hunk. They produce more commits on average than RUCANGLE: 11.18 and 29.71, respectively, compared with 8.64. However, they have the lowest proportions of histories runnable after every commit.

A smaller proportion of tests temporarily fail in histories generated by RUCANGLE. Among the tests that pass in the final version, we measure the fraction that change from passing to failing after at least one commit. Averaged across the evaluated histories, this fraction is 1.7% for RUCANGLE, compared with 16.4%–32.4% for the four baselines. Thus, fewer such tests temporarily fail in RUCANGLE’s histories. This result is observed even though RUCANGLE does not use the tests’ pass/fail results to revise the commit grouping or order.

Test gains are less concentrated in histories generated by RUCANGLE, as measured by the metrics in Section 3.2.1. Its mean MTP is 0.552, compared with 0.610–0.739 for the baselines, and its mean TPE is 0.562, compared with 0.361–0.419. The comparison with HUNKSPLIT shows why commit counts alone do not describe functional decomposition: HUNKSPLIT produces more than three times as many commits on average, but has a higher mean MTP of 0.610. Finer structural splitting therefore does not necessarily reduce the concentration of test gains. These observations suggest that, compared to other untangling methods, RUCANGLE can distribute a tangled patch’s functional gains more evenly across commits, indicating that RUCANGLE’s histories are more likely to provide commits that serve single development purposes.

Regarding the cost, RUCANGLE has the lowest average untangling time among the three LLM-based methods. Across all 131 patches, including unsuccessful attempts, RUCANGLE takes 276.3 seconds per patch on average, 74.6% of Armchr’s time and 18.0% of Atomizer’s. It generates an average of 11.9K output tokens per patch, close to Armchr’s 11.7K and only 25.9% of Atomizer’s 46.0K. These measurements include constructing, checking, and revising the history, but exclude the subsequent bisect step and regression repair.

Answer to RQ1: RUCANGLE is the only tested untangling method to reconstruct all 131 patches while keeping the code runnable after every commit, with a median of eight commits per patch. It also temporarily breaks a smaller proportion of tests: among tests that pass in the final version, an average of 1.7% change from passing to failing during the commit sequence, compared with 16.4%–32.4% for the baselines. Its histories show less concentrated test gains across commits than baseline histories.

5.2 RQ2. How does commit untangling affect regression repair by coding agents?

For each patch, repair agents start with the same failing code, task description and failure information, and the same repair budget. Under NoHistory, they receive no untangled commits. We provide the information generated by RUCANGLE, ARMCHR and ATOMIZER to the agent context. In total, we evaluate six models. Qwen3-Coder-Next and Nemotron are evaluated on 453 patches, with three attempts per patch under each condition. MiMo-V2.6-Flash, Qwen3.7-Flash, DeepSeek-V4-Flash, and GPT-6-Luna are evaluated under

Table 2. Repair results for Qwen3-Coder-Next and Nemotron on 453 patches, with three attempts per patch and condition. The baseline ARMCHR and ATOMIZER conditions provide commits from corresponding untangling methods. The RUCANGLE condition provides its commits together with bisect results. When the method fails, we reuse the corresponding NoHistory outcomes. All values are percentages, with metric definitions given in Section 3.2.2. Bold and underlined values mark column maxima within each model.

Model	Condition	pass@1	pass@3	Partial repair	Full repair	Patch generation
Qwen3-Coder-Next	NoHistory	26.9%	40.4%	43.9%	32.8%	56.0%
	ARMCHR	28.8%	40.4%	47.2%	36.1%	58.6%
	ATOMIZER	28.8%	40.4%	47.6%	35.7%	57.6%
	RUCANGLE	<u>31.3%</u>	<u>43.5%</u>	<u>51.1%</u>	<u>39.7%</u>	<u>63.0%</u>
Nemotron	NoHistory	30.8%	41.1%	61.0%	41.2%	80.4%
	ARMCHR	31.3%	43.7%	62.5%	42.2%	83.8%
	ATOMIZER	32.3%	44.2%	62.5%	42.9%	82.4%
	RUCANGLE	<u>36.0%</u>	<u>47.7%</u>	<u>69.5%</u>	<u>49.4%</u>	<u>88.3%</u>

NoHistory and RUCANGLE on the shared 150-patch sample (Section 4.1.2), with one attempt per patch due to budget limits.

A successful repair makes all target tests pass without breaking other evaluation tests that passed before repair. Pass@1 measures the proportion of successful attempts. For the models with three attempts per patch, pass@3 measures the proportion of patches successfully repaired in at least one attempt. Partial and full repair indicate whether at least one or all target tests pass, respectively; both can include new test failures.

Our experiment shows that *RUCANGLE improves repair success for four of the six evaluated models*. Compared with NoHistory, RUCANGLE raises Qwen3-Coder-Next’s pass@1 from 26.9% to 31.3% and Nemotron’s from 30.8% to 36.0%. Armchr and Atomizer also help these two models repair more patches, but achieve smaller improvements than RUCANGLE. For MiMo-V2.6-Flash and Qwen3.7-Flash, RUCANGLE raises pass@1 from 29.3% to 36.0% and from 50.0% to 53.3%, respectively. However, RUCANGLE does not improve DeepSeek-V4-Flash’s pass@1, which remains at 61.3%. GPT-6-Luna has a lower pass@1 with RUCANGLE, 48.7% against 50.0% under NoHistory, corresponding to 73 successful repairs instead of 75 on the 150-patch sample.

For Qwen3-Coder-Next and Nemotron, RUCANGLE also achieves the highest pass@3. Compared with NoHistory, the rates increase from 40.4% to 43.5% and from 41.1% to 47.7%, respectively. Thus, more patches receive at least one successful repair within three attempts. For Qwen3-Coder-Next, Armchr and Atomizer increase pass@1 but leave pass@3 unchanged at 40.4%.

Additionally, we observe that *agents can repair the target regressions while introducing new ones*. With RUCANGLE, all six models more often fix the regressions targeted by the repair task. However, five models also more often cause previously passing tests to fail, indicating that their repairs introduce new regressions. On the 150-patch sample, RUCANGLE raises GPT-6-Luna’s full repair rate from 70.7% to 79.3%. However, its rate of introducing new regressions also increases from 34.0% to 43.3%. Its pass@1 falls from 50.0% to 48.7%, since this metric also requires previously passing tests to remain passing. For DeepSeek, RUCANGLE similarly raises the full repair rate but leaves pass@1 unchanged. For Qwen3-Coder-Next and Nemotron, RUCANGLE improves pass@1, although their rates of introducing new regressions also increase.

Patch generation also increases for all six models. This indicates that more attempts leave code changes, including changes left when an agent reaches its step limit. The repair metrics establish whether those changes fix the original failures and preserve previously passing tests. These results show why checking only the original failures would overstate repair success. Repairs made with untangled histories should also be checked against tests that passed before the repair. Section 5.3 examines individual trajectories to understand how agents use the supplied information when choosing and implementing repairs.

Table 3. Repair outcomes of MiMo-V2.6-Flash, Qwen3.7-Flash, DeepSeek-V4-Flash, and GPT-6-Luna on the shared 150-patch sample, with one attempt per patch and condition. All outcome columns are percentages with denominator 150. Partial repair counts attempts that fix at least one test failure, including full repairs. Bold and underlined values mark column maxima within each model.

Model	Condition	pass@1	Partial repair	Full repair	Patch generation
MiMo-V2.6-Flash	NoHistory	29.3%	38.0%	35.3%	42.0%
	RUCtANGLE	<u>36.0%</u>	<u>45.3%</u>	<u>41.3%</u>	<u>51.3%</u>
Qwen3.7-Flash	NoHistory	50.0%	81.3%	62.7%	88.7%
	RUCtANGLE	<u>53.3%</u>	<u>88.7%</u>	<u>73.3%</u>	<u>95.3%</u>
DeepSeek-V4-Flash	NoHistory	<u>61.3%</u>	86.0%	72.0%	91.3%
	RUCtANGLE	<u>61.3%</u>	<u>92.0%</u>	<u>79.3%</u>	<u>93.3%</u>
GPT-6-Luna	NoHistory	<u>50.0%</u>	87.3%	70.7%	96.0%
	RUCtANGLE	48.7%	<u>94.7%</u>	<u>79.3%</u>	<u>98.0%</u>

Answer to RQ2: RUCtANGLE improves pass@1 for four of the six evaluated models, leaves one unchanged, and yields a slightly lower rate for one. It also achieves the highest pass@3 for both models evaluated with repeated attempts. Across all six models, the original failures are fixed more often, but new test failures also become more frequent for most models, limiting the improvement in overall repair success.

5.3 RQ3. How do coding agents utilize untangled histories?

We examine some repair trajectories from RQ2 to understand how agents use untangled histories. The following three observations describe how agents inspect changes, interpret commit messages, and decide what to repair. We illustrate each observation with concrete attempts on the same patch.

Observation 1: Agents use history to check what changed before deciding what to repair. Several parts of the code may affect the same failing behavior. Commit diffs and earlier versions let an agent check which parts were changed by the patch and which were already present.

In one task from our dataset, a coding agent modified the `poethepoet` project. It mistakenly deleted code that adds the required line breaks at the end of the program’s command-line help text. The resulting help text no longer matched the expected format, causing eight previously passing tests to fail. In its first NoHistory attempt, Nemotron removes the helper’s `rstrip()` and another trimming call without restoring the deleted newline handling. The resulting patch fixes none of the eight target test failures. In its first RUCtANGLE attempt, Nemotron inspects the candidate commit supplied by RUCtANGLE’s `git bisect` step and restores the deleted code. The third attempt shows another use of history: the agent also suspects `rstrip()`, but checks an earlier version and confirms that the call was already present. It leaves the helper unchanged and restores the newline handling. All three RUCtANGLE attempts fix all eight target failures without introducing additional test failures, whereas none of the three NoHistory attempts achieves a full repair.

Observation 2: Agents consult commit messages to check which behavior a repair should preserve. A change introducing errors may also add behavior that should be retained. By reading its commit message, agents check why the change was made and decide what to preserve while repairing.

In another example, a coding agent introduced an error while modifying XML parsing in the `pydantic-xml` project. It added a `consume()` call to mark an XML element as processed, preventing the parser from reporting it again as unexpected input. However, this call also clears the element’s contents. The agent placed it before the code that reads those contents for the result, causing a previously passing test to fail. In one NoHistory repair attempt, Qwen recognizes the call’s purpose from the code documentation but ultimately removes it. The original failing test then passes, but three previously passing tests fail because processed elements are now reported as unexpected input. In one RUCtANGLE repair attempt, Qwen consults the commit message to check why `consume()` was added. The message explains that processed elements should no longer be treated

as unhandled input. The agent then examines the code that checks for unexpected input and retains the call. It changes the order so that the parser reads and saves the contents before calling `consume()`. This preserves the returned data while still marking the element as processed. The resulting patch fixes the original test failure without causing additional test failures.

Observation 3: Agents may be misled by commit messages. A generated commit message may incorrectly describe a faulty change as an improvement. An agent may then question a correct repair because it conflicts with the explanation in the message.

In the `poethepoet` case described above, `RUCANGLE` locates the commit that deleted the required newline handling. However, the generated message calls these line breaks “redundant” because “callers already handle line breaks.” The diff shows the deletion responsible for the failing output, but the message gives a reason to keep that deletion. In another `RUCANGLE` case, `Qwen` identifies the deleted code and repeatedly proposes restoring it. However, it keeps returning to the message and questions whether the tests should change instead. Along with repeatedly checking the number of line breaks, it keeps revisiting this uncertainty until reaching the step limit without changing the code. In the second attempt, `Qwen` receives the same commit and message and raises similar doubts, but eventually restores the deleted code and tests the repair, fixing all target test failures without introducing any additional failures.

This observation motivates further study of how commit messages generated during untangling affect agents’ repair decisions. Future work could adapt commit message generation methods [7, 18] to describe and explain untangled changes more accurately, helping agents understand development purposes [35, 17].

Answer to RQ3: Agents use commit diffs and earlier code to check which changes need repair. They consult commit messages to understand why a change was made and decide what behavior to preserve. However, misleading messages may lead agents to question a correct repair, even after the faulty change has been located.

6 Discussion

6.1 How Do Coarser Histories Affect Repair?

`RUCANGLE` does not prescribe an exact number of untangled commits. The agent chooses how to group the changes, subject to a limit of 20 commits, and subsequent revisions may change this grouping. Separating dependent changes can leave the code unable to run before all required changes have been applied. Nevertheless, in RQ1, `RUCANGLE` keeps the code runnable after every commit while producing a median of 8 commits per patch, compared with 5 for `Armchr` and 6 for `Atomizer`. We now examine whether this finer-grained decomposition contributes to regression repair, or whether commits can be merged without reducing repair success.

We use the shared sample of 150 patches described in Section 4.1.2. Each selected patch has a `RUCANGLE` history containing at least four commits. We compare two settings: (1) `RUCANGLE-ORIGINAL`, which retains the history generated by `RUCANGLE`, and (2) `RUCANGLE-HALVED`, which reduces a history of k commits to $\lceil k/2 \rceil$ commits. To construct `RUCANGLE-HALVED`, we randomly retain $\lceil k/2 \rceil - 1$ of the $k - 1$ boundaries between adjacent commits and merge across the remaining boundaries. This produces consecutive groups that preserve the original order but need not contain the same number of commits. The commit messages are concatenated if the commits are merged.

Merging removes intermediate versions but does not modify the code at the retained versions. The initial and final versions therefore remain unchanged, and every retained intermediate version is drawn from the original runnable history. Consequently, `RUCANGLE-HALVED` remains runnable after every commit. We rerun the bisect step on the merged history and provide the resulting candidate commit and history to the repair agent. All other repair settings follow the `RUCANGLE` condition in RQ2. To limit evaluation cost, we evaluate only `Nemotron`, using three repair attempts per patch and a limit of 80 agent steps per attempt. Table 4 compares the two conditions.

Table 4. Nemotron repair outcomes with original and halved histories on 150 patches, with three attempts per patch. Both settings provide the history and its bisect result. Bold and underlined values mark column maxima, except for new regression, where the minimum is marked.

	History	pass@1	Partial repair	Full repair	New regression	Patch generation
RUCANGLE-ORIGINAL		<u>34.9%</u>	<u>65.1%</u>	<u>46.9%</u>	<u>25.6%</u>	83.3%
RUCANGLE-HALVED		31.8%	64.7%	45.1%	28.4%	<u>86.4%</u>

We observe that pass@1 decreases from 34.9% with RUCANGLE-ORIGINAL to 31.8% with RUCANGLE-HALVED. To better understand this difference, we examine the other repair outcomes. Partial repair counts attempts that fix at least one target failure, while full repair requires all target failures to be fixed. The partial repair rate changes little, from 65.1% to 64.7%, but the full repair rate decreases from 46.9% to 45.1%. New regressions occur when a repair breaks tests that previously passed; their rate increases from 25.6% to 28.4%. Thus, after merging, fewer attempts fix all target failures and more introduce failures elsewhere.

We also examine patch generation, which counts attempts that leave a nonempty diff, including edits left by attempts that reach their limit. This rate increases from 83.3% to 86.4%. The agent therefore leaves changes more often with RUCANGLE-HALVED, but fewer attempts satisfy the complete repair requirements. These observations suggest that retaining the original histories may help the agent complete repairs while preserving existing behavior. However, merging also changes the scope of the candidate commit identified by bisect and the messages presented together. The results reflect this combined change in the information available to the agent, rather than the effect of commit count alone.

6.2 Implications and Research Opportunities

Adopting established software engineering practices. Our results illustrate the continued value of established software engineering practices in agentic coding. Organizing changes into runnable histories and providing bisect results improves repair for several models without retraining them. Recent training pipelines, such as SWE-Gym and SWE-smith, use test outcomes to identify successful coding trajectories [26, 40]. Passing these tests alone does not establish whether the resulting changes are easy to inspect and maintain; their usability must also be considered [41]. Agent training and development workflows should therefore include preparing changes for later review and maintenance, alongside completing the current coding task.

Training agents to use project history. Providing history does not ensure that an agent will use it effectively. This motivates research on *history-aware coding agents*: agents trained to identify relevant commits, compare earlier and current code, and check historical explanations against code and tests. Training tasks could include both useful and misleading historical information, allowing researchers to examine whether agents learn when to rely on it. Evaluation should measure whether this training improves repair decisions while preserving previously working behavior.

Designing project histories for subsequent agents. Alongside commits and review discussions, recorded agent runs can preserve which files were inspected, which edits were attempted, what tests returned, and how the agent explained its decisions. These records could help later agents understand a change, but repeated exploration, abandoned approaches, and mistaken explanations may obscure relevant information. Future systems could link selected actions and test results to commits and let agents retrieve further details as needed to assess changes without reconstructing the entire development process. Recorded explanations should also remain distinguishable from observed edits and test outcomes.

Evaluating commit messages through subsequent decisions. We did not independently control message quality in our experiments, leaving its effect on repair as an open research question. Beyond accuracy and readability, evaluation could examine whether a message helps an agent identify relevant changes, preserve required behavior, and avoid introducing new regressions.

6.3 Threats to Validity

Threats to Internal Validity. Our comparison with Atomizer depends on its Python adaptation, including the embedding model, grouping threshold, and commit ordering. We document these choices, use the same LLM across the three LLM-based untangling methods, and also compare with Armchr’s independent implementation. We use the project containers and select tests with consistent outcomes across three executions of candidate snapshots. The main repair experiment uses three attempts per patch and condition, with the same failing code, tests, and repair budget across conditions. The additional-model experiment uses one attempt per patch and provides less evidence about variation across runs. Our repair comparisons evaluate the complete RUC_{TANGLE} system, including bisect localization, without isolating component contributions.

Threats to External Validity. Our experiments cover multiple repositories and models, but are limited to Python projects from SWE-CI. The additional-model results show that the repair benefit varies across models, and the merging analysis concerns only histories with at least four commits and available localization results. We focus on agent-based regression repair because coding agents increasingly participate in maintenance, and tests allow us to compare repair outcomes automatically. Whether untangled histories also help agents perform code review and refactoring, or help human developers review and integrate changes, remains to be evaluated. The checking stage’s effectiveness for projects in other languages or with different deployment requirements remains to be evaluated.

7 Related Work

7.1 Commit Untangling

Commit untangling separates changes that serve different development purposes. Early studies show that tangled changes obscure the relationship between code changes and bug reports and affect defect prediction [11, 12]. Methods therefore seek to identify which changes belong together. EpiceaUntangler [6] groups recorded edits using time gaps, intervening edit counts, and whether they modify the same class. Flexeme [27] groups changes using code dependencies and the identifiers involved in those dependencies. UTango [19] learns to group changes using their dependencies and surrounding code. Recent methods use LLMs to interpret the purposes of changes. CoLaUntangle [14] combines agents that examine code dependencies and conceptual relationships, then review and revise their proposed groups. Atomizer [42] asks agents to infer each change’s purpose and check whether grouped changes serve the same purpose.

Dependencies also matter when groups of changes become separate commits. SmartCommit [30] constrains grouping to avoid compilation errors and orders commits so that required changes appear before their dependents. Armchr [1], which we include in our experiments, also groups changes and orders the resulting patches using their dependencies. Earlier tools address related tasks: Crisp [29] constructs compilable versions containing selected changes to help developers locate failures, while ChgCutter [10] helps developers select related changes and the tests affected by them. Safe-commit [38] identifies subsets of workspace changes that can be committed without causing additional failures of existing tests. These tools establish the importance of dependencies, but still ignore the runnability requirement that this paper emphasizes.

RUC_{TANGLE} constructs an ordered history that preserves every change in the input patch. It applies the proposed commits and checks whether the tests can be collected after each commit. Our experiments compare this approach with Armchr and a Python adaptation of Atomizer, alongside file-based and hunk-based splitting. Our experiments exclude other works [27, 19, 30] designed for specific programming languages.

7.2 Evaluating Untangled Commits

A common evaluation combines several developer-written commits into one patch and asks an untangling method to recover the original groups [27, 19]. Atomizer [42] also compares its output with expert annotations of real commits. These evaluations measure agreement with a reference grouping, which need not be the only reasonable way to split a patch. Developer studies provide another form of evidence: Barnett et al. [2] investigate whether developers find automatically separated changes useful for understanding code reviews.

Recent benchmarks evaluate histories produced by coding agents. GitGoodBench [21] includes a task in which agents organize uncommitted changes into commits and an LLM assesses the resulting history. Its iterative committing setup restricts the agent to changes within one file. AtomicCommitBench [20] evaluates histories reconstructed from squashed developer commits. It measures agreement with the original groups and checks whether the generated commits can be applied in order. It also separately reverses each generated commit’s changes from the final code and reruns selected tests to examine whether the resulting failures are confined to particular commits.

TANGLEVAL examines the versions produced as commits are applied in order. For agent-generated patches without a reference grouping, it checks whether the history reconstructs the original patch, whether the code remains runnable after each commit, and whether tests temporarily fail along the history. It also evaluates the use of these histories in regression repair. This extends the evaluation from how commits are grouped and how their changes affect tests to whether coding agents can use the generated histories to repair errors.

8 Conclusion

We presented RUCANGLE, an agentic method that uses execution feedback to regroup changes and reorder commits while preserving the final code of the original patch. On 131 agent-generated patches, RUCANGLE produces histories that pass the runnability checks after every commit, with a median of eight commits per patch. We also introduced TANGLEVAL to evaluate how untangled histories help coding agents repair regressions. On 453 patches that introduce regressions, providing RUCANGLE’s histories together with git bisect results raises pass@1 by 16.6% on average across two models, relative to repair without untangled commits. Our case studies show how agents use commit diffs to identify relevant changes and consult commit messages to preserve required behavior. They also reveal an attempt in which an agent repeatedly questions a correct repair after reading a misleading message and exhausts its budget without producing a patch.

These results demonstrate how an established software engineering practice can support coding agents in maintaining code. They motivate further research on how agents can actively use software history, while checking its explanations against code and tests, to guide development decisions.

References

- [1] Armchr. Armchr: Code splitter agent. GitHub repository, 2026. <https://github.com/armchr/armchr/tree/main/code-splitter-agent>. Evaluated upstream revision 11da42e34f66; accessed 2026-10-02.
- [2] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 37th International Conference on Software Engineering*, volume 1 of *ICSE ’15*, pages 134–144. IEEE, 2015. doi: 10.1109/ICSE.2015.35.
- [3] Ruisheng Cao, Mouxian Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiayi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. Qwen3-Coder-Next technical report, 2026. <https://arxiv.org/abs/2603.00729>.
- [4] Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkeley, CA, 2nd edition, 2014. doi: 10.1007/978-1-4842-0076-6.
- [5] Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. SWE-CI: Evaluating agent capabilities in maintaining codebases via continuous integration, 2026. <https://arxiv.org/abs/2603.03823>.
- [6] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER ’15, pages 341–350. IEEE, 2015. doi: 10.1109/SANER.2015.7081844.
- [7] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. FIRA: Fine-grained graph-based code change representation for automated commit message generation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE ’22, pages 970–981. ACM, 2022. doi: 10.1145/3510003.3510069.
- [8] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2 edition, 2018. ISBN 9780134757599.

- [9] GitHub. About protected branches. GitHub Docs. <https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-protected-branches/about-protected-branches>. Accessed 2026-10-01.
- [10] Bo Guo, Young-Woo Kwon, and Myoungkyu Song. Decomposing composite changes for code review and regression test selection in evolving software. *Journal of Computer Science and Technology*, 34(2):416–436, 2019. doi: 10.1007/s11390-019-1917-9.
- [11] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR ’13, pages 121–130. IEEE, 2013. doi: 10.1109/MSR.2013.6624018.
- [12] Kim Herzig, Sascha Just, and Andreas Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21(2):303–336, 2016. doi: 10.1007/s10664-015-9376-6.
- [13] Kosei Horikawa, Hao Li, Yutaro Kashiwa, Bram Adams, Hajimu Iida, and Ahmed E. Hassan. Agentic refactoring: An empirical study of AI coding agents, 2025. <https://arxiv.org/abs/2511.04824>.
- [14] Bo Hou, Xin Tan, Kai Zheng, Fang Liu, Yinghao Zhu, and Li Zhang. LLM-driven collaborative model for untangling commits via explicit and implicit dependency reasoning. *ACM Transactions on Software Engineering and Methodology*, 2026. doi: 10.1145/3822177.
- [15] 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? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024. https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.
- [16] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP ’23, pages 611–626. ACM, 2023. doi: 10.1145/3600006.3613165.
- [17] Jiawei Li and Iftekhar Ahmed. Commit message matters: Investigating impact and evolution of commit message quality. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 806–817. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00076.
- [18] Jiawei Li, David Faragó, Christian Petrov, and Iftekhar Ahmed. Only diff is not enough: Generating commit messages leveraging reasoning and action of large language model. *Proceedings of the ACM on Software Engineering*, 1(FSE): 745–766, 2024. doi: 10.1145/3643760.
- [19] Yi Li, Shaohua Wang, and Tien N. Nguyen. UTANGO: Untangling commits with context-aware, graph-based, code change clustering learning model. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE ’22, pages 221–232. ACM, 2022. doi: 10.1145/354025.03549171.
- [20] Zhihao Lin, Mingyi Zhou, and Li Li. AtomicCommitBench: Can coding agents reconstruct commit histories from squashed patches?, 2026. <https://arxiv.org/abs/2607.03332>.
- [21] Tobias Lindenbauer, Egor Bogomolov, and Yaroslav Zharov. GitGoodBench: A novel benchmark for evaluating agentic performance on Git. In *Proceedings of the 1st Workshop for Research on Agent Language Models*, REALM 2025, pages 272–288. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.realm-1.19.
- [22] Linux Kernel Posting Patches. Posting patches. Linux kernel documentation. <https://www.kernel.org/doc/html/latest/process/5.Posting.html>. Accessed 2026-10-02.
- [23] Linux Kernel Submitting Patches. Submitting patches: The essential guide to getting your code into the kernel. Linux kernel documentation. <https://docs.kernel.org/process/submitting-patches.html>. Accessed 2026-09-29.
- [24] Wenqiang Luo, Jacky Keung, Xiaoyu Shi, Yicheng Sun, Boyang Yang, Zhou Yang, and Haoye Tian. Refine after generation: Toward correct and concise patches in LLM-based program repair, 2026. <https://arxiv.org/abs/2608.13292>.
- [25] NVIDIA Corporation. NVIDIA-Nemotron-3.5-Lightning-30B-A3B-BF16. Model card, 2026. <https://huggingface.co/nvidia/NVIDIA-Nemotron-3.5-Lightning-30B-A3B-BF16>.
- [26] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *Proceedings of the 42nd International Conference*

- on *Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 47717–47737. PMLR, 2025. <https://proceedings.mlr.press/v267/pan25g.html>.
- [27] Profir-Petru Pârțachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. Flexeme: Untangling commits using lexical flows. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '20, pages 63–74. ACM, 2020. doi: 10.1145/3368089.3409693.
 - [28] Sundar Pichai. Sundar Pichai shares news from Google Cloud Next 2026. Google Blog, April 2026. <https://blog.google/innovation-and-ai/infrastructure-and-cloud/google-cloud/cloud-next-2026-sundar-pichai/>. Accessed 2026-06-22.
 - [29] Xiaoxia Ren, Ophelia C. Chesley, and Barbara G. Ryder. Identifying failure causes in Java programs: An application of change impact analysis. *IEEE Transactions on Software Engineering*, 32(9):718–732, 2006. doi: 10.1109/TSE.2006.90.
 - [30] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. SmartCommit: A graph-based interactive assistant for activity-oriented commits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '21, pages 379–390. ACM, 2021. doi: 10.1145/3468264.3468551.
 - [31] Skylenage AI. SWE-CI dataset. Hugging Face dataset, 2026. <https://huggingface.co/datasets/skylenage-ai/SWE-CI>. SWE-CI-full v2.0.0; accessed 2026-09-02.
 - [32] Skylenage AI. SWE-CI trajectory data. Hugging Face dataset, 2026. <https://huggingface.co/datasets/skylenage-ai/SWE-CI-trajectory>. Accessed 2026-09-02.
 - [33] SWE-agent Team. mini-SWE-agent: A minimal software engineering agent. GitHub repository, 2026. <https://github.com/SWE-agent/mini-swe-agent>.
 - [34] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 180–190. IEEE, 2015. doi: 10.1109/MSR.2015.24.
 - [35] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 2389–2401. ACM, 2022. doi: 10.1145/3510003.3510205.
 - [36] Zhao Tian, Zifan Zhang, Tao Xiao, Dong Wang, Masanari Kondo, Junjie Chen, and Yasutaka Kamei. “Refactoring Runaway”: Understanding and mitigating tangled refactorings in coding agents for issue resolution, 2026. <https://arxiv.org/abs/2605.22526>.
 - [37] Yiru Wang and Zhou Yang. Do AI-generated pull requests get rejected more? (yes but why?). In *Proceedings of the 23rd International Conference on Mining Software Repositories*, MSR '26, pages 752–756. ACM, 2026. doi: 10.1145/3793302.3793568. <https://doi.org/10.1145/3793302.3793568>.
 - [38] Jan Wloka, Barbara Ryder, Frank Tip, and Xiaoxia Ren. Safe-commit analysis to facilitate team software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 507–517. IEEE, 2009. doi: 10.1109/ICSE.2009.5070549.
 - [39] Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. C-Pack: Packed resources for general Chinese embeddings. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '24, pages 641–649. ACM, 2024. doi: 10.1145/3626772.3657878.
 - [40] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025. doi: 10.52202/085713-3239. https://proceedings.neurips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abs tract-Datasets_and_Benchmarks_Track.html.
 - [41] Zhou Yang, Zhensu Sun, Terry Yue Zhuo, Premkumar Devanbu, and David Lo. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code, 2024. <https://arxiv.org/abs/2403.07506>.
 - [42] Kangchen Zhu, Zhiliang Tian, Shangwen Wang, Mingyue Leng, and Xiaoguang Mao. Atomizer: An LLM-based collaborative multi-agent framework for intent-driven commit untangling. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, ICSE '26, pages 1419–1431. ACM, 2026. doi: 10.1145/3744916.3787773.