ARTIFACT MEMORY: GUIDING AGENTIC ARTIFACT EDITING WITH EDIT GRAPHS AND CONFLICT RETRIEVAL

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon artifact editing requires agents to manage interactions among components and dependencies across editing decisions. Many edit consequences become apparent only after rendering, yet retaining every intermediate version and observation is costly and does not explain recurring repair failures. We propose *Artifact Memory* to address both the representational and algorithmic needs of artifact editing. Its edit graph shares unchanged components across recoverable versions and links edits to component-level validation outcomes. Relevance-based retrieval selects prior attempts and partial successes, while conflict analysis connects outcomes across versions to identify repairs that undo earlier progress. This context supports coordinated repairs, backtracking, and alternative editing sequences while retaining lessons from abandoned branches. Across HTML/CSS poster editing, SVG editing, and Blender scene editing with Claude Opus 4.8 and GPT-5.6 Terra, Artifact Memory improves constraint-satisfaction success rates over validator-feedback baselines by 10.0–31.8 percentage points. and reduces mean inference cost by 26–39% on tasks solved by both methods. Further trajectory analyses show lower capped repair-cycle recovery costs across domains, more productive backtracking and less repetition of failed continuations.

1 INTRODUCTION

Artifact editing is an important application of generative AI, including poster and slide design (Pang et al., 2025; Zheng et al., 2025), SVG editing (Chen et al., 2025), and 3D scene construction (Yin et al., 2026). Beyond *one-shot synthesis* (Si et al., 2024), these tasks are increasingly approached through *agentic artifact editing*, in which an agent repeatedly inspects, modifies, and renders an evolving artifact (Huang et al., 2024; Yin et al., 2026). This creates a long-horizon decision problem with both spatial and temporal dependencies, whose consequences emerge through rendering and validation. First, *spatial dependencies* couple components within an artifact: resizing an image can push neighboring text outside its container, while moving one object can obscure another. A local edit can therefore affect requirements elsewhere, often in ways that become apparent only after rendering (Pang et al., 2025; Huang et al., 2024). Second, *temporal dependencies* couple decisions across turns: earlier edits determine the conditions under which later edits operate, and a later repair can undo earlier progress. Such cycles can require revisiting an earlier layout decision rather than continuing local repairs. Ideally, memory should link observed effects to the edits and versions that produced them to guide the agent.

Simply accumulating artifact sources, edits, and observations incurs high cost with duplicated content, and provides no mechanism for selecting the information for the agent to resolve the spatial and temporal challenges in artifact editing. Existing memory designs have proposed separation of persistent memory from working context and retrieve relevant experience (Packer et al., 2023; Xu et al., 2025), while context-management methods compress interaction histories (Ye et al., 2025). Persistent data structures and version-control mechanisms preserve recoverable states (Driscoll et al., 1989; Chacon & Straub, 2014; Wu et al., 2025). However, artifact editing requires more than managing history: effective memory design should connect observed outcomes across components and turns to explain recurring repair failures and identify earlier decisions worth revisiting.

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.

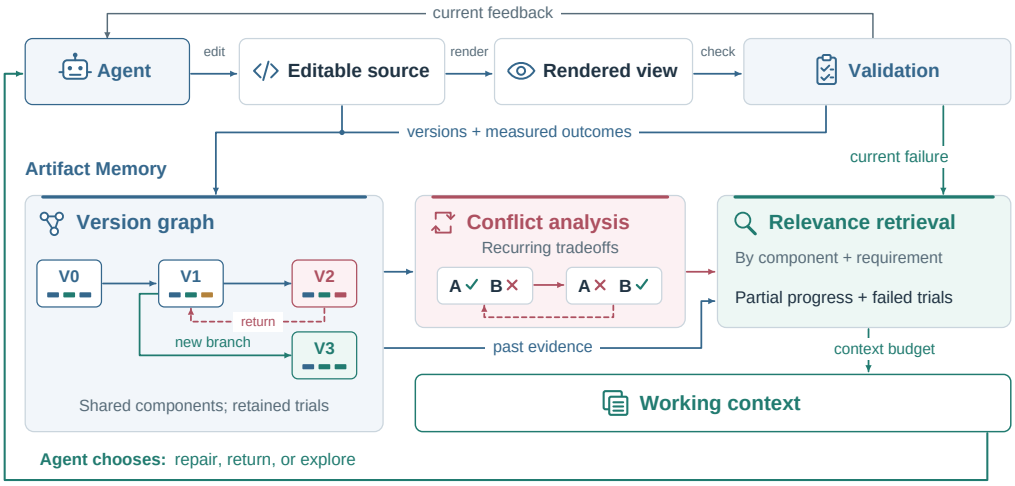


Figure 1: **Artifact Memory overview.** The edit graph shares unchanged components across recoverable versions and links edits to validation outcomes. Bounded retrieval supplies relevant evidence for coordinated repairs and backtracking. The detailed designs are in Section 4.

We introduce *Artifact Memory*, which addresses both representational and algorithmic challenges (Figure 1). *Representationally*, its *edit graph* stores recoverable source versions as nodes and executed edits as edges, sharing unchanged components and recognizing revisits by source identity. It links edits to component-level validation outcomes, recording improvements, ineffective attempts, and regressions, including effects on unmodified components. These records preserve which requirements were satisfied together and the configurations in which outcomes were observed. *Algorithmically*, conflict analysis connects outcomes across versions to identify recurring repairs that undo earlier progress, while relevance-based retrieval selects prior attempts and partial successes relevant to current constraints and components under a fixed memory budget. This evidence helps the agent coordinate repairs, revisit useful configurations, and explore alternative continuations while retaining experience from abandoned branches.

We evaluate Artifact Memory on constrained artifact-editing tasks derived from public benchmarks and scene sources, spanning scientific poster editing with HTML/CSS (Pang et al., 2025), SVG editing (Chen et al., 2025), and Blender scene editing (Gu et al., 2025). Using Claude Opus 4.8 and GPT-5.6 Terra in a controlled API harness, Artifact Memory improves constraint-satisfaction success rates over validator-feedback baselines by 10.0–31.8 percentage points under matched, domain-specific budgets. On tasks solved by both methods, it reduces mean inference cost by 26–39%. Beyond aggregate performance, we analyze recovery from observed repair cycles, productive backtracking, and repetition of failed continuations in complete runs (Section 5.3). These analyses provide evidence that Artifact Memory supports more effective reuse of editing experience when revisiting earlier decisions and exploring alternative continuations, connecting artifact editing with the motivation of conflict-driven backtracking (Marques-Silva & Sakallah, 1999).

We make three contributions to memory design for agentic artifact editing. First, an edit graph links recoverable versions, executed edits, and component-level validation outcomes. Second, relevance-based retrieval selects prior attempts and partial successes within a fixed historical-context budget. Third, conflict analysis connects outcomes across versions to inform coordinated repairs, backtracking, and alternative continuations.

2 Related Work

Agent memory and context management. Agent memory methods extend the information available for each decision by selectively retaining and retrieving past experience. MemGPT manages information across memory tiers (Packer et al., 2023), A-MEM organizes experience into interconnected notes (Xu et al., 2025), and AgentFold compresses interaction histories through context folding (Ye et al., 2025). Persistent data structures preserve recoverable versions through structural

sharing (Driscoll et al., 1989), while version control records source changes (Chacon & Straub, 2014). Git Context Controller applies branching, merging, and historical retrieval to agent context (Wu et al., 2025). Artifact Memory connects these mechanisms through explicit links among recoverable artifact versions, executed edits, and component-level observations of their effects.

Artifact editing vs coding agents. Artifact editing and coding agents (Yang et al., 2024; Wang et al., 2025) both modify source through iterative feedback, but evaluating visual artifacts requires connecting source changes to rendered consequences, and execution alone does not determine layout quality or visual fidelity. Paper2Poster uses rendered feedback to refine posters (Pang et al., 2025), and BlenderAlchemy combines visual evaluation with search and hypothesis reversion (Huang et al., 2024). SVGenius benchmarks understanding, editing, and generation of structured graphics (Chen et al., 2025). More targeted feedback appears in ComUICoder's element-level comparisons (Xiao et al., 2026a) and PosterVisor's requirement contracts and rechecking during repair (Li et al., 2026). These works show the importance validators that make rendered consequences explicit and associated with editable components. We retain these observations in edit graphs and retrieve their interactions so past outcomes remain useful without accumulating all frames.

Agentic search, backtracking, and conflict analysis. Agentic search methods aim to improve decision-making by exploring alternatives and using failure information to revisit earlier choices. Tree of Thoughts explores alternative reasoning paths (Yao et al., 2023), while LATS combines reasoning and acting through explicit tree search (Zhou et al., 2024). AgentOccam supports plan revision through branching and pruning (Yang et al., 2025), and WebRollback enables returns to earlier navigation states (Zhang et al., 2026). Stream of Search learns from trajectories containing unsuccessful paths (Gandhi et al., 2024), while self-backtracking training develops recovery behavior within model policies (Yang et al., 2026b). In constraint solving, dependency-directed backtracking uses failure dependencies to identify decisions worth revisiting (Stallman & Sussman, 1977), and conflict-driven satisfiability solving learns constraints to avoid repeated inconsistent assignments (Marques-Silva & Sakallah, 1999). Artifact Memory draws on these approaches and supplies artifact-specific evidence for repair and backtracking at inference time, enabling agents to revisit prior versions while retaining lessons from failed continuations.

3 Problem Formulation

Artifact editing. We consider *compositional artifacts* with editable source, such as HTML documents, SVG graphics, and Blender scenes. An artifact state is a tuple $x = \langle \mathcal{C}, \mathcal{D}, \{\theta_c\}_{c\in\mathcal{C}} \rangle \in \mathcal{X}$, where $\mathcal{C}$ is a set of identifiable components, $\mathcal{D}$ records structural dependencies such as containment, ordering, and shared style references, and θ_c specifies the editable parameters of component c. Under an environment η specifying assets, fonts, and rendering settings, a rendering engine $E_\eta : \mathcal{X} \to \mathcal{O}$ produces rendered representations $o = E_\eta(x) = (y, z)$, where y is the visual output and z contains render-derived layout and geometric information. An artifact editing agent is given a set of source-edit actions $\mathcal{A}$, such as changing component parameters or content, adding or removing components, and restoring earlier source content. At each turn t, the agent receives context κ_t, such as task instructions, retained interaction history, and auxiliary information, and chooses an action $a_t \in \mathcal{A}$ that transforms artifact x_t to x_{t+1}. Starting from initial artifact x_0, editing generates the trajectory

$$\tau : (x_0, o_0, \kappa_0) \xrightarrow{a_0} (x_1, o_1, \kappa_1) \xrightarrow{a_1} \cdots \xrightarrow{a_{T-1}} (x_T, o_T).$$

where x_T is the terminal artifact ready for judgement.

Task objective and hard constraints. We study instruction-guided editing of artifacts under *hard constraints* in addition to the overall task quality objective. Each task specifies an initial editable source x_0, edit instructions u, and relevant reference material ρ. The agent seeks to produce an artifact that fulfills the instruction with high quality, as assessed by the final judge $J(x_T, y_T; u, \rho)$, while satisfying all constraints in a fixed constraint set $\mathcal{H}$. Each constraint $h_k \in \mathcal{H}$ measures a constraint violation score $h_k(x, z) \geq 0$ from the render-derived information z, with zero indicating satisfaction. We also associate a (token) cost of c_t for each editing step t and set a bound B on the cumulative budget. Thus, the overall optimization problem that the agent aims to solve is:

$$\max_{a_0,\ldots,a_{T-1}} J(x_T, y_T; u, \rho) \quad \text{subject to} \quad h(x_T, z_T) = 0 \ \ \forall h \in \mathcal{H}, \text{ and } \sum_{t=0}^{T-1} c_t \leq B. \tag{1}$$

Here, the separation of task objective and constraint satisfaction is critical. For instance, a Blender scene may aim to maximize visual realism and compositional quality, which are soft objectives with varying degrees of achievement, but it must satisfy hard constraints such as preventing unintended object intersections and keeping required objects visible. These constraints cannot be traded off for a higher quality score, thus requiring the constrained optimization formulation. Throughout editing, although intermediate violations of constraints are allowed in (1), persistently high or non-decreasing scores of $h(x_t, z_t)$ along the editing trajectory τ can signal an unproductive editing path, which can suggest revisiting a previous layout decision.

Validator feedback on constraints. We construct *validator* V that evaluates the artifact properties against the constraints in $\mathcal{H}$ by examining the relevant components based on the render-derived information in z. At turn t, the validator outputs a structured report with one entry per constraint:

$$V(x_t, z_t) = \{\langle k, \mathcal{C}_k(x_t),\ m_{t,k},\ h_k(x_t, z_t)\rangle\}_{h_k \in \mathcal{H}}, \tag{2}$$

where $\mathcal{C}_k(x_t)$ identifies the components involved in constraint h_k and $m_{t,k}$ contains its diagnostic measurements. Each record identifies the requirement being checked, the components it concerns, the measured quantities, and the resulting violation. A zero violation indicates that the check passes. For example, an overflow constraint h_k requires an HTML card's content to fit within the allocated height allocated by its parent container. Suppose the render-derived information z_t shows content height 184 pixels and available height 160 pixels. Then the corresponding validator entry contains:

$$\mathcal{C}_k(x_t) = \{\text{card}, \text{column}\},\ m_{t,k} = (184, 160),\ h_k(x_t, z_t) = \max\{0, 184 - 160\} = 24.$$

Edit consequences and their temporal relationships. Comparing the validator feedback before and after an edit reveals its *edit consequence*: what components are affected and how measured quantities and constraint satisfaction are changed, formulated as $r_t = \langle x_t,\ a_t,\ x_{t+1},\ \mathcal{C}_t^{\text{edit}},\ V_t,\ V_{t+1}\rangle$, where $\mathcal{C}_t^{\text{edit}}$ identifies the edited components, while the validator reports all components involved Edit consequences can have different scopes. A edit can be *locally confined* or exhibit *cross-component effect propagation*, where editing one component changes constraint satisfaction involving other components. For example, moving a box in a Blender scene may reveal target object A while obscuring target object B. The box is the edited component, while the other two target objects are affected components. Such effect propagation makes it difficult to assess an edit solely through its intended outcome. Individual edit consequences reveal further information when considering their *temporal relationships* across artifact versions throughout editing. Considered separately, each edit repairs a currently failing visibility constraint. Considered together, the temporal relation shows that the later repair reverses the earlier gain and reintroduces a previously resolved violation. Earlier artifact versions may also preserve useful combinations of satisfied constraints that are lost in subsequent edits, making them good starting points for another attempt.

Artifact editing thus presents challenges *both* in the effects of edits across components and in the temporal relationships among them. Chronological interaction history contains the underlying edits and observations, but does not explicitly surface how edits affected the relevant components, under which configurations, and whether subsequent attempts preserved or reversed earlier progress.

4 Artifact Memory

Motivated by the challenges of artifact editing above, we propose Artifact Memory, which connects edit consequences across artifact versions and retrieves relevant experience to help the agent decide whether to continue a repair, restore an earlier configuration, or explore a different continuation. Artifact Memory combines three components: an *edit graph* preserves artifact versions and their measured edit consequences; *relevance-based retrieval* selects earlier attempts and partial successes relevant to the current constraints and components; and *conflict analysis* connects related consequences to identify competing repair outcomes and recurring failures.

4.1 Edit Graphs

Versions and edits. The edit graph $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$ has artifact source versions as nodes and executed edits as directed edges. It preserves the initial artifact x_0 and recoverable later versions. Each node retains validator output $V(x_t, z_t)$ and each edge $x_t \xrightarrow{a_t} x_{t+1}$ the edit consequence r_t defined in §3.

Records are indexed by edited components $\mathcal{C}_t^{\text{edit}}$ and constraint participants $\mathcal{C}_k(x_t)$ for relevance-based retrieval. When the agent restores an earlier artifact version, the graph retains the previously attempted edits and their consequences, allowing a different continuation to benefit from those attempts. The graph therefore preserves both the configurations in which constraints were satisfied together and the editing paths explored from them.

4.2 Relevance-based Retrieval and Conflict Analysis

Retrieving relevant edit consequences. Starting from current constraint violations and recent edits, Artifact Memory retrieves related edit consequences from the edit graph. It follows component associations, structural dependencies $\mathcal{D}$, and observed cross-component effect propagation to identify relevant constraints, indexed by Q. This allows a failure involving one component to retrieve edits to another component that affected it. The retrieved records include earlier attempts and partial successes, together with their artifact versions and violation scores $h_k(x_j, z_j)$ for $k \in Q$ under comparable evaluation conditions. An earlier version need not satisfy every constraint to be useful: the constraints satisfied together in that version indicate what restoring it would recover. Remaining violations and current progress that would be lost also accompany the candidate, allowing the agent to assess the benefits and costs of returning.

Measured effects and repair conflicts. For each constraint with comparable validator measurements, the edit consequence r_t provides the change in violation: $\Delta_{t,k} = h_k(x_{t+1}, z_{t+1}) - h_k(x_t, z_t)$. An *improvement* is a fail-to-pass transition or a decrease exceeding a measurement-specific tolerance; a *regression* is a pass-to-fail transition or an increase exceeding the corresponding tolerance. These tolerances account for measurement units and, where applicable, the preceding violation magnitude. We call an edit consequence a *repair conflict* when it improves one constraint while regressing another. The conflict links the edit to its improved and regressed constraints and their participating components $\mathcal{C}_k(x_t)$ and $\mathcal{C}_k(x_{t+1})$. For example, moving a box to reveal a planter may obscure a board, creating a conflict between the two visibility requirements. Attempts that leave targeted violations largely unchanged are also retained as ineffective repairs, even when they do not produce competing improvements and regressions.

Connecting conflicts across versions. A single repair conflict may be resolved by the next edit. Conflict analysis connects related edit consequences to identify when later repairs instead reverse earlier progress. For example, a previously resolved violation recurs when, within an editing continuation, there are intervals $i < j$ such that

$$h_k(x_i, z_i) > 0 \wedge h_k(x_{i+1}, z_{i+1}) = 0 \wedge h_k(x_j, z_j) = 0 \wedge h_k(x_{j+1}, z_{j+1}) > 0.$$

A *repair cycle*, or trap, arises when related repairs repeatedly undo earlier gains or transfer violations among constraints that remain jointly unresolved. In the visibility example, revealing the planter may hide the board, revealing the board may hide the planter again, and another planter repair may hide the plate. Connecting these consequences reveals the repeated failures when the artifact need not revisit an identical configuration. Analysis retains the supporting edits and relevant constraints, whose satisfaction must be considered together when assessing the conflict. Deliberately restoring an earlier artifact is distinguished from a repair that reintroduces a violation.

Learning from abandoned continuations. When the agent restores an earlier version, the abandoned continuation and its measured consequences remain in the edit graph. These records provide contextual *nogood* information: a particular continuation was unsuccessful under the recorded conditions. Their relevance depends on the components, constraints, and configuration involved; repeating an edit family alone does not establish repetition of the same failure. Together with retrieved partial successes, this information supports a different continuation from the restored version.

Memory updates and agent decisions. Let M_t denote the edit graph and its associated records at turn t, initialized with the initial artifact x_0 and validator report V_0. After an evaluated edit, memory is updated and relevant information is retrieved:

$$M_{t+1} = \text{Update}(M_t, r_t), \text{ and } D_{t+1} = \text{Retrieve}(M_{t+1}, x_{t+1}, V_{t+1}).$$

The update records the new edit consequence and connects it to earlier attempts through conflict analysis. Retrieval uses the current artifact and constraint violations to select relevant consequences, partial successes, and associated analysis. The resulting memory context D_{t+1} is supplied as auxiliary information in the agent's context κ_{t+1}. It helps the agent assess whether to continue the current

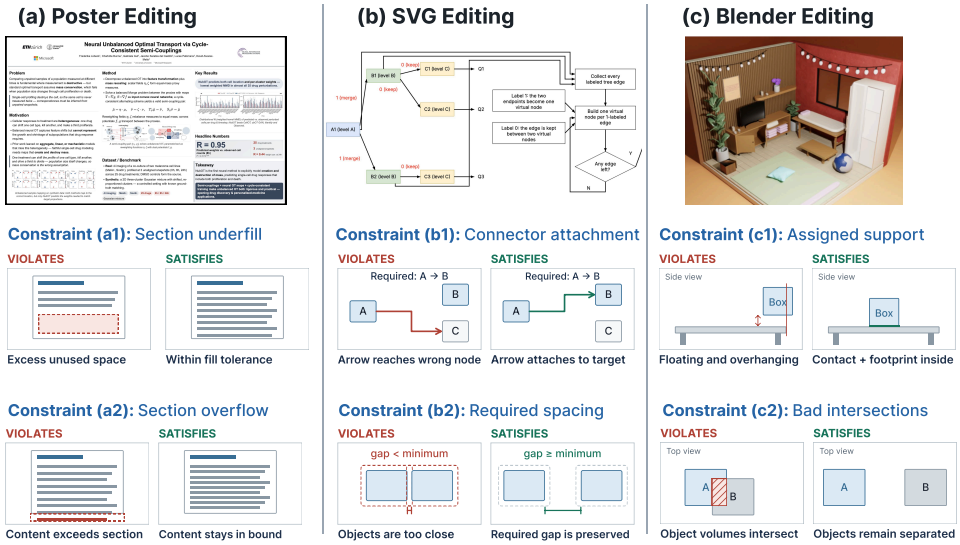


Figure 2: **Three artifact-editing domains.** (a) Poster editing with HTML/CSS. (b) SVG editing. (c) Blender scene editing. Each example illustrates representative hard constraints (simplified for presentation). Complete task specifications and constraints are in the Appendix

repair, restore an earlier version, or explore a different continuation. The agent chooses and executes the action, whose evaluated consequences inform the next memory update. Restoring an earlier artifact preserves the accumulated memory, allowing a new continuation to benefit from previously unsuccessful attempts. Section 5.3 examines whether this mechanism supports recovery from repair cycles, productive backtracking, and avoidance of repeated failures.

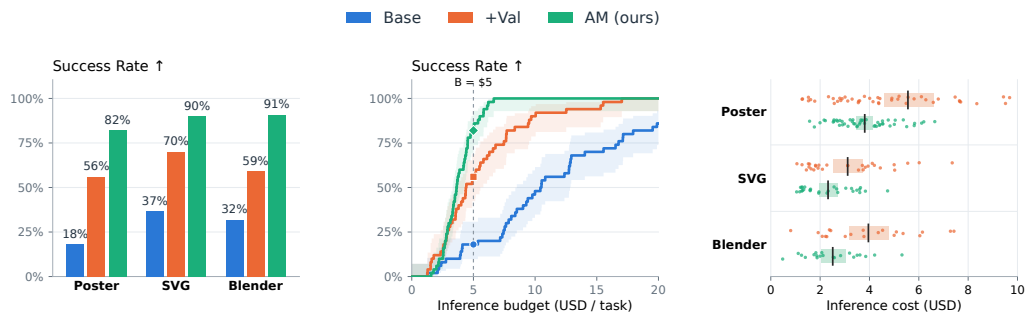
5 Experiments

5.1 Experimental Setup

Benchmark environments and tasks. We evaluate Artifact Memory on scientific poster editing, SVG editing, and Blender scene editing. We construct challenging variations of tasks in existing public benchmarks, including scientific poster generation (Pang et al., 2025; Xiao et al., 2026b), SVG generation and editing (Chen et al., 2025; Su et al., 2026), and Blender scene generation and editing (Gu et al., 2025; Yin et al., 2026; Gao et al., 2026; Pfaff et al., 2026; Raistrick et al., 2024). We add additional hard constraints and prepare initial artifacts that expose interactions among content, layout, and spatial relationships. The hard constraints specify concrete acceptance conditions on the source, its evaluated structure, or its rendered appearance. Figure 2 illustrates representative constraints examples. The resulting suite contains 50 poster editing tasks, 30 SVG editing tasks, and 22 Blender editing tasks. Detailed task specifications are provided in the Appendix.

Agents and baselines. We instantiate tool-using editing agents with Claude Opus 4.8 (Anthropic, 2026) and GPT-5.6 Terra (OpenAI, 2026) through a controlled multi-turn API harness. We compare editing agents without (Base) and with (+Val) per-turn validator feedback, and with our Artifact Memory method (AM), which additionally supplies memory in the context. All conditions retain a fixed-length window of conversation history. Runs follow domain-specific resource limits and stopping rules. Detailed agent implementation and method comparison are in the Appendix.

Evaluation protocol: performance and efficiency. We evaluate performance through validator-based constraint satisfaction and visual-judge scores for instruction fulfillment and visual quality. Validator metrics include success rate within the inference budget and the mean number of failed validator checks; visual-judge scores use a 1–5 scale on multiple dimensions of the instruction-following and visual quality of the artifacts. We measure efficiency using model turns and model-inference cost. One turn comprises one model response and its associated tool calls, and inference cost is reported in USD. Detailed evaluation protocols are provided in the Appendix.



(a) Success at fixed budgets. (b) Poster: success versus budget. (c) Inference cost among successful runs.

Figure 3: **Success rates and inference efficiency with Claude Opus 4.8.** (a) Constraint-satisfaction success under fixed budgets. (b) Poster success rate versus inference budget. (c) Inference cost among each method's successful runs. Full numerical results appear in Table 1, with additional plots in the Appendix.

(a) Performance

Method	Poster (N=50) SR(%)↑	Poster (N=50) #Fail↓	Poster (N=50) Judge↑	SVG (N=30) SR(%)↑	SVG (N=30) #Fail↓	SVG (N=30) Judge↑	Blender (N=22) SR(%)↑	Blender (N=22) #Fail↓	Blender (N=22) Judge↑
`claude-opus-4-8`									
Base	18	4.3	3.08	36.7	5.87	2.6	31.8	4.73	3.45
+Val	56	1.18	3.04	70	2.47	2.77	59.1	0.82	**3.91**
AM (ours)	**82**	**0.58**	**3.17**	**90**	**2.10**	**2.9**	**90.9**	**0.27**	3.86
`gpt-5.6-terra`									
+Val	66	1.08	2.57	80	2.97	2.07	72.7	1.73	3.7
AM (ours)	**80**	**0.48**	**2.71**	**90**	**0.93**	**2.2**	**86.4**	**1.09**	**3.75**

(b) Efficiency

Method	Poster (N=50) Turns↓	Poster (N=50) Cost($)↓	SVG (N=30) Turns↓	SVG (N=30) Cost($)↓	Blender (N=22) Turns↓	Blender (N=22) Cost($)↓
`claude-opus-4-8`						
+Val	81.4	5.56	26.1	3.12	30.1	3.96
AM (ours)	**47.8**	**3.81**	**15.5**	**2.32**	**20.3**	**2.51**
`gpt-5.6-terra`						
+Val	58.4	6.14	16.7	2.27	31.3	2.2
AM (ours)	**39.2**	**4.57**	**12.6**	**1.65**	**23.9**	**1.34**

Table 1: **Performance and efficiency across three artifact-editing benchmarks.** (a) SR: final constraint satisfaction within inference-cost budget (%); #Fail: mean failed validator checks at the evaluated checkpoint; Judge: mean overall visual-judge score (1–5). Base is evaluated only with `claude-opus-4-8`. (b) Mean model turns and inference cost (USD) to constraint satisfaction.

5.2 Main Results

Validator feedback strengthens the baseline. Table 1 and Figure 3 shows that explicit validator feedback substantially improves constraint satisfaction over the Base. With `claude-opus-4-8`, the success rate increases from 18% to 56% on Poster, from 36.7% to 70% on SVG, and from 31.8% to 59.1% on Blender, with fewer failed validator checks across all three domains. Mean visual-judge scores also improve on SVG (2.60 to 2.77) and Blender (3.45 to 3.91). These results show that measured constraint feedback can also benefit judged artifact quality.

Artifact Memory achieves the strongest overall performance across domains and models. Among the compared methods, Artifact Memory achieves the highest success rates and fewest failed validator checks in all settings (Figure 3 and Table 1). For example, with `claude-opus-4-8`, success reaches 82%, 90%, and 90.9% on Poster, SVG, and Blender, respectively, exceeding +Val by 20.0–31.8 percentage points. Artifact Memory also improves visual-judge scores; for example, `gpt-5.6-terra`'s Poster, SVG, and Blender scores increase from 2.57 to 2.71, from 2.07 to 2.2, and and from 3.7 to 3.75. These results demonstrate that Artifact Memory improves both constraint satisfaction, instruction fulfillment, and artifact quality overall.

Artifact Memory reduces editing turns and inference cost. On tasks solved by both methods under the efficiency protocol, Artifact Memory reaches full constraint satisfaction with fewer turns and lower inference cost (Table 1). Across all six model–benchmark settings, Artifact Memory reduces mean turns by 24–41% and mean inference cost by 26–39% relative to +Val. For example, on jointly solved `claude-opus-4-8` Poster tasks, mean turns decrease from 81.4 to 47.8 and mean cost from $5.56 to $3.81. Figure 3(c) shows the distributions for each method's own successful runs, showing that Artifact Memory improves both reliability and efficiency. Full results for both models and all three benchmarks and error analyses are provided in the Appendix.

5.3 Analysis and Ablation: How Artifact Memory Supports Effective Search

The main results establish Artifact Memory's improvements in artifact validity, quality, and editing efficiency. We now examine how these gains relate to the use of previous editing experience: recovering useful partial solutions, escaping repair cycles, and avoiding repeated failures.

Memory connects repair conflicts to a useful earlier version. Figure 4 illustrates this mechanism in a Blender scene. Successive edits reveal one object while obscuring another, transferring visibility failures among the board, planter, and plate. At version 10, retrieved edit consequences connect these attempts, and conflict analysis identifies how subsequent repairs have reversed earlier gains. The retrieved history also identifies version 8 as a useful partial solution: the planter and plate were visible together, although the board remained insufficiently visible. The agent restores this configuration and explores a different box placement coordinated with neighboring objects, satisfying all three illustrated visibility constraints. The example highlights two complementary roles: connected conflicts explain why to reconsider the current continuation, while retrieved partial progress identifies a concrete configuration from which to try another approach.

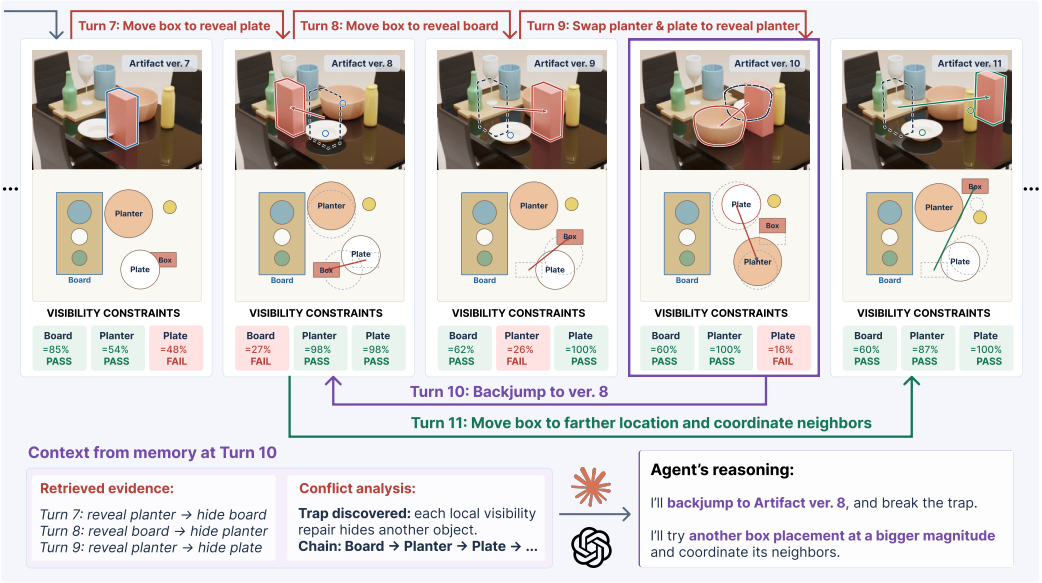


Figure 4: **Memory-guided backtracking in a Blender example.**

Quantitative analysis of agentic search behavior. We next turn to quantitative analysis of search behavior: recovery after repair cycles and the outcomes of backtracking. We analyze the Claude Opus 4.8 runs on 20 matched Poster tasks, 20 matched SVG tasks, and 20 matched Blender tasks across the validator baseline (+Val) method and our Artifact Memory method (AM). We use frontier model Claude Fable 5 to analyze and annotate the artifact editing trajectories. The analysis and annotation procedures are detailed in the Appendix.

Artifact Memory improves recovery from repair cycles. Table 2(a) evaluates each run's first qualifying repair cycle independently. A cycle occurs when a repair resolves a violation while introducing a linked failure on another component, and a later related repair reintroduces the earlier violation. *Recovery@*H measures the fraction of these episodes that achieve confirmed joint satisfaction of their recovery requirements within $H = 20$ additional agent turns. The recovery scope covers affected-column requirements in Poster, checks involving implicated entities in SVG, and retained whole-scene requirements in Blender.

Artifact Memory improves recovery from observed repair cycles, reducing capped recovery cost across all three domains: from 17.40 to 14.83 turns on Poster, 9.22 to 6.89 on SVG, and 14.53 to 10.85 on Blender. These reductions of approximately 15–25% measure progress toward jointly satisfying the recovery requirements, including the coupled constraints affected by previous repairs. Recovery rates also increase from 33.3% to 41.7% on Poster and from 41.2% to 84.6% on Blender, while remaining at 77.8% on SVG. Fewer AM runs encounter qualifying cycles on Poster (12/20

	Poster (N=20)		SVG (N=20)		Blender (N=20)	
(a) Recovery from observed repair cycles						
Method	Rec.@H ↑	Turns ↓	Rec.@H ↑	Turns ↓	Rec.@H ↑	Turns ↓
+Val	33.3% (5/15)	17.40	**77.8%** (7/9)	9.22	41.2% (7/17)	14.53
AM (ours)	**41.7%** (5/12)	**14.83**	**77.8%** (7/9)	**6.89**	**84.6%** (11/13)	**10.85**
(b) Backtracking and failed-trial repetition						
Method	Productive ↑	Repeat ↓	Productive ↑	Repeat ↓	Productive ↑	Repeat ↓
+Val	28.2% (35/124)	18.5% (23/124)	75.6% (25/33)	15.1% (5/33)	47.4% (9/19)	**0.0%** (0/19)
AM (ours)	**71.4%** (30/42)	**2.4%** (1/42)	**100.0%** (14/14)	**6.7%** (1/15)	**100%** (6/6)	**0.0%** (0/6)

Table 2: **Recovery and search behavior.** (a) Recovery rate within $H = 20$ turns of each run's first qualifying repair cycle and mean capped recovery turns; (b) Productive backtracking within 20 turns and repetition of an abandoned failing trial in the next relevant continuation.

versus 15/20) and Blender (13/22 versus 17/22). Together, these observations support the value of connecting edit consequences across versions: agents can use earlier gains and regressions to reconsider repairs that repeatedly transfer failures among components.

Artifact Memory supports more effective backtracking. Table 2(b) examines what happens after agents return to earlier relevant component configurations following measured unsuccessful trials. *Productive backtracking rate* is the fraction of eligible returns followed by a different continuation that jointly resolves the associated requirements within 20 turns, with confirmation from subsequent validation. *Failed-trial repetition rate* is the fraction of eligible returns whose next relevant continuation re-enters a previously abandoned failing trial under comparable conditions.

Artifact Memory improves the productivity of backtracking on Poster and reduces observed failure repetition on Poster and SVG. On Poster, productive backtracking increases from 28.2% (35/124) to 71.4% (30/42), while failed-trial repetition decreases from 18.5% (23/124) to 2.4% (1/42). AM therefore makes fewer returns, but a substantially larger fraction lead to successful continuations. This pattern supports more selective reuse of earlier configurations: retrieved partial progress provides useful starting points, while retained failed trials inform what to change after returning. Under the broader SVG definition, repetition decreases from 15.1% (5/33) to 6.7% (1/15). Together, the recovery and search analysis support Artifact Memory's central role in making editing experience actionable: helping agents reconsider unproductive repairs, recover useful partial solutions, and explore alternatives with knowledge of previous failures.

6 Limitations and Concluding Remarks

In this paper, we studied how memory can help agents manage long editing histories and reconsider decisions when local repairs stop making progress. We formulated artifact editing around task objectives, hard constraints, and measured edit consequences, then designed Artifact Memory to preserve and retrieve the evidence needed for subsequent decisions. Our evaluation examined both final outcomes and editing behavior, including recovery from repair cycles, productive backtracking, and repetition of failed continuations. The results support organizing memory around interactions among edits and constraints, so that returning to an earlier artifact also brings forward the experience gained since its first visit.

There are several limitations that can be addressed by future work. First, the effectiveness of our memory design depends on reliable validators and their mappings from rendered measurements to source components. Incomplete checks can miss important failures, and ambiguous attribution can misdirect conflict analysis. Moreover, observed improvements and regressions establish associations rather than causal explanations, especially when an edit changes several components at once. Retrieved failures are therefore conditional evidence, not grounds for ruling out an entire editing strategy. Second, the evaluation covers three domains and two models under specified constraints and budgets. The methods will likely need to be extended when transferring to less structured artifacts, subjective requirements, or substantially longer runs.

AI USE STATEMENT

We used generative AI tools to assist with code implementation, benchmark construction and manuscript polishing. We also used LLM-based editing agents and model-assisted trajectory annotation as described in the experimental setup and analysis. All AI-assisted activities were conducted under full author supervision. The authors take full responsibility for the final content of this work, including its text, code, benchmark tasks, analyses, and claims.

ETHICS STATEMENT

Artifact-editing agents can introduce inaccurate content, remove qualifications, or create misleading visual emphasis. A structural validity gate does not establish factual accuracy, accessibility, or appropriate use; these require task-specific checks and human judgment. The persistent edit graph can also retain information that was removed from the current artifact, so deployment should provide access controls and an explicit retention policy. Benchmark inputs and generated outputs should be used in accordance with their licenses and applicable privacy requirements.

REPRODUCIBILITY STATEMENT

Reproducing Artifact Memory requires the source snapshots or lossless changes, ordered transactions, validator reports, environment configuration, identity rules, and conflict retrieval settings. Given the same recorded inputs and fixed settings, edit-graph updates and conflict retrieval are deterministic; agent generations need not be. Appendix B specifies the edit-graph records and conflict retrieval policy, Appendix C describes the poster interface, Appendix **??** describes the screenshot validator, and Appendix D states the search instructions. The final experimental report should release these resources together with model versions and complete run configurations.

REFERENCES

Dan Abramov. Redux DevTools: Time-travel debugging for Redux. `https://github.com/reduxjs/redux-devtools`, 2015.

Anthropic. System card: Claude Opus 4.8. `https://www.anthropic.com/news/claude-opus-4-8`, May 2026. System card, 28 May 2026. Model ID `claude-opus-4-8`.

Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI)*, 1997.

Thomas Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*, 1(3):269–294, 1994.

Scott Chacon and Ben Straub. *Pro Git*. Apress, 2nd edition, 2014.

Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.

Siqi Chen, Xinyu Dong, Haolei Xu, Xingyu Wu, Fei Tang, Hang Zhang, Yuchen Yan, Linjuan Wu, Wenqi Zhang, Guiyang Hou, Yongliang Shen, Weiming Lu, and Yueting Zhuang. Svgenius: Benchmarking llms in svg understanding, editing and generation, 2025. URL `https://doi.org/10.1145/3746027.3758287`.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Toby Cubitt. undo-tree.el: Treat undo history as a tree. `https://www.dr-qubit.org/undo-tree.html`, 2009.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Mahir Labib Dihan, Tanzima Hashem, Mohammed Eunus Ali, and Md Rizwan Parvez. Web-operator: Action-aware tree search for autonomous agents in web environment. 2025. URL https://arxiv.org/abs/2512.12692.

James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. First return, then explore. *Nature*, 590:580–586, 2021.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014.

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of search (SoS): Learning to search in language. In *Conference on Language Modeling (COLM)*, 2024.

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D. Goodman. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective STaRs. *arXiv preprint arXiv:2503.01307*, 2025.

Yipeng Gao, Lei Shu, Genzhi Ye, Xi Xiong, Ameesh Makadia, Meiqi Guo, Laurent Itti, and Jindong Chen. 3dcodebench: Benchmarking agentic procedural 3d modeling via code, 2026. URL https://arxiv.org/abs/2606.01057.

John Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie Mellon University, 1979.

Yunqi Gu, Ian Huang, Jihyeon Je, Guandao Yang, and Leonidas Guibas. Blendergym: Benchmarking foundational model systems for graphics editing. 2025. URL https://arxiv.org/abs/2504.01786.

Yi Gui, Zhen Li, Zhongyi Zhang, Guohao Wang, Tianpeng Lv, Gaoyang Jiang, Yi Liu, Dongping Chen, Yao Wan, Hongyu Zhang, Wenbin Jiang, Xuanhua Shi, and Hai Jin. LaTCoder: Converting webpage design to code with layout-as-thought. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2025.

Ian Huang, Guandao Yang, and Leonidas Guibas. Blenderalchemy: Editing 3d graphics withnbsp;vision-language models. In *Computer Vision – ECCV 2024: 18th European Conference, Milan, Italy, September 29–October 4, 2024, Proceedings, Part LXXXIX*, pp. 297–314, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-73023-8. doi: 10.1007/978-3-031-73024-5_18. URL https://doi.org/10.1007/978-3-031-73024-5_18.

Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. 2026. URL https://arxiv.org/abs/2407.01476.

Runze Li, Yukun Zhao, Can Xu, Yucheng Shen, Shuaiqiang Wang, Jianmin Wu, Lingyong Yan, and Dawei Yin. From transient prompts to persistent control: Scientific poster generation via recursive semantic-geometric contracts. 2026. URL https://arxiv.org/abs/2609.17326.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology (CRYPTO)*, 1987.

OpenAI. GPT-5.6 system card. https://deploymentsafety.openai.com/gpt-5-6, July 2026. Covers the Sol, Terra and Luna tiers; released 9 July 2026. Model ID gpt-5.6-terra.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Wei Pang, Kevin Qinghong Lin, Xiangru Jian, Xi He, and Philip Torr. Paper2Poster: Towards multimodal poster automation from scientific papers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2505.21497.

Nicholas Pfaff, Thomas Cohn, Sergey Zakharov, Rick Cory, and Russ Tedrake. Scenesmith: Agentic generation of simulation-ready indoor scenes, 2026. URL https://arxiv.org/abs/2602.09153.

Pranav Putta, Edmund Mills, Naman Garg, Sumeet Motwani, Chelsea Finn, Divyansh Garg, and Rafael Rafailov. Agent q: Advanced reasoning and learning for autonomous ai agents. 2024. URL https://arxiv.org/abs/2408.07199.

Alexander Raistrick, Lingjie Mei, Karhan Kayan, David Yan, Yiming Zuo, Beining Han, Hongyu Wen, Meenal Parakh, Stamatis Alexandropoulos, Lahav Lipson, Zeyu Ma, and Jia Deng. Infinigen indoors: Photorealistic indoor scenes using procedural generation. 2024. URL https://arxiv.org/abs/2406.11824.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*, 2025.

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

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2Code: Benchmarking multimodal code generation for automated front-end engineering. *arXiv preprint arXiv:2403.03163*, 2024.

Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

Xiaoyan Su, Peijie Dong, Zhenheng Tang, Song Tang, Yuyao Zhai, Kaitao Lin, Liang Chen, Gai Yuhang, Yuyu Luo, Qiang Wang, and Xiaowen Chu. Vcg-bench: Towards a unified visual-centric benchmark for structured generation and editing, 2026. URL https://arxiv.org/abs/2605.15677.

Cong Wang and Yusheng Zheng. Fork, explore, commit: OS primitives for agentic exploration. *arXiv preprint arXiv:2602.08199*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel 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. In *International Conference on Learning Representations (ICLR)*, 2025.

Junde Wu, Minhao Hu, Jiayuan Zhu, Jiazhen Pan, Yuyuan Liu, Min Xu, and Yueming Jin. Git context controller: Manage the context of LLM-based agents like git. *arXiv preprint arXiv:2508.00031*, 2025.

Violet Xiang, Charlie Snell, Kanishk Gandhi, Alon Albalak, Anikait Singh, Chase Blagden, Duy Phung, Rafael Rafailov, Nathan Lile, Dakota Mahan, Louis Castricato, Jan-Philipp Franken, Nick Haber, and Chelsea Finn. Towards system 2 reasoning in LLMs: Learning how to think with meta chain-of-thought. *arXiv preprint arXiv:2501.04682*, 2025.

Jingyu Xiao, Jiantong Qin, Shuoqi Li, Man Ho Lam, Yuxuan Wan, Jen tse Huang, Yintong Huo, and Michael R. Lyu. Comuicoder: Component-based reusable ui code generation for complex websites via semantic segmentation and element-wise feedback. 2026a. URL https://arxiv.org/abs/2602.19276.

Lingao Xiao, Yalun Dai, Yangyu Huang, Qihao Zhao, Wenshan Wu, Hugo He, Ruishuo Chen, Jin Jiang, Qianli Ma, Jiahuan Zhang, Xin Zhang, Ying Xin, Yang Ou, Yan Xia, Scarlett Li, Longbo Huang, Zhipeng Zhang, Yang He, Kim Hui Yap, and Yan Lu. ResearchStudio-Reel: Automate the last mile of research from paper to poster, video, and blog. *arXiv preprint arXiv:2607.04438*, 2026b.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents. *arXiv preprint arXiv:2502.12110*, 2025.

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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Ke Yang, Yao Liu, Sapana Chaudhary, Rasool Fakoor, Pratik Chaudhari, George Karypis, and Huzefa Rangwala. Agentoccam: A simple yet strong baseline for llm-based web agents. 2025. URL https://arxiv.org/abs/2410.13825.

Tianyi Yang, Dawei Fu, Youpeng Wu, Zixun Kou, Linrui Chen, Ruobing Jiang, Zijian Wang, and Qiang Li. PosterHarness: Turning scientific poster generation into an auditable instruction-following benchmark. *arXiv preprint arXiv:2607.03006*, 2026a.

Xiao-Wen Yang, Xuan-Yi Zhu, Ding-Chu Zhang, Wen-Da Wei, Jie-Jing Shao, Zhi Zhou, Lan-Zhe Guo, and Yu-Feng Li. Step back to leap forward: self-backtracking for symbolic reasoning and planning in language models. In *Proceedings of the Fortieth AAAI Conference on Artificial Intelligence and Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence and Sixteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'26/IAAI'26/EAAI'26. AAAI Press, 2026b. ISBN 978-1-57735-906-7. doi: 10.1609/aaai.v40i33.39986. URL https://doi.org/10.1609/aaai.v40i33.39986.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: deliberate problem solving with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. AgentFold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025.

Shaofeng Yin, Jiaxin Ge, Zora Zhiruo Wang, Chenyang Wang, Xiuyu Li, Michael J. Black, Trevor Darrell, Angjoo Kanazawa, and Haiwen Feng. Vision-as-inverse-graphics agent via interleaved multimodal reasoning, 2026. URL https://arxiv.org/abs/2601.11109.

Zhisong Zhang, Tianqing Fang, Kaixin Ma, Wenhao Yu, Hongming Zhang, Haitao Mi, and Dong Yu. WebRollback: Enhancing web agents with explicit rollback mechanisms. In Vera Demberg, Kentaro Inui, and Lluís Marquez (eds.), *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 187–197, Rabat, Morocco, March 2026. Association for Computational Linguistics. ISBN 979-8-89176-381-4. doi: 10.18653/v1/2026.eacl-short.12. URL https://aclanthology.org/2026.eacl-short.12/.

Hao Zheng, Xinyan Guan, Hao Kong, Jia Zheng, Weixiang Zhou, Hongyu Lin, Yaojie Lu, Ben He, Xianpei Han, and Le Sun. PPTAgent: Generating and evaluating presentations beyond text-to-slides. *arXiv preprint arXiv:2501.03936*, 2025.

Shawn Wanxiang Zhong, Junxuan Liao, Jing Liu, Mai Zheng, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Don't let AI agents YOLO your files: Information and control in agent-native filesystems. *arXiv preprint arXiv:2604.13536*, 2026.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

A ARTIFACT MEMORY: FULL DESIGN

This appendix gives the full construction underlying Section 4. Artifact Memory combines an *edit graph* with *retrieval mechanisms*. The graph stores recoverable artifact versions through shared components and connects edits to requirement-level validation outcomes. Conflict records organize this evidence around unsuccessful repairs and interactions across component groups. At each iteration, retrieval selects relevant versions, failed attempts, and candidate return points as bounded working memory, presented to the agent as a digest. The agent uses this memory to choose and revise editing decisions; abandoned branches remain available to inform later continuations. The historical context allowance is fixed independently of run length, although persistent storage and retrieval computation can still grow. We use $v_t = V(x_t, o_t)$ and $h_{t,k} = h_k(x_t, o_t)$ throughout.

The full history at iteration t is

$$H_t = (x_0, o_0, v_0, a_0, x_1, o_1, v_1, \ldots, a_{t-1}, x_t, o_t, v_t).$$

Let M_t be persistent memory and D_t the working memory retrieved for the current iteration. In Artifact Memory, M_t comprises the edit graph and its conflict records, and conflict retrieval constructs D_t. We seek an update and retrieval mechanism

$$M_{t+1} = U(M_t, a_t, x_{t+1}, v_{t+1}), \qquad D_t = R(M_t, x_t, v_t, u; L), \qquad \text{tokens}(D_t) \le L, \quad (3)$$

where the memory allowance L is fixed independently of run length. The policy selects an edit from instruction u, a structured view of x_t and v_t, and retrieved history D_t. When rendered observations are supplied, this can be written as $a_t \sim \pi(\cdot \mid u, \text{view}(x_t, v_t), o_t, D_t)$, where view constructs the current structured view. The current observation and ordinary tool outputs have their own context costs; bounding D_t does not bound the entire prompt.

The edit graph must preserve concrete versions for recovery and link edits to their measured effects. Conflict retrieval must select the versions and outcomes relevant to the next decision, including evidence from abandoned branches. It need not retain every rendered frame in working context or replay every previous observation. The relevant search state is (x_t, M_t): returning to the same artifact with additional failed attempts in memory should permit a different continuation. We evaluate memory by the editing behavior and outcomes it supports under fixed budgets, rather than by transcript reconstruction alone.

Illustrated recovery and conflict retrieval. Figure **??** illustrates working memory on one poster column. (A) The edit graph. The trunk runs from V28 through V37 and V38 to the goal. Edit e29 added four words to the headline-numbers section (#hn) and overshot the band, so the agent reverted to V28 with e30. The return is linked to the existing state V28, and the abandoned state V29 is retained as a failed-branch record. Edits e37 and e38 then rewrote the same section by three words in each direction and traded the same pixels back and forth, a zero-sum trade. At V38 the conflict analysis fires. Edit e42 follows its suggestion, pulls a continuous padding lever sized from the recorded slope, and reaches the goal. (B) The conflict analysis the digest renders at V38: the pattern across the four unresolved conflicts on the column, the net effect on every check, a reflow victim that worsened without being edited, the sized suggestion to switch lever family, and the nogood N1 with its victims and its lesson (a recorded slope of about -14 px per word; the failed branch remains available as evidence against repeating the same attempt).

A.1 THE EDIT GRAPH: SHARED VERSIONS AND MEASURED EDITS

Shared state representation. Each artifact version is an immutable structured snapshot with a content identity. In the DOM setting, nodes record tags, editable attributes, text, and ordered children. A content-addressed store shares identical subtrees across versions, following persistent data structures and Merkle-style versioning (Driscoll et al., 1989; Merkle, 1987; Chacon & Straub, 2014). Stable component identifiers support correspondence across edits. A second comparison that omits only harness-generated identifiers can recognize restored content after those identifiers change. User-authored identifiers, meaningful whitespace, styles, and other rendering-relevant properties must be retained. Canonicalization must not merge artifacts merely because they look similar.

The snapshot also identifies the environment and validator configuration under which it was measured. Reports can be reused only when these dependencies match. Exact source recovery requires

either a lossless source representation or the original source bytes alongside the structural index. Structural sharing reduces duplication; it does not make arbitrary histories constant-size. If N_0 is the initial stored size and δ_t is the new material introduced at step t, storage is

$$O\left(N_0 + \sum_{i<t} \delta_i + |\mathcal{E}_t| + Q_t\right), \tag{4}$$

where $\mathcal{E}_t$ is the transition log and Q_t is the stored validation and metadata volume. A local tree change copies the affected paths; a global rewrite may introduce an entire new artifact. Parsing, rendering, and validation remain separate costs.

Transitions with measured outcomes. Let $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$ be the edit graph, whose nodes are stored artifact versions and whose edges are executed edit transactions. Each changed transaction adds an edge

$$e_t = (s_t, s_{t+1}, a_t, \mathcal{C}_t^{\text{edit}}, K_t, \Delta_t), \qquad \Delta_{t,k} = h_{t+1,k} - h_{t,k}, \tag{5}$$

where s_t identifies a stored version, $\mathcal{C}_t^{\text{edit}}$ is the set of changed component identifiers, and $K_t \subseteq \{k : h_k \in HC\}$ contains the indices of inferred target constraints. The sets $\mathcal{C}_{t,k}$ in v_t identify the components involved in each constraint, while $\xi_{t,k}$ retains its diagnostic measurements and verdict. Targets are inferred from changed sections and affected selectors, and can be uncertain for broad style changes. The edge records all measured effects, including changes to untargeted checks. Tree differencing provides edit descriptions such as additions, rewrites, moves, and restyles (Chawathe et al., 1996; Falleri et al., 2014). Measurements before and after a transaction establish its observed effect under a fixed environment; they do not isolate the contribution of each operation inside a batch.

Revisits without forgetting. A transition to an already stored artifact reuses its state node while retaining a new, timestamped edge. The edit graph can therefore contain cycles. The immutable subtree store is a directed acyclic graph, and the chronological event log remains ordered; these are different structures. An active path of state occurrences identifies the branch currently being extended. When the agent returns to an earlier occurrence, the intervening path remains recorded as an abandoned branch. Reusing a state never removes the experience gathered since its first visit.

Subtree identities also recognize partial recovery. Restoring one section while retaining changes elsewhere produces a new whole-artifact state, but the recovered component can still be linked to its earlier version. Its old validation result is evidence about that earlier context, not a certificate for the new composition. The resulting artifact is validated again.

Indices for historical access. Component-to-edit and check-to-outcome indices connect a current failure to relevant transitions. State hashes support lookup after the new snapshot has been parsed and hashed. These indices avoid repeatedly comparing full historical frames, although a similarly indexed log could provide the same access primitives. The contribution is the combination of identity, shared versions, measured transitions, and their use in selecting the agent's working memory.

A.2 Recording conflicts in the edit graph

A validator report tells the agent what fails now. Conflict records in the edit graph relate that failure to attempted repairs, neighboring requirements, and recoverable alternatives. They retain the core conflicts from unsuccessful editing sequences together with pointers to the versions and measured outcomes that support them. Conflict retrieval can then bring this evidence into the working memory when similar requirements or component groups are involved again.

Conflict detection. A repair can fail by making no measurable progress on a failing target or by creating a regression elsewhere. Let $\epsilon_k > 0$ be the minimum meaningful improvement and $\gamma_k > 0$ the regression tolerance for check k. We flag transaction e_t when

$$\text{stalled}(e_t) = \mathbf{1}[\exists k \in K_t :\ h_{t,k} > 0 \ \wedge\ h_{t,k} - h_{t+1,k} < \epsilon_k \ \wedge\ h_{t+1,k} > 0], \tag{6}$$

$$\text{regressed}(e_t) = \mathbf{1}[\exists h_k \in HC :\ (h_{t,k} = 0 \wedge h_{t+1,k} > 0) \ \vee\ \Delta_{t,k} \geq \gamma_k]. \tag{7}$$

Either condition creates a conflict record. A check that remains satisfied is not a stalled repair. A restoration is recorded separately: worsening a local check while returning to an earlier decision need not be classified as a failed repair. Initial construction is likewise distinguished from subsequent repair attempts.

Each record names the implicated transaction, its targets, every regressed check, and the relevant component group. A record is inactive when its target and regression checks pass, or when a changed structural context makes the old attempt inapplicable. It remains in memory and can become relevant again. This prevents a repaired target from hiding an unresolved side effect.

Failure patterns and conditional lessons. Diagnostic measurements, such as signed residuals relative to an acceptance band, distinguish a change in the wrong direction from an overshoot. Near-zero movement suggests an ineffective edit; target improvement accompanied by a neighboring regression suggests coupling. Several conflicts involving the same column or page band are summarized together to expose repeated trades, reversals, and effects on untouched components. Table **??** gives the associated search options. These are observations and hypotheses, not diagnoses guaranteed by the graph.

A failed branch record contains its starting version, executed transitions, endpoint, per-check net effects, and affected components. We use *nogood* as shorthand for this empirical record, following the motivation of dependency-directed backtracking (Stallman & Sussman, 1977). It does not denote a learned logical clause. Returning to a known failing artifact under the same validator reproduces its violations, but does not prove that no useful continuation exists from it. Similarly, reusing an edit family, or even an edit string, in a different starting state can have a different outcome. Exact re-entry flags and family-level warnings therefore guide the policy rather than forbid reachable states.

Local action estimates. An attempt ledger records the edit families tried on each target, their outcomes, and applicable families without observations. Families include text changes, component additions or removals, figure sizing, spacing, shared-style changes, and movement across layout groups. A batch can combine several families. For a scalar edit parameter q with signed diagnostic residual $r_{t,k}$ at step t, extracted from $\xi_{t,k}$, a local response estimate is

$$\widehat{\beta}_{k,q} = \mathrm{mean}_{i \in I_{k,q}} \frac{r_{i+1,k} - r_{i,k}}{q_{i+1} - q_i}, \qquad \Delta q \approx \frac{r_k^{\mathrm{goal}} - r_{t,k}}{\widehat{\beta}_{k,q}}, \tag{8}$$

where $I_{k,q}$ includes comparable, nonzero changes under compatible conditions. The digest exposes the sample count and observed variation. Mixed batches do not identify individual slopes, and unstable or near-zero estimates do not justify a sizing recommendation. Text reflow is discontinuous, so calibration supplies a trial size rather than a reliable inverse model.

A.3 Conflict retrieval for bounded working memory

The edit graph and conflict retrieval address complementary parts of the representational challenge. Structural sharing avoids duplicating unchanged artifact content in persistent storage. Conflict retrieval selects the history relevant to the current editing decision, so the agent need not receive every intermediate version and observation. The graph remains available even when an episode is absent from the digest or the conversation is compacted; a conflict from an old branch can be retrieved again when it becomes relevant.

Current observation. The structured view describes the current artifact: component identifiers and locations, measurements and verdicts, and source references for repair. It is regenerated from the current state and validator report. This view answers *what is wrong now*; the memory digest answers *what has already been tried and which earlier decisions may matter*. A rendered image may supplement either view, but is not required by the memory interface.

Conflict retrieval policy. Conflict retrieval starts from the currently failing checks and the latest transaction's effects. It follows check and component indices to active conflicts, related component groups, failed branches, and earlier versions before implicated edits. Standing commitments and relevant successful versions are also candidates. Candidates carry the source state and edge identifiers that ground them, so a lesson remains connected to a recoverable artifact.

```
e1[conflict]  rewrote(#A)  text 36->47w
  failure mode: wrong-sign (moved #A need 28px further from band;
                            invert the change at ~100% size)
  effect on target: #A need +28->+56 | +28px worse
  effect that propagates to others:
    #B need  +0->-25 | gate flipped P->F
    #C need -20->-40 | -20px worse
```

Figure 5: One conflict edge as the digest renders it (from the engine's test scenarios). The edge rewrote section #A, moved its own target the wrong way, and by reflow flipped #B from pass to fail and worsened #C. The digest reports the measured movement and a sized hint. It does not tell the agent to revert.

We use a deterministic priority policy under the token allowance L. The digest reserves a short header for the current state, revisit status, best validated state under the gate criterion, and remaining budget. It then selects: (i) active conflicts and their return candidates; (ii) applicable failed-branch lessons and standing commitments; (iii) attempted and untried edit families for the selected targets; and (iv) calibration and recent transitions that explain these records. Within a priority group, current-failure relevance precedes recency, with stable identifiers breaking ties. Repeated evidence is aggregated; lower-priority records are shortened or omitted when the budget is exhausted. Truncation is marked rather than silently presented as complete history. Exact packing rules are given in Appendix B.

This policy can retrieve a remote failure while omitting many recent but unrelated edits. It also retains the evidence needed to act: a candidate return includes the affected scope and a compact difference from the current version, rather than only an opaque state identifier. Larger recovery patches can require additional ordinary file reads or edits. A fixed digest cannot expose all details of an arbitrarily large past state at once.

A.4 Exploring editing sequences with Artifact Memory

The edit graph preserves alternative versions and the outcomes of earlier attempts; conflict retrieval exposes the evidence relevant to choosing among continuations. The agent continues to act through ordinary file tools and chooses which transaction to execute. The base interface provides no dedicated graph-query or restore action. Historical differences in the digest support recovery through ordinary editing, after which state and subtree identities verify what was actually restored.

Continue, coordinate, or backtrack. When a repair improves its target without material regression, the agent can continue the current branch. When several attempts exchange violations across a shared group, it can coordinate a batch or reopen the shared design decision. When the branch offers little progress, memory identifies earlier decision points by tracing the affected checks to transactions on the active path. The state before the earliest implicated transaction in the selected conflict episode is a candidate fork, not a proven point before failure became inevitable. Among candidates, the policy weighs residual violations, restoration scope, and the progress that a return would discard.

A whole-artifact return revisits a stored state. A scoped return reconstructs selected components from an earlier version while preserving the rest of the current artifact. Scoped recovery is appropriate only when the retained components' relevant boundary conditions, such as column width and shared styles, remain compatible; otherwise the revision must include the affected group. In both cases validation checks the resulting composition. The memory records an exact return only after observing matching content, regardless of the agent's stated intent.

Backtracking with accumulated experience. The abandoned branch and its core conflicts remain in the edit graph after recovery. Conflict retrieval brings applicable lessons from its attempts into the next working memory: use a different edit family, change a shared allocation, or retry at a finer scale when the measurements support doing so. Revisiting x_j therefore produces the search state (x_j, M_t), with $t > j$, rather than returning to the agent's earlier knowledge. This distinction is the algorithmic role of memory: a return changes the available continuation because the failed sequence has been retained and made relevant.

The editing loop. At each iteration the harness (1) supplies the current observation and selected digest; (2) executes the agent's transaction; (3) validates the resulting artifact; (4) adds its version and measured edit to the edit graph; and (5) updates conflict records, abandoned branches, and indices for the next conflict retrieval. A short playbook asks the agent to inspect relevant prior attempts, choose a repair scope and edit family, and validate any recovery (Appendix D). A gate-accepted submission or exhausted budget ends the run. The supplied memory and these instructions jointly define the method; behavior cannot be attributed to representation alone without controlling the instructions.

Scope of the algorithmic claim. Artifact Memory supports conflict-guided local exploration with one executed transaction at a time. It does not enumerate a frontier or claim complete, optimal, or soundly pruned search. Long detours and repeated conflicts remain possible. Its intended advantage is to reduce redundant exploration by making previous outcomes available at the decision where they matter. The persistent store grows with novel experience, while the historical material inserted into each prompt remains bounded. Whether that tradeoff improves valid completion, quality, or total cost is assessed empirically.

B EDIT-GRAPH RECORDS AND CONFLICT RETRIEVAL DETAILS

State and transition records. Each version node in the edit graph has a state record containing the lossless artifact reference, structural root identity, component identities, first-seen event, and validator/environment fingerprint. Each edit edge has a transition record containing source and destination state identifiers, event order, tool operations, changed components, inferred targets, and before/after measurements. Agent-stated intent may be retained as context but is not evidence of a successful recovery. Multiple operations between validations form one transaction; a turn with no artifact change does not create a changed-state edge. Repeated transitions retain distinct event identifiers even when their endpoints match.

Validation records. The validation report v_t stores violation scores $h_{t,k}$ together with diagnostic measurements, acceptance criteria, pass indicators, and links to affected components. Violation magnitudes use check-specific scales; Boolean failures can use unit magnitude. Signed residuals retain direction for repair and calibration. For example, a section-fit diagnostic can measure excess height or deviation from a prescribed fill band, depending on the task. These diagnostics supplement the nonnegative violations used for validity and conflict detection.

Conflict records. Each conflict stores its supporting transitions, target checks, regressed checks, component group, failure-pattern tags, and structural context. In the poster profile, a change below 5 is treated as negligible for response analysis, and worsening an already failing fit check by at least 15 is a material regression. A pass-to-fail transition is material regardless of its size. Other check types require their own units and thresholds. Multiple tags may apply to one transaction: an overshoot can also harm a neighbor. A conflict becomes inactive when all of its target and regression checks pass, or its recorded context no longer applies; neither case deletes the evidence.

Branches, returns, and re-entry. A branch record identifies the fork occurrence, ordered transitions, tip state, affected components, per-check endpoint differences, and conflict references. The active path uses event occurrences to disambiguate repeated visits to the same state. A confirmed whole-artifact return to an earlier active-path occurrence closes the intervening branch. A partial return identifies recovered subtrees and creates a new whole-artifact version. Re-entry matches under the configured identity rules are exact observations; warnings about analogous edit families are heuristic. Neither means that all future continuations are infeasible. A no-change operation is not a backtrack, and identical tool bytes on different starting artifacts do not establish state identity.

Candidate forks and recovery scope. For each selected conflict group, the check index retrieves the active-path transitions associated with its current failure episode. The version preceding the earliest implicated transition is a candidate fork. Evidence may support additional candidates, including the best previously validated state in the group. These are candidates for revising a decision, not uniquely correct causal explanations. The recovery record names the components to restore,

their old and current values or source differences, and the surrounding conditions on which the old measurements depended. A whole-state rank uses the number of failing hard checks, then task-normalized total severity, then event order. This rank is a gate-based heuristic and does not estimate the terminal judge.

Attempt and calibration ledgers. An attempt entry links a target and edit family to its applicable context, count, transitions, and outcomes. An untried family means one that is applicable under the task profile and has no recorded attempt in that context. It does not mean that every edit in a tried family has been exhausted. Calibration entries include parameter units, sample identifiers, local response estimates, and observed variation. Changes in layout regime make old estimates stale. A `frozen:` declaration records an agent commitment to preserve a component; the digest flags later edits to it, but the commitment can be explicitly revised when its assumptions change.

Deterministic packing for conflict retrieval. The task profile fixes the tokenizer, allowance L, compact record templates, and per-record limits. The header uses a reserved allowance that must fit within L. Candidate records are generated from current failing checks, the latest transaction, their component groups, and standing commitments. The priority classes are: active conflicts with candidate recovery information; applicable branch lessons and commitments; attempt ledgers; calibration and explanatory transitions. Within each class, records directly involving a current failure precede records linked through its group, followed by newer supporting events and stable record identifiers. Evidence shared by several records is rendered once with references.

The renderer tries each candidate in this order, first using its full template and then its compact template. It admits a record only if the complete record fits the remaining allowance; otherwise it moves to the next candidate. A small reserved footer reports omitted record counts. It never truncates in the middle of an identifier or recovery operation. If the header and footer cannot fit, the profile is invalid. An omitted record remains in the persistent store and can be selected again when the failure context changes. Replaying a trace reproduces the digest only when the templates, tokenizer, configuration, and any commitment declarations are also supplied.

Storage and execution costs. Path copying bounds the new structural storage for a local change by the affected nodes and ancestor paths, not by a constant independent of edit size. Whole-source parsing or hashing may still scan the artifact, and materializing a recovered file incurs output cost. Validator reports may be stored as checkpoints plus deltas where exact reconstruction is preserved. The bounded digest limits historical prompt tokens per iteration; it does not bound graph growth, retrieval computation, current-view tokens, or the number of iterations required to solve a task.

C HARNESS AND POSTER VALIDATOR DETAILS

Execution boundary. The editing harness provides generic file and shell tools in an isolated workspace and records model replies, tool calls, source changes, and validator reports. All artifact changes in one transaction are committed before the next validation. The task profile specifies the artifact path, renderer and checker, hard requirements, and run budgets. The run configuration sets the turn and measurement limits. The harness invokes the canonical gate directly and reads its subprocess status and structured output. A model's assertion that it is finished, or a transformed fragment of checker output, cannot substitute for this result. Exhausted runs retain their artifact and report unresolved violations.

Artifact structure. The poster input is a pre-extracted paper package with a structured `paper_spec.md` and figures. The output is a self-contained HTML document on a fixed canvas, such as landscape 60×36 inches or A0 portrait. The template contains fixed-height flex columns and section cards with `data-section` identifiers. Cards marked `.grow` absorb remaining column height; figures and reusable content widgets occupy space within cards. Layout, style, header, and color settings define the initial template. These shared layout choices are relevant boundary conditions for interpreting later edits.

Measurements and gate. For each section the checker records content height, available height, fill ratio, padding containment, and a signed fit residual. The task profile specifies the accepted fill

band. The displayed verdicts distinguish empty, sparse, full, spillage, and overflow states. Figure checks include broken images, containment, and whether a figure reaches the specified fraction of its card on at least one axis; the described profile uses 90%. Column capacity checks identify negative slack when rigid content cannot fit. Advisory polish signals, such as widows and trailing space, remain separate from hard validity checks.

The checker emits structured records for sections, figures, and columns. Its exit status distinguishes a pass, failed checks, and a measurement-budget stop. A preliminary feasibility check compares column capacity with figure floors and minimum text requirements. When this test fails, the agent must revise the layout allocation rather than continue small fill adjustments. This is a task-specific feasibility test, not a proof that the full generation task has no solution.

Memory interface. Section-fit checks attach to cards; column-capacity checks attach to their shared parent. The edit graph records both local changes and reflow effects on other cards. Changed selectors help infer targets for style edits, while the full validator report detects effects beyond those inferred targets. Render settings, assets, and checker configuration are included in the state dependencies. A recovered section is always checked inside the resulting poster rather than inheriting an old passing verdict.

D SEARCH PLAYBOOK

Artifact Memory supplies the following instructions with the current view and the digest selected by conflict retrieval. They define how the policy is encouraged to use memory; they do not add a hidden search controller or hard-prune actions.

1. Select a current failure and inspect the relevant historical attempts before editing. Consider both its target and the checks affected by earlier repairs.
2. Prefer an applicable untried edit family after repeated failures under comparable conditions. Do not interpret a failed family as proof that every edit in it is impossible.
3. Use a recorded response estimate to size a trial only when its units and layout conditions match. If evidence is weak or mixed, make a small diagnostic change.
4. When an edit conflicts with a requirement, inspect its failure pattern and measured side effects before extending the same approach.
5. For interacting requirements, consider changing their shared allocation or coordinating a batch across the group. Judge the complete transaction by its new validator report.
6. When backtracking is warranted, use the candidate fork and recovery difference to identify the earlier decision and the scope to reopen. Preserve other progress only when its boundary conditions remain compatible.
7. Verify the resulting state after recovery. A declared revert is not a confirmed return, and a recovered component must pass in its current surroundings.
8. Use the retained failed branch to choose a different continuation. Treat exact re-entry and analogous-failure warnings as evidence to assess, not as universal prohibitions.
9. Preserve settled components and explicit commitments where possible. Reopen them when changed dependencies require it, and update the commitment rather than silently relying on stale assumptions.

The context contains stable task instructions, a freshly generated conflict retrieval digest, the current structured view and validator report, a bounded recent interaction window, and the next instruction. Old digests need not be accumulated in the transcript. Conversation compaction does not erase the external edit graph; nevertheless, it can still affect non-artifact information that the graph does not represent. Deterministic memory construction and agent-controlled editing remain separate parts of the loop.

E Extended Related Work

Agent memory and context management. MemGPT manages information across memory tiers, and AgentFold compresses long interaction histories through context folding (Packer et al., 2023; Ye et al., 2025). A-MEM, Zep, and Mem0 organize persistent experience through structured notes, temporal relations, or extracted memories (Xu et al., 2025; Rasmussen et al., 2025; Chhikara et al., 2025). Reflexion retains verbal feedback across trials, while Self-Refine iterates on feedback within a refinement loop (Shinn et al., 2023; Madaan et al., 2023). Artifact Memory specializes these ideas to an evolving external artifact. Its memory links recoverable source versions, executed edits, and observations of their rendered consequences. Structural sharing reduces duplication, and conflict retrieval selects evidence about the current requirements, including failures from abandoned branches. This selection must preserve enough detail to revisit a concrete editing decision, beyond recalling a general lesson or summarizing a conversation.

Versioned workspaces and recoverable state. Persistent data structures, Merkle-style identities, and version control share unchanged data across versions (Driscoll et al., 1989; Merkle, 1987; Chacon & Straub, 2014); undo trees, selective undo, and time-travel debugging support revisiting earlier changes (Cubitt, 2009; Berlage, 1994; Abramov, 2015). Git Context Controller provides version-control operations over agent context, YoloFS retains branching filesystem history, and BranchFS supplies copy-on-write branches with commit and rollback semantics (Wu et al., 2025; Zhong et al., 2026; Wang & Zheng, 2026). These mechanisms preserve what can be restored. Artifact Memory additionally records what each edit did to the artifact's measured requirements and retrieves that evidence when choosing a return point. Source identity alone does not establish the validity of a rendered composition: restoring a component under changed surrounding geometry, styles, or assets requires revalidation. Our base interface supplies recovery differences through memory, with restoration performed through ordinary file edits.

Explicit search and return-based exploration. Tree of Thoughts, LATS, Agent Q, and tree search for web agents explore alternative reasoning or action paths using explicit search procedures (Yao et al., 2023; Zhou et al., 2024; Putta et al., 2024; Koh et al., 2026). Go-Explore separates returning to a previously reached state from exploring beyond it (Ecoffet et al., 2021), while Tree-GRPO uses tree-structured interaction rollouts for agent reinforcement learning (**?**). These works establish the usefulness of revisiting decisions and allocating computation across alternatives. Artifact Memory instead studies the evidence available to an agent editing one artifact in place. The edit graph retains executed branches, and conflict retrieval exposes relevant attempts and return candidates; the agent chooses the next edit without a separate frontier-expansion controller. A return preserves the experience accumulated on the intervening branch, allowing the same artifact state to support a different continuation.

Choosing when and where to backtrack. Several agent systems explicitly encourage selective recovery. AgentOccam introduces `branch` and `prune` actions for revising plans and selectively replaying history; these planning actions do not themselves restore the web environment (Yang et al., 2025). WebRollback separates a critique that decides whether to continue from a rollback module that selects an earlier step, enabling multi-step returns (Zhang et al., 2026). WebOperator combines action-aware search with reversibility checks and verified replay for safer backtracking (Dihan et al., 2025). These are direct precedents for making recovery a deliberate choice. Our emphasis is how to ground that choice in artifact-specific evidence: vectors of requirement outcomes reveal improvements and regressions across component groups, and their links to edits identify decisions worth reopening. Conflict retrieval bounds the historical evidence presented to the agent; scoped recovery can preserve compatible progress elsewhere, while source identity and fresh validation check the resulting return. We use *backjumping* for a return to an implicated earlier decision that may skip several recent edits, without claiming that the selected point is a uniquely correct cause of failure.

Learning backtracking and self-correction. Stream of Search trains models on serialized search trajectories that include unsuccessful paths (Gandhi et al., 2024). Self-backtracking training teaches models to signal when to retreat during symbolic reasoning and planning (Yang et al., 2026b), and controlled studies of reasoning behaviors identify backtracking and verification as useful ingredients for subsequent self-improvement (Gandhi et al., 2025). DeepSeek-R1 and Meta Chain-of-Thought

further motivate reasoning that revises intermediate decisions (DeepSeek-AI, 2025; Xiang et al., 2025). In interactive settings, WebCoT distills reflection, branching, and rollback rationales into web-agent training (**?**). These approaches develop the policy's capacity to revise its decisions. Artifact Memory supplies the external evidence needed to exercise that capacity during artifact editing, without additional model training. Recovery must change the actual source and recheck its rendered consequences; revising a reasoning trace alone is insufficient. Memory and the accompanying search instructions jointly support this behavior.

Coding agents and execution feedback. SWE-agent designs interfaces for repository navigation, source editing, and test execution (Yang et al., 2024); OpenHands provides agents with file, command-line, and browser interaction in executable environments (Wang et al., 2025). Artifact editing can use these same tools, and software behavior also depends on execution rather than source text alone. The additional requirement in our setting is to track rendered properties such as layout, clipping, occlusion, and visual fidelity, which successful program execution does not certify. Such properties can be tested, but require dedicated validators that inspect rendered outputs and evaluated structure and map their findings to editable components. A source snapshot remains recoverable even when its rendered observations have been omitted from context; conversely, a screenshot alone does not preserve the source needed for recovery. Artifact Memory connects both through the edit graph, retaining requirement-level outcomes and observed interactions alongside source versions. Conflict retrieval then selects evidence for a repair or return, and revalidation checks whether recovered components remain valid under their new dependencies. This memory complements the editing and execution capabilities of coding-agent frameworks.

Artifact generation across the code–rendering gap. Design2Code and LaTCoder study producing web code from visual references (Si et al., 2024; Gui et al., 2025); PPTAgent, Paper2Poster, and ResearchStudio-Reel generate editable presentations, posters, and related research artifacts (Zheng et al., 2025; Pang et al., 2025; Xiao et al., 2026b). These tasks separate the representation an agent edits from the result a user sees. Layout, occlusion, and visual fidelity remain implicit in the source until the rendering engine evaluates it. BlenderAlchemy directly addresses this separation through visual edit generation, evaluation of multiple program hypotheses, and hypothesis reversion (Huang et al., 2024). It is therefore a close precedent for both visual feedback and recovery in artifact editing. Our focus is persistent memory across a long editing history: retaining shared versions and requirement-level outcomes, then retrieving conflicts that inform later repairs and returns.

Validators as a record of artifact observations. Visual refinement already uses feedback more specific than a terminal score. Paper2Poster's render-and-comment loop checks panels, ComUICoder produces element-wise feedback from matched UI elements, and PosterHarness evaluates auditable poster requirements (Pang et al., 2025; Xiao et al., 2026a; Yang et al., 2026a). PosterVisor compiles semantic and geometric requirements into contracts and rechecks affected checkpoints during repair (Li et al., 2026). Artifact Memory builds on this need for explicit observation. We design separate, task-specific validators to inspect rendered outputs and evaluated structure, associate measurements and violations with editable components, and store these reports alongside the corresponding versions and edits. The graph can maintain code compactly, but a source difference alone cannot establish which visual requirements improved or regressed. Recording those observations makes them comparable across iterations and available for conflict retrieval without accumulating every rendered frame in context. Because a recovered component can behave differently in new surroundings, it must be validated again. The distinction is the connection between validation, persistent edit history, and retrieval for search; component-level feedback or rechecking alone is not our novelty.

Conflict-directed backjumping and structural analysis. Dependency-directed backtracking and backjumping use failure dependencies to revisit relevant decisions, while conflict-driven satisfiability solvers retain learned constraints to avoid repeated search (Stallman & Sussman, 1977; Gaschnig, 1979; Marques-Silva & Sakallah, 1999; Bayardo & Schrag, 1997). Structural differencing identifies changes within hierarchical artifacts (Chawathe et al., 1996; Falleri et al., 2014). Artifact Memory combines these motivations under an incomplete model of rendering and component interaction. A conflict links observed violations to executed edits and recoverable versions, but does not establish a logical implication or prove an edit family infeasible. Failed-branch records are conditional

empirical evidence, used to select working memory and candidate return points. This preserves the connection between the paper's two motivations: compact access to past observations supports informed exploration, and current conflicts determine which past observations deserve that access.

F EXPERIMENTS

F.1 EXPERIMENTAL SETUP

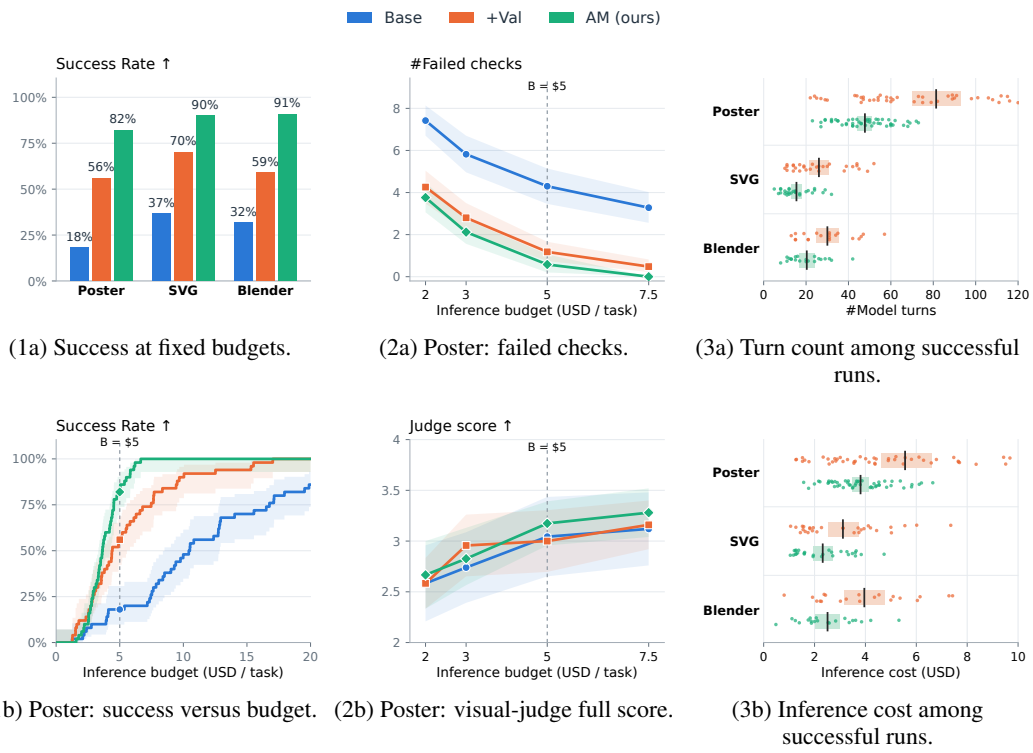
Artifact-editing tasks. We evaluate Artifact Memory on scientific poster editing, SVG editing, and Blender scene editing. These domains instantiate the artifact editing problem in Section 3 with different artifact sources and rendering engines. Across all three domains, agents iteratively edit artifacts according to natural-language instructions, using reference materials and optional validator feedback to fulfill the requests while satisfying hard constraints on the resulting artifact.

To create a more demanding evaluation for frontier models with well-designed agent harnesses, we construct new tasks based on existing benchmarks, including scientific poster generation (Pang et al., 2025; **?**), SVG generation and editing (Chen et al., 2025; Su et al., 2026), and Blender scene generation and editing (Gu et al., 2025; Yin et al., 2026; Gao et al., 2026; Pfaff et al., 2026; Raistrick et al., 2024). We introduce new task objectives and hard constraints, and prepare initial artifacts that expose interactions among content, layout, and spatial relationships. The resulting suite contains 50 poster editing tasks, 30 SVG editing tasks, and 22 Blender editing tasks. Detailed task specifications are provided in the Appendix.

Task objectives and hard constraints. The tasks require changes to an artifact's content or arrangement while preserving designated properties. Hard constraints specify concrete acceptance conditions on the source, its evaluated structure, or its rendered appearance. Figure 2 illustrates representative constraints examples. ① **Poster editing.** The agent edits HTML/CSS to organize paper content and figures into a scientific poster. The constraints mainly govern how content occupies the available space. For example, the HTML sections must satisfy specified fill bounds without excessive underfill or overflow. ② **SVG editing.** The agent revises SVG graphics to incorporate requested content and relationships. The constraints mainly preserve required elements, text, and connectivity, while enforcing label containment, object separation, and connector attachment. For example, adding a node to a diagram must preserve the required connections while maintaining clearance between connectors and unrelated objects. ③ **Blender editing.** The agent edits a Python program to rearrange objects in a Blender scene to fulfill new spatial requirements. The constraints mainly focus on object support, containment, and visibility. For example, moving a box to reveal one object must keep the box supported and preserve sufficient visibility of other objects. Detailed task objectives, constraints, and validator implementations are provided in the Appendix.

Agents and compared methods. We instantiate tool-using editing agents with Claude Opus 4.8 (Anthropic, 2026) and GPT-5.6 Terra (OpenAI, 2026) through a controlled multi-turn API harness. We compare editing agents without (Base) and with (+Val) per-turn validator feedback, and with our Artifact Memory method (AM), which additionally supplies memory in the context. All conditions retain a fixed-length window of conversation history. At each turn, the agent reasons and selects tool calls to inspect the rendered artifact, modify its source, or read relevant assets. The harness executes these calls and returns their outputs, including validator feedback when enabled. Runs follow domain-specific resource limits and stopping rules. Detailed agent loop implementation and method comparison setup are provided in the Appendix.

Evaluation protocol: performance and efficiency. We evaluate performance through validator-based constraint satisfaction and visual-judge scores for instruction fulfillment and visual quality. Validator metrics include success rate within the inference budget and the mean number of failed validator checks; visual-judge scores use a 1–5 scale on multiple dimensions of the instruction-following and visual quality of the artifacts. We measure efficiency using model turns and model-inference cost. One turn comprises one model response and its associated tool calls, and inference cost is reported in USD. Detailed evaluation protocols are provided in the Appendix.



(1a) Success at fixed budgets.

(2a) Poster: failed checks.

(3a) Turn count among successful runs.

(1b) Poster: success versus budget.

(2b) Poster: visual-judge full score.

(3b) Inference cost among successful runs.

Figure 6: Main results with Claude Opus 4.8. (1a) Success at per-task budgets of $5 for Poster, $4 for SVG, and $5 for Blender. Success requires final validator PASS and total run cost within budget. (1b, 2a, 2b) Poster success rate, mean failed checks, and mean overall visual-judge score (1–5) versus budget; dashed lines mark $5. Judge scores use matched evaluated subsets across methods ($n = 24, 23, 23, 25$ at $2, $3, $5, and $7.50). (3a–3b) Model turns and inference cost for each method's own successful runs across all benchmarks; dots show runs and black bars show means. Shading shows 95% Wilson intervals for success rates and task-bootstrap intervals for means. Points above 120 turns or $10 are omitted only from the respective displays; means and intervals retain all successful runs. Full results for both models and all benchmarks are in the appendix.

F.2 Main Results

Validator feedback improves constraint satisfaction. Table **??** shows that explicit validator feedback substantially improves constraint satisfaction over Base. With Claude Opus 4.8, success increases from 18% to 56% on Poster, from 36.7% to 70% on SVG, and from 31.8% to 59.1% on Blender. The mean number of failed checks also decreases in all three domains. These results support providing measured constraint feedback during editing. However, +Val still leaves 30–44% of tasks unsuccessful within the reporting budgets, motivating additional support for using editing experience.

Artifact Memory improves success across domains and models. Adding Artifact Memory to validator feedback improves success in all six model–benchmark settings. For Claude, AM achieves success rates of 82%, 90%, and 90.9% on Poster, SVG, and Blender, respectively, improving over +Val by 26.0, 20.0, and 31.8 percentage points. For GPT-5.6 Terra, the corresponding rates are 80%, 90%, and 86.4%, with improvements of 10–14 percentage points over +Val. AM also reduces the mean number of failed checks in every setting; for example, Claude's Poster and Blender means decrease from 1.18 to 0.58 and from 0.82 to 0.27. The Poster budget curves provide a complementary view: AM attains high success rates at smaller budgets and has fewer failed checks at every plotted budget (Figure **??**, panels 1b and 2a). The benefit therefore extends beyond a single reporting threshold.

(a) Performance

Method	Poster (N=50) SR(%)↑	Poster #Fail↓	Poster Judge↑	SVG (N=30) SR(%)↑	SVG #Fail↓	SVG Judge↑	Blender (N=22) SR(%)↑	Blender #Fail↓	Blender Judge↑
`claude-opus-4-8`									
Base	18	4.3	3.08	36.7	5.87	2.6	31.8	4.73	3.45
+Val	56	1.18	3.04	70	2.47	2.77	59.1	0.82	**3.91**
AM (ours)	**82**	**0.58**	**3.17**	**90**	**2.10**	**2.9**	**90.9**	**0.27**	3.86
`gpt-5.6-terra`									
+Val	66	1.08	2.57	80	2.97	2.07	72.7	1.73	3.7
AM (ours)	**80**	**0.48**	**2.71**	**90**	**0.93**	**2.2**	**86.4**	**1.09**	**3.75**

(b) Efficiency

Method	Poster (N=50) Turns↓	Poster Cost($)↓	SVG (N=30) Turns↓	SVG Cost($)↓	Blender (N=22) Turns↓	Blender Cost($)↓
`claude-opus-4-8`						
+Val	81.4	5.56	26.1	3.12	30.1	3.96
AM (ours)	**47.8**	**3.81**	**15.5**	**2.32**	**20.3**	**2.51**
`gpt-5.6-terra`						
+Val	58.4	6.14	16.7	2.27	31.3	2.2
AM (ours)	**39.2**	**4.57**	**12.6**	**1.65**	**23.9**	**1.34**

Figure 7: **Performance and efficiency across three artifact-editing benchmarks.** (a) SR: final validator PASS within the total inference-cost budget (%); #Fail: mean failed blocking checks at the evaluated checkpoint; Judge: mean overall visual-judge score (1–5) on the evaluated subset. Base is evaluated only with Claude. (b) Mean model turns and inference cost (USD) to first full validator PASS under a separate efficiency protocol, using the same jointly solved subset for both methods within each model–benchmark pair. N is the full benchmark size, not a subset size. Arrows indicate better values; bold marks the best observed value within each model, benchmark, and metric. See the appendix for budgets, evaluated-subset sizes, confidence intervals, and full results.

AM improves mean visual-judge scores in five of six settings. On the evaluated subsets, AM obtains higher mean overall judge scores than +Val in five of the six model–benchmark settings (Table **??**). For Claude, Poster and SVG scores increase from 3.04 to 3.17 and from 2.77 to 2.90, while Blender decreases slightly from 3.91 to 3.86. For GPT-5.6 Terra, scores increase in all three domains. These changes are modest, and the Poster budget curve shows that AM does not lead in judge score at every budget (Figure **??**, panel 2b). The strongest and most consistent improvements concern constraint satisfaction, with smaller changes in judged instruction fulfillment and visual quality.

AM reaches validator PASS with fewer turns and lower cost. Table 7 compares the methods on tasks that both solve under the separate efficiency protocol, measuring turns and inference cost to the first full validator PASS. Across the six model–benchmark settings, AM reduces mean turns by approximately 24–41% and mean inference cost by 26–39% relative to +Val. For example, on jointly solved Claude Poster tasks, mean turns decrease from 81.4 to 47.8 and mean cost from $5.56 to $3.81. These matched-task comparisons show that the higher success rates are accompanied by lower search cost on shared successes. Figure **??** (panels 3a–3b) additionally shows the distributions for each method's own successful runs, which need not contain the same tasks. Section 5.3 examines how memory influences the editing trajectories underlying these results.

Full results for both models and all three benchmarks, including budget-dependent curves, evaluated-subset sizes, confidence intervals, and detailed efficiency comparisons, are provided in the Appendix.

Even when requirements are stated, visual inspection alone does not reliably produce compliant artifacts under a fixed budget. Validator feedback improves constraint satisfaction while preserving or improving independently assessed visual quality. Compare B and B+V

F.3 Analysis and Ablation: How Artifact Memory Supports Effective Search

The main results establish the benefits of validator feedback and Artifact Memory under a fixed budget. We further investigate how Artifact Memory helps agents make the right editing decisions.

Measured edit consequences provide evidence. Comparing validator measurements before and after an edit reveals which constraints improved or regressed, and in which components those consequences occurred. These edit consequences, linked to the affected components and artifact version, become an *evidence*. The effects may be locally confined or exhibit *cross-component effect propagation*, where editing one component changes the constraint status of other components. Figure 4 is an example of it: moving an object to reveal a planter can obscure a board from the evaluation camera. Such effect propagation arises through the rendering process and the shared spatial relationships among components. Propagation can produce improvements as well as regressions, and we call it *repair conflict* if it produce regressions.

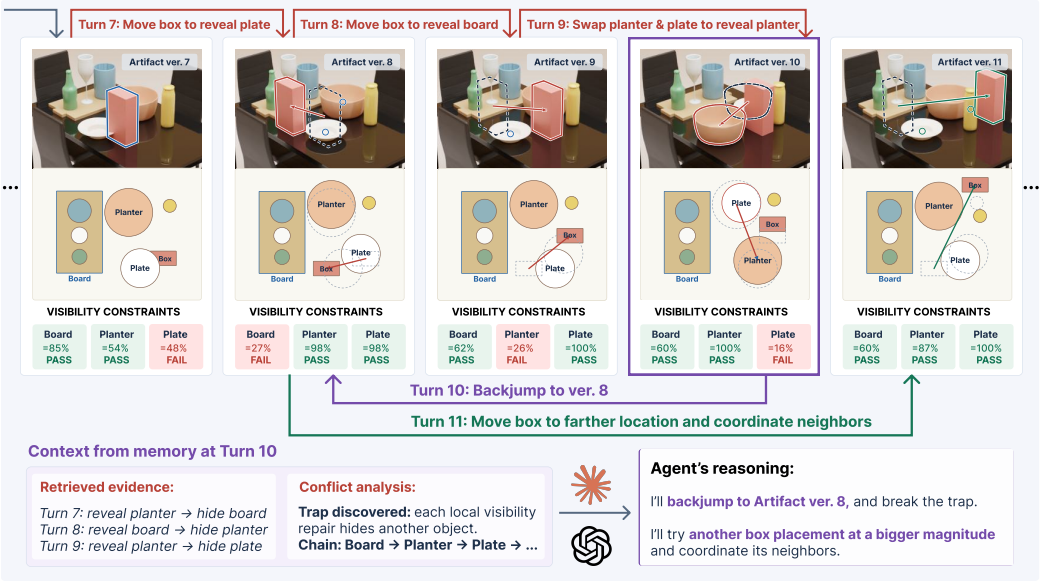


Figure 8: **Artifact Memory overview.** Across HTML, SVG, and Blender artifacts, an edit graph shares source components and records validation outcomes; bounded conflict retrieval guides coordinated repairs and backtracking.

Connecting evidence reveals recurring repair cycles. An individual repair conflict may be resolved by a subsequent edit. A more difficult situation arises when related repairs repeatedly undo earlier progress. We call this pattern a *repair cycle*, or a trap: a sequence of related repairs in which a later repair reintroduces a previously repaired violation. Validator feedback exposes each transition, but recognizing this *temporal pattern* requires connecting evidence across artifact versions. Artifact Memory supports this process through its three components. The artifact version graph associates attempts and outcomes with specific configurations and preserves earlier return points. Relevance-based retrieval selects previous edits involving the current components and requirements, including configurations that achieved useful partial progress. Conflict analysis connects these records to identify recurring regressions and reversals. Together, they provide evidence about both why the current continuation is unproductive and which earlier configuration offers a useful starting point for another attempt.

Agent learns to backtrack and discover a new branch. The individual transitions exhibit cross-component repair conflicts; their connection reveals a continuation that repeatedly transfers the visibility violation among objects. At version 10, the relevant memory context brings these transitions together with an earlier partial success: in version 8, the planter and plate were simultaneously visible, although the board remained insufficiently visible. The conflict analysis identifies the recurring exchange of violations, while the retrieved version provides a concrete return point with known benefits and costs. Returning to version 8 discards the subsequent compensating edits and recovers that earlier arrangement.

The agent then explores a different box placement coordinated with neighboring objects and checks the visibility requirements jointly. This continuation satisfies all three illustrated requirements. The example shows complementary roles for conflict analysis and retrieval: the former provides a reason to reconsider the current continuation, and the latter supplies a useful configuration from which to try another. The agent chooses and executes the return and the new edits.

Recovery from repair cycles. We evaluate whether this mechanism helps agents recover from recurring conflicts using a common, prefix-based cycle detector across methods. The detector uses only edits and measurements available up to each checkpoint; it does not depend on whether a method emits a memory warning. Detection criteria and domain-specific measurement tolerances are specified in Appendix.

	Poster (N=20)		SVG (N=20)		Blender (N=20)	
(a) Recovery from shared repair-cycle checkpoints						
Method	Rec.@H ↑	Turns ↓	Rec.@H ↑	Turns ↓	Rec.@H ↑	Turns ↓
+Val	33.3% (5/15)	17.40	**77.8%** (7/9)	9.22	41.2% (7/17)	14.53
AM (ours)	**41.7%** (5/12)	**14.83**	**77.8%** (7/9)	**6.89**	**84.6%** (11/13)	**10.85**
(b) Search behavior in complete runs						
Method	Productive ↑	Repeat ↓	Productive ↑	Repeat ↓	Productive ↑	Repeat ↓
+Val	28.2% (35/124)	18.5% (23/124)	75.6% (25/33)	15.1% (5/33)	47.4% (9/19)	**0.0%** (0/19)
AM (ours)	**71.4%** (30/42)	**2.4%** (1/42)	**100.0%** (14/14)	**6.7%** (1/15)	**100%** (6/6)	**0.0%** (0/6)

Table 3: **Recovery and search behavior.** (a) Recovery rate within H turns and capped recovery cost. (b) Productive backtracking rate and failed-trial repetition rate. B+V denotes baseline with validator feedback. All rates are percentages. We enable backtracking in SKILL.

To compare recovery under the same difficulty, we resume identical repair-cycle checkpoints with and without Artifact Memory. Both conditions receive the same artifact, ordinary interaction history, validator feedback, model configuration, and remaining budget. The memory condition additionally receives its retrieved evidence and conflict analysis, constructed solely from that history.

We report *Recovery@*H, the fraction of episodes recovered within H additional agent turns, and *capped recovery cost*, the mean number of turns to recovery, capped at H. Unrecovered episodes contribute H to the latter. Recovery requires the episode's involved requirements to become simultaneously satisfied and remain satisfied over the confirmation window, without transferring the violation to another implicated component. Table 2a shows that Artifact Memory increases recovery within the allotted horizon and reduces recovery cost. These results support the benefit of organizing historical evidence beyond providing the current validator report.

Productive returns and avoidance of failed trials. We additionally analyze search behavior in complete runs. The *productive backtracking rate* is the fraction of verified, conflict-related returns followed by a different continuation that resolves the associated conflict within the evaluation horizon. This measure includes backjumps over multiple editing decisions; return verification and continuation matching are described in Appendix. We report the number of eligible returns alongside the rate, since frequent backtracking alone does not establish effective search.

The *failed-trial repetition rate* measures how often a return to an earlier configuration is followed by re-entering an already abandoned failing continuation. We count only opportunities for which such a continuation is known, and require comparable relevant component configurations. This avoids treating a justified retry under changed conditions as a repeated failure.

Table 2b shows that returns under Artifact Memory more often lead to successful alternative continuations and less often repeat previously abandoned failures. Together with repair-cycle recovery, these results indicate that memory supports more effective agentic search: agents use prior outcomes to reconsider unproductive continuations, recover useful partial solutions, and explore alternatives with less repetition.

Table 4: Performance on three artifact-editing benchmarks. SR is the task success rate (%): final validator PASS with total inference cost within the reporting budget. #Fail is the mean number of failed blocking checks at the evaluated checkpoint, and Judge is the mean overall visual-judge score (1–5) on the evaluated subset. Higher SR and Judge and lower #Fail are better. N denotes the full benchmark size; judge scores use only evaluated tasks. Base is evaluated only with Claude. The best observed value within each model, benchmark, and metric is bolded. See the appendix for reporting budgets, evaluated-subset sizes, confidence intervals, and full results.

	Poster (N=50)			SVG (N=30)			Blender (N=22)		
Method	SR(%)↑	#Fail↓	Judge↑	SR(%)↑	#Fail↓	Judge↑	SR(%)↑	#Fail↓	Judge↑
`claude-opus-4-8`									
Base	18	4.3	3.08	36.7	5.87	2.6	31.8	4.73	3.45
+Val	56	1.18	3.04	70	2.47	2.77	59.1	0.82	**3.91**
AM (ours)	**82**	**0.58**	**3.17**	**90**	**2.10**	**2.9**	**90.9**	**0.27**	3.86
`gpt-5.6-terra`									
+Val	66	1.08	2.57	80	2.97	2.07	72.7	1.73	3.7
AM (ours)	**80**	**0.48**	**2.71**	**90**	**0.93**	**2.2**	**86.4**	**1.09**	**3.75**

Table 5: Efficiency on jointly solved tasks under the separate efficiency evaluation protocol. We report mean model-inference cost (USD) and model turns to the first full validator PASS. Within each model–benchmark pair, both methods are evaluated on the same subset of tasks that both solve under this protocol. N denotes the full benchmark size, not the size of the jointly solved subset. Lower is better; the lowest mean within each model, benchmark, and metric is bolded. Additional results are provided in the appendix.

	Poster (N=50)		SVG (N=30)		Blender (N=22)	
Method	Turns↓	Cost($)↓	Turns↓	Cost($)↓	Turns↓	Cost($)↓
`claude-opus-4-8`						
+Val	81.4	5.56	26.1	3.12	30.1	3.96
AM (ours)	**47.8**	**3.81**	**15.5**	**2.32**	**20.3**	**2.51**
`gpt-5.6-terra`						
+Val	58.4	6.14	16.7	2.27	31.3	2.2
AM (ours)	**39.2**	**4.57**	**12.6**	**1.65**	**23.9**	**1.34**

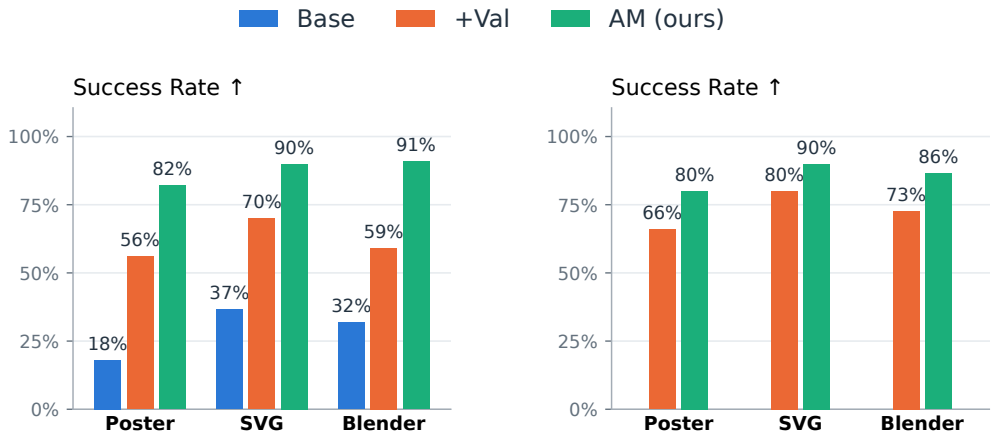
G FULL EXPERIMENT RESULTS

Table 6: Performance on three artifact-editing benchmarks. SR is the task success rate (%): final validator PASS with total inference cost within the reporting budget. #Fail is the mean number of failed blocking checks at the evaluated checkpoint, and Judge is the mean overall visual-judge score (1–5) on the evaluated subset. Higher SR and Judge and lower #Fail are better. N denotes the full benchmark size; judge scores use only evaluated tasks. Base is evaluated only with Claude. The best observed value within each model, benchmark, and metric is bolded. See the appendix for reporting budgets, evaluated-subset sizes, confidence intervals, and full results.

	Poster (N=50)			SVG (N=30)			Blender (N=22)		
Method	SR(%)↑	#Fail↓	Judge↑	SR(%)↑	#Fail↓	Judge↑	SR(%)↑	#Fail↓	Judge↑
`claude-opus-4-8`									
Base	18	4.3	3.08	36.7	5.87	2.6	31.8	4.73	3.45
+Val	56	1.18	3.04	70	2.47	2.77	59.1	0.82	**3.91**
AM (ours)	**82**	**0.58**	**3.17**	**90**	**2.10**	**2.9**	**90.9**	**0.27**	3.86
`gpt-5.6-terra`									
+Val	66	1.08	2.57	80	2.97	2.07	72.7	1.73	3.7
AM (ours)	**80**	**0.48**	**2.71**	**90**	**0.93**	**2.2**	**86.4**	**1.09**	**3.75**

Table 7: Efficiency on jointly solved tasks under the separate efficiency evaluation protocol. We report mean model-inference cost (USD) and model turns to the first full validator PASS. Within each model–benchmark pair, both methods are evaluated on the same subset of tasks that both solve under this protocol. N denotes the full benchmark size, not the size of the jointly solved subset. Lower is better; the lowest mean within each model, benchmark, and metric is bolded. Additional results are provided in the appendix.

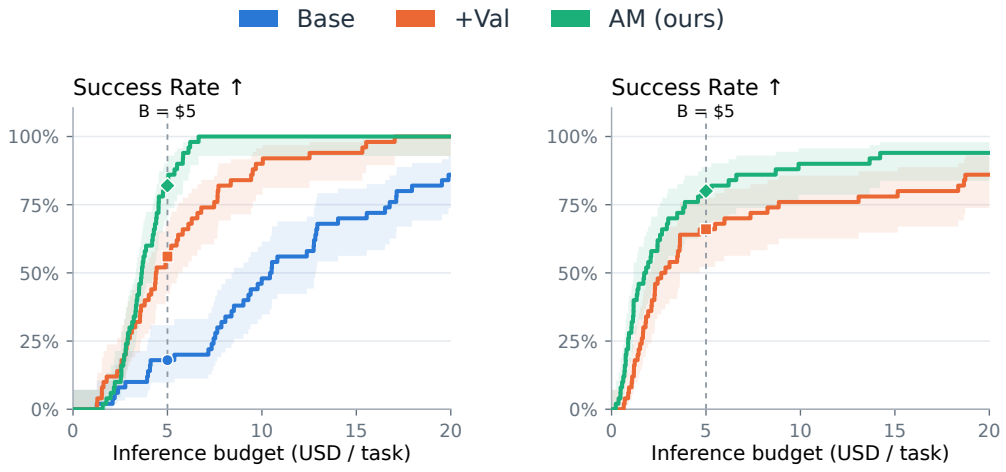
	Poster (N=50)		SVG (N=30)		Blender (N=22)	
Method	Turns↓	Cost($)↓	Turns↓	Cost($)↓	Turns↓	Cost($)↓
`claude-opus-4-8`						
+Val	81.4	5.56	26.1	3.12	30.1	3.96
AM (ours)	**47.8**	**3.81**	**15.5**	**2.32**	**20.3**	**2.51**
`gpt-5.6-terra`						
+Val	58.4	6.14	16.7	2.27	31.3	2.2
AM (ours)	**39.2**	**4.57**	**12.6**	**1.65**	**23.9**	**1.34**



(a) Claude Opus 4.8

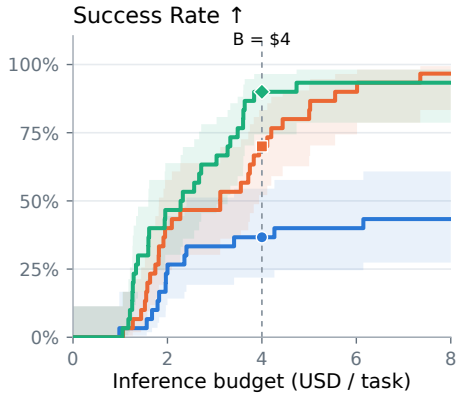
(b) GPT-5.6 Terra

Figure 9: Success at fixed per-task inference-cost budgets.

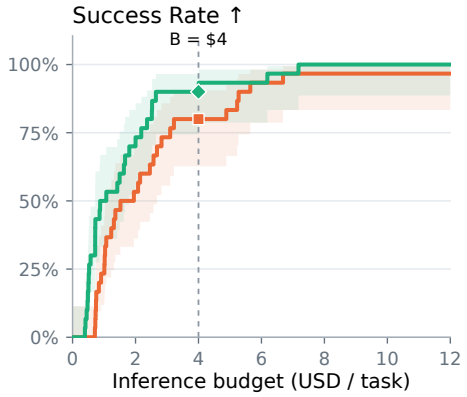


(a) Poster — Claude Opus 4.8

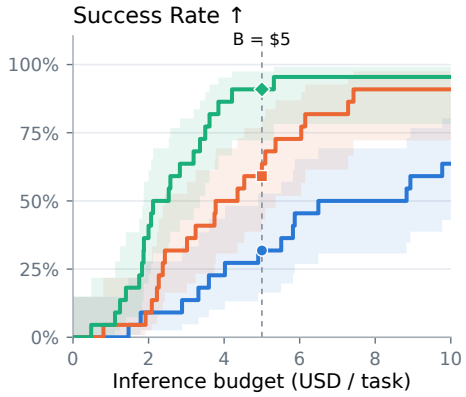
(b) Poster — GPT-5.6 Terra



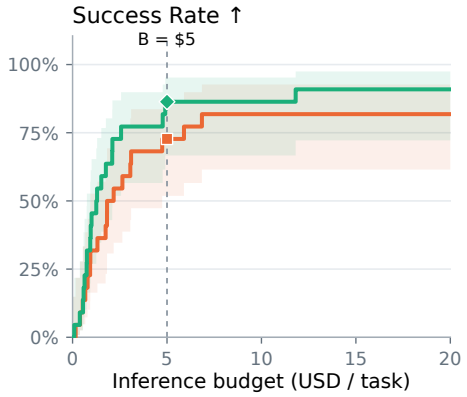
(c) SVG — Claude Opus 4.8



(d) SVG — GPT-5.6 Terra

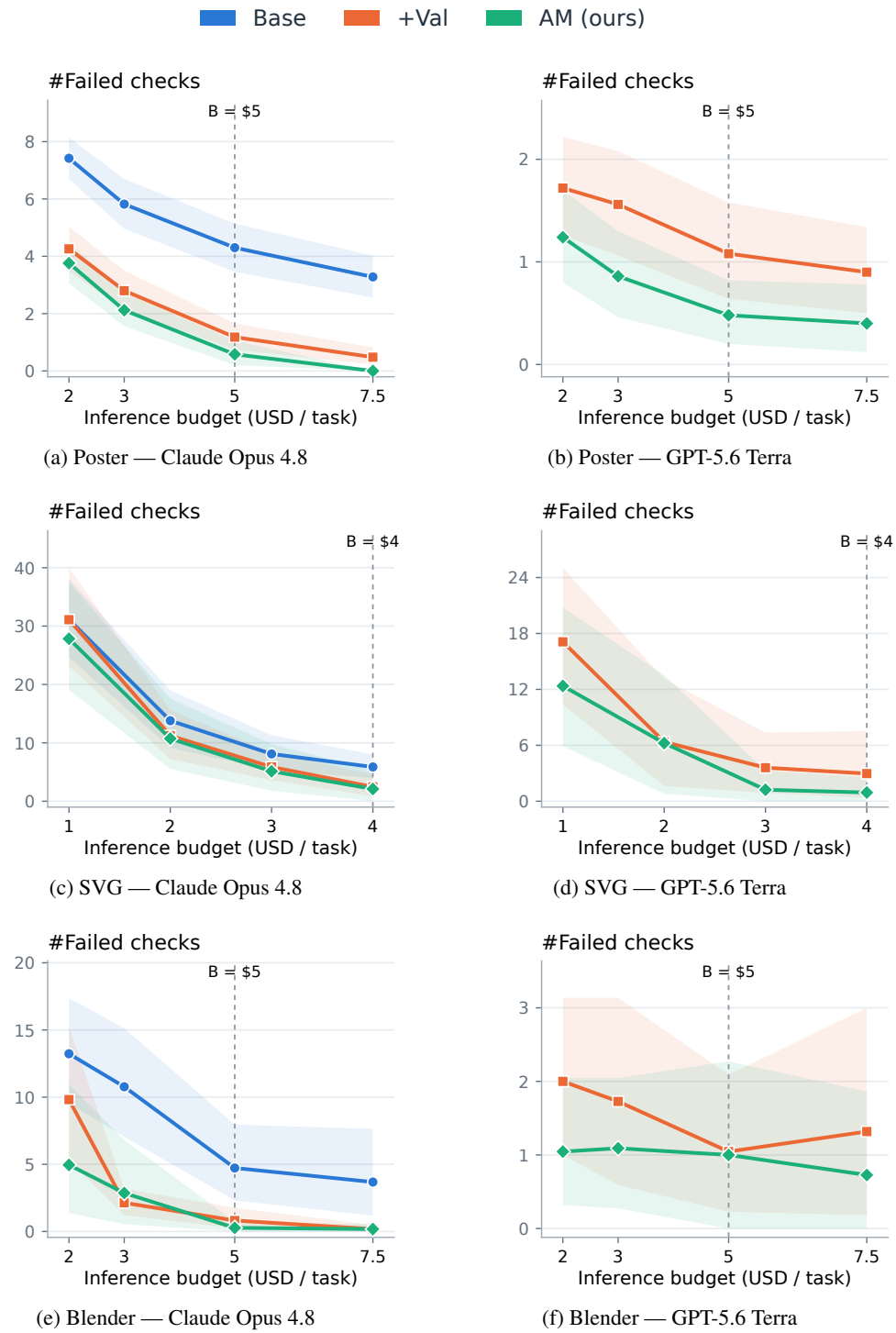


(e) Blender — Claude Opus 4.8



(f) Blender — GPT-5.6 Terra

Figure 10: Success versus inference-cost budget for both models on all three benchmarks. The numerator counts final validator PASS with total run cost at most the budget; the denominator includes all tasks, including failures and runs beyond the displayed axis. Bands are pointwise 95% Wilson intervals. Dashed lines mark the fixed budgets. These are retrospective total-cost cutoffs, not reconstructed first-PASS costs. Base is evaluated only with Claude.



(a) Poster — Claude Opus 4.8

(b) Poster — GPT-5.6 Terra

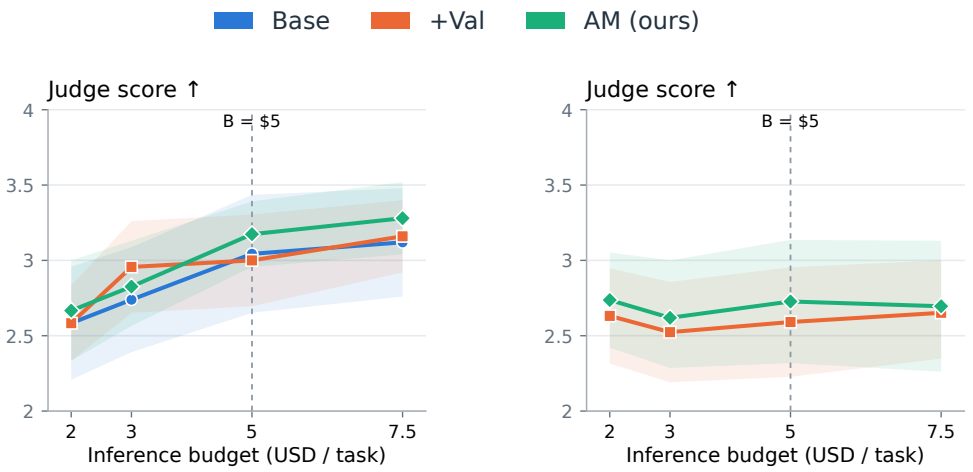
(c) SVG — Claude Opus 4.8

(d) SVG — GPT-5.6 Terra

(e) Blender — Claude Opus 4.8

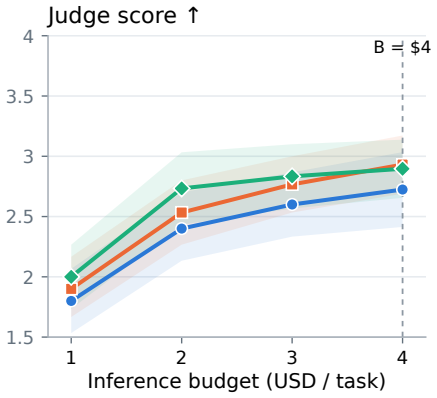
(f) Blender — GPT-5.6 Terra

Figure 11: Mean failed blocking checks at four inference-cost budgets; lower is better.



(a) Poster — Claude Opus 4.8

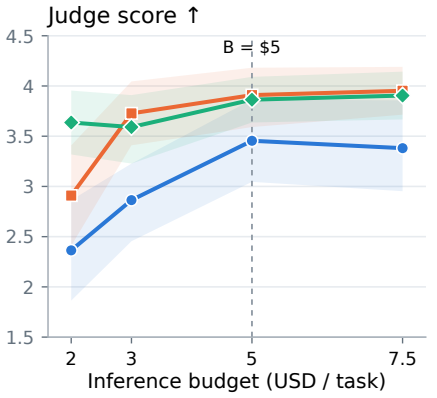
(b) Poster — GPT-5.6 Terra



(c) SVG — Claude Opus 4.8



(d) SVG — GPT-5.6 Terra

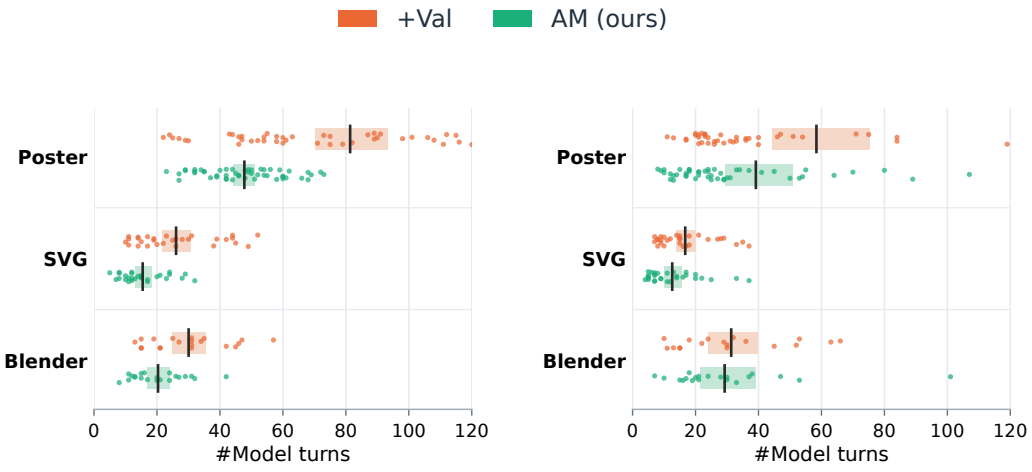


(e) Blender — Claude Opus 4.8



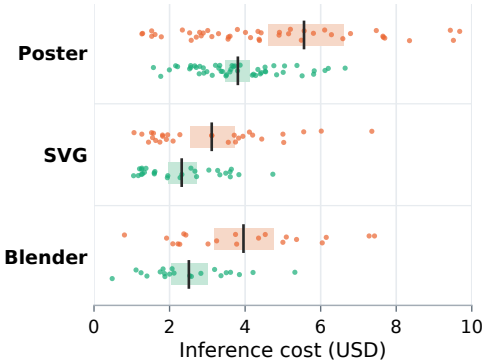
(f) Blender — GPT-5.6 Terra

Figure 12: Visual-judge overall scores (1–5; higher is better) on the evaluated subset. At each model, benchmark, and budget, all methods use the same tasks with available judgments.

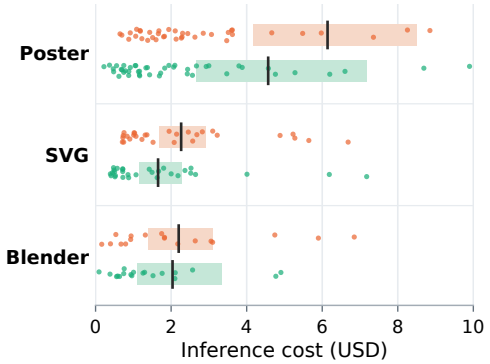


(a) Model turns — Claude Opus 4.8

(b) Model turns — GPT-5.6 Terra



(c) Inference cost — Claude Opus 4.8



(d) Inference cost — GPT-5.6 Terra

Figure 13: Total model turns and inference cost among each method's own successful runs. Dots denote runs, black bars show means, and borderless bands show 95% task-bootstrap intervals (4,000 resamples, seed 0). Each metric is cropped only for display: 18 points above 120 turns and 20 points above $10 are omitted across both models. All successful runs remain in the means and intervals.