FINALE: RECONSTRUCTING LOCAL SUPERVISION WITH TERMINAL-ANCHORED REPLAY FOR END-TO-END WEB AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

End-to-end Web development requires implementing and integrating interdependent features. Complete teacher trajectories demonstrate both local implementation and the overall development process. We investigate whether more explicit local supervision from existing teacher demonstrations can improve end-to-end performance beyond complete-trajectory training alone. Identifying these local tasks is difficult because repeated file revisions leave multiple candidate endpoints, while local objectives and action boundaries remain unlabeled. We introduce FINALE, which reconstructs these tasks using retained file states and successful builds as observable endpoint anchors. It comprises two reconstruction workflows: ArtifactReplay groups file histories contributing to shared functionality retained in the delivered project, while BuildReplay extracts edit phases ending in successful builds. For each unit, FINALE recovers the repository prestate, synthesizes a local query from recorded target evidence, and uses endpoint-supported teacher actions as supervision for the model. We mix these units with complete trajectories for training. Across the Web and Game sets of ArtifactsBench and Cookie-Bench, FINALE improves Qwen3.5-35B-A3B-Base to an average score of 47.29, surpassing its official post-trained counterpart by 17.2%. Applied to Qwen3.6-27B, it raises the score from 40.73 to 62.22, exceeding DeepSeek-V4-Pro (59.05). In controlled comparisons, FINALE outperforms complete-trajectory training by 4.8 points under the same two-epoch schedule and by 17.2 points at comparable processed-token budgets, while surpassing intermediate replay and same-target reweighting.

1 INTRODUCTION

Developing an interactive Web application from a single user request requires an agent to implement and integrate interdependent features (Wang et al., 2024; Zhang et al., 2024). Complete teacher trajectories demonstrate this process at two levels: the overall development workflow and the local implementation work that realizes individual features. Training on these trajectories provides supervision for both, while the immediate objectives of local implementation often remain implicit in the application-level request and interaction history. We investigate whether more explicit supervision of local implementation capabilities can improve end-to-end development beyond complete-trajectory training alone.

Consider the game about a little bird exploring an autumn park in Figure 1. Building it involves rendering the environment, implementing movement and flight, making the camera follow the bird, and adding environmental interactions. Implementing an interaction with a leaf pile, for example, requires checking proximity, recording the interaction, updating exploration progress, and triggering visual and audio feedback. The development work needed to realize this objective connects game logic, interface updates, and feedback mechanisms, illustrating the coordinated implementation work that local supervision could target.

However, extracting such local tasks from complete trajectories is not straightforward. In our analysis of 2,000 Web-development trajectories, 55.30% modify at least one previously written file again, leaving multiple versions of the same file along the development process. These versions offer several possible stopping points, but the trajectories provide no explicit labels for local objectives or action

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.

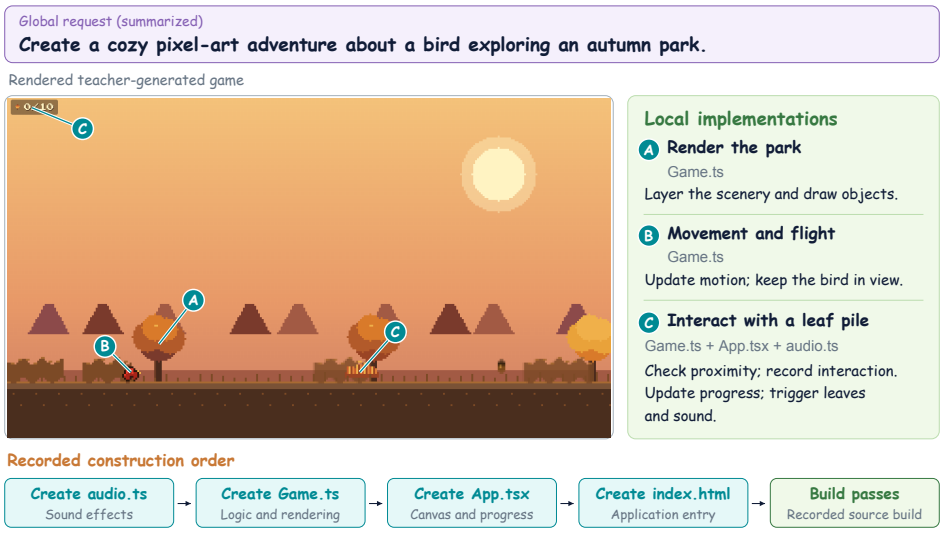


Figure 1: **One global request, several local implementations.** A rendered teacher-generated game. Callouts A–C link the scene, bird, and leaf pile to their implementation files.

boundaries to guide the choice. Without an observable anchor, post hoc segmentation therefore makes it difficult to justify where a local training task should end.

Our central observation is that task-completing agent trajectories expose two observable result endpoints that can anchor the selection of target action sequences. A *file terminal* occurs when a file reaches the state retained in the delivered project and is not changed again. A *build terminal* is the repository snapshot after a phase of file-changing actions reaches a successful build. We call these *result terminals*, providing reproducible evidence that the trajectory reaches a retained artifact state or a buildable project state.

Turning these endpoints into self-contained training examples poses two challenges. First, an endpoint supplies only a right-boundary anchor; recovering the corresponding start still requires a workflow-specific rule. Starting too late omits required edits; starting too early mixes unrelated changes. Second, an action interval alone is not a self-contained training example: it lacks both the project state before the first action and a local user query specifying what the interval should accomplish. Without these conditions, the model may misjudge whether the edits fit the current project, producing hallucinated changes that conflict with the actual repository state (Zhang et al., 2025b).

We introduce FINALE, which uses two reconstruction workflows to build terminal-anchored local supervision. *ArtifactReplay* reconstructs implementation units anchored by file content retained at delivery, and *BuildReplay* reconstructs repository-level edit phases anchored by successful builds. To address the first challenge, *ArtifactReplay* traces each retained file back to its first effective modification and groups file histories that contribute to a shared functionality. *BuildReplay* uses the first failed build or the previous successful build as the phase's left boundary. To address the second challenge, FINALE recovers the actual project state before each interval by undoing recorded file-changing actions in reverse order, then provides the recovered state as repository context for the local task. An auxiliary model then synthesizes a local query using the global request, recovered prestate, and target evidence. Finally, each local unit pairs the synthesized query with the recovered prestate and selected teacher action sequence, supervising endpoint-supported actions while retaining the remaining history as context. During training, local units are mixed with complete trajectories to retain supervision for the overall development process.

We evaluate FINALE across the Game and Web subsets of ArtifactsBench and Cookie-Bench using two starting backbones. Starting from Qwen3.5-35B-A3B-Base, FINALE attains an average Total score of 47.29, surpassing its official post-trained counterpart by 17.2%. When applied to Qwen3.6-27B, FINALE raises the average Total score to 62.22, outperforming DeepSeek-V4-Pro (59.05). Further controlled experiments compare the replay workflows, terminal selection, and local conditioning. FINALE improves the average Total score over complete-trajectory training by 4.8 points under

the same two-epoch schedule and by 17.2 points at comparable processed-token budgets. It also outperforms matched intermediate replay and same-target reweighting, supporting the use of terminal anchors and local task conditioning. We will release the trained models and accompanying local-supervision dataset to support reproducibility and further research on Web-development agents.

2 Related Work

Supervision from agent trajectories. CodeAct represents agent actions as executable code within multi-turn interactions, while AgentOhana unifies diverse trajectories into a standardized training format (Wang et al., 2024; Zhang et al., 2024). Although training on these complete trajectories demonstrates end-to-end task execution, immediate implementation objectives remain implicit across long-context histories. Rather than training strictly end-to-end, FINALE extracts local implementation tasks from recorded trajectories and mixes them with complete trajectories to balance high-level process coordination with fine-grained implementation supervision.

Trajectory relabeling and selective supervision. Selective reuse and relabeling have been explored in prior trajectory-based learning. Hindsight Experience Replay (Andrychowicz et al., 2017) re-indexes transitions using achieved outcomes, while recent agent frameworks extract successful sub-trajectories (Li et al., 2022), annotate subgoals (Hashemzadeh et al., 2025), or select critical steps with an LLM and mask loss on other steps (Chen et al., 2025). Similarly, HSL (Li et al., 2026) prompts an LLM to identify achieved goals and mask irrelevant steps in existing trajectories. FINALE differs fundamentally in how task boundaries and contexts are grounded: rather than relying on heuristic or LLM-inferred action boundaries, we ground endpoints in file state retained at delivery and recorded successful build outcomes. Moreover, FINALE rolls back the exact repository prestate to supply context, decoupling semantic query generation from deterministic state-transition verification.

3 Problem Formulation

We formulate the task of mapping a successful application-level agent trajectory into terminal-anchored local units that provide targeted supervision alongside complete trajectories. At the event level, an input trajectory is represented as

$$\tau = (q^G, S_T, (e_t)_{t=1}^T), \tag{1}$$

where q^G denotes the global task instruction, S_T is the delivered repository state after the final event e_T, and each event $e_t = (c_t, o_t)$ pairs a tool call c_t with its corresponding execution observation o_t in chronological order.

From τ, FINALE constructs a terminal-anchored local unit of the form

$$u = (q_u, P_u, Y_u, m_u), \tag{2}$$

where $Y_u = (e_b, \ldots, e_e)$ is a contiguous span of source events indexed by $1 \le b \le e \le T$, ending at an observable result recorded in the trajectory. q_u is a natural language query describing the immediate objective of Y_u, P_u serializes the repository prestate S_{b-1} preceding event e_b as file paths and contents, and m_u selects the tool calls in Y_u that receive supervision. Teacher actions in Y_u are reused from the source trajectory.

Training objective. Training mixes local replay units with complete trajectories. Let C denote the serialized conditioning prefix and X the tokenized interaction sequence. For a local unit, C serializes (q_u, P_u) and X serializes Y_u; tokens in tool calls selected by m_u receive supervision, while all other tokens have zero loss. For a complete trajectory, C serializes q^G and X serializes the full interaction history with its original source loss mask. Writing the token-level mask as $\bar{m}_r \in \{0, 1\}$, both types use the same objective:

$$\ell(\theta; C, X) = -\sum_{r=1}^{|X|} \bar{m}_r \log p_\theta(X_r \mid C, X_{<r}). \tag{3}$$

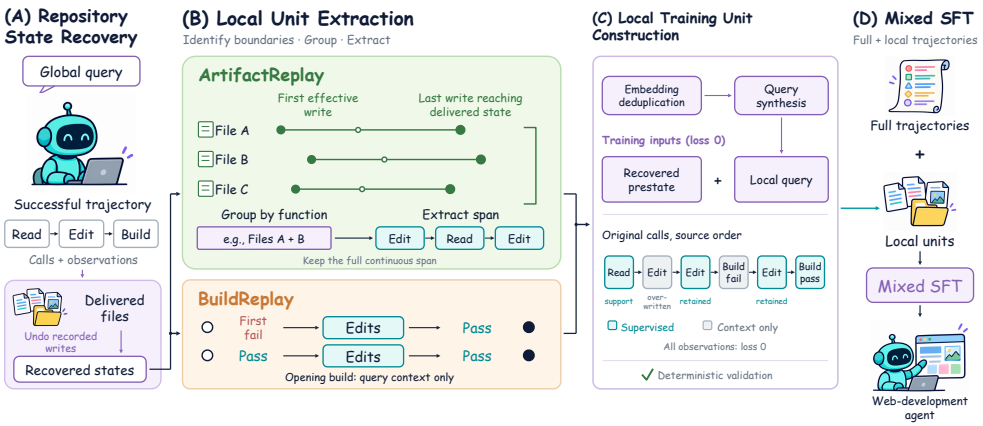


Figure 2: Overview of FINALE.

4 FINALE

FINALE constructs terminal-anchored local training units from complete, successful trajectories. ArtifactReplay uses delivered file contents as endpoints, while BuildReplay uses repository states after successful builds. Within each extracted span, both workflows retain the recorded events and assign supervision to terminal-supported writes and their supporting actions.

Figure 2 groups construction into three parts. *(A) Repository State Recovery* reconstructs repository states from recorded observations and file modifications; *(B) Local Unit Extraction* selects endpoints and start boundaries, then forms continuous source spans with call-level supervision masks. (C) *Local Training Unit Construction* renders the prestate, filters near-duplicates, synthesizes a local query and validates the unit against deterministic gates.

4.1 REPOSITORY STATE RECOVERY

For an extracted replay sequence $Y_u = (e_b, \ldots, e_e)$, we reconstruct its start state S_{b-1} by inverting recorded writes backward from the known terminal files. A unit is retained when every write can be inverted unambiguously and forward replay reproduces the terminal state. To recover this start state, Recover combines authoritative state observations, such as `list_files` and full-file `read_file` results, with deterministic state-changing calls, such as `create_file` and `edit_file`, to reconstruct the state trace.

4.2 LOCAL UNIT EXTRACTION

ArtifactReplay. To identify the start boundary and endpoint for each candidate delivered file f, let $F_f = S_T[f]$ denote its content at delivery, and let S_t denote the recovered repository state after event e_t. ArtifactReplay defines:

$$\begin{aligned} &\mathcal{W}_f = \{t : c_t \text{ writes } f,\ S_t[f] \neq S_{t-1}[f]\}, \\ &b_f = \min \mathcal{W}_f, \qquad d_f = \max\{t \in \mathcal{W}_f : S_t[f] = F_f\}, \qquad I_f = \{b_f, \ldots, d_f\}. \end{aligned} \tag{4}$$

Here, $\mathcal{W}_f$ collects the event indices of effective writes to f, b_f marks the first effective write, and d_f marks the final write that reaches terminal content F_f. I_f spans the trajectory from the initial state-changing modification to the final terminal-converging write.

Although the interval I_f bounds the file's edit history, individual writes within it may still include temporary edits or formatting changes later superseded. To select write calls containing changes retained at delivery, we use a deterministic alignment check, TERMINALMATCH. For each write event $t \in I_f$, TERMINALMATCH compares the pre- and post-write states ($S_{t-1}[f]$ and $S_t[f]$) against the delivered content F_f, setting $z_t^f = 1$ when an added token persists at a uniquely aligned position

in F_f or a deleted token remains absent. The tool-call supervision mask is

$$z_t^f = \mathbf{1}[t \in \mathcal{W}_f \wedge \mathrm{TerminalMatch}(S_{t-1}[f], S_t[f], F_f)]\,,$$
$$m_t^f = \mathbf{1}\Big[z_t^f = 1 \vee \mathrm{Support}_t^f \vee \mathrm{PassBuild}_t\Big]\,. \tag{5}$$

$\mathrm{Support}_t^f$ selects prior Read or Locate calls that directly inform a positive write by retrieving its target path or required code context. $\mathrm{PassBuild}_t$ denotes a passing build within I_f. All observations o_t, write calls without qualifying terminal-retained changes carry zero loss.

Single-file intervals can be semantically incomplete when a feature spans interdependent files. Given the delivered application, we first identify the set of eligible candidate files $\mathcal{E}_A = \{f : \exists t \in I_f,\ z_t^f = 1\}$ that contain terminal-supported edits. Based on each file's terminal content and modification history, an auxiliary model, ***GroupLM***, proposes functional groups of candidate files that jointly implement a shared objective. We then normalize these proposals by removing duplicate groups and strict supersets of narrower groups, rejecting any grouping that leaves an eligible file uncovered. The accepted group family $\mathcal{G} = \{G_k\}$ satisfies

$$\forall k,\ G_k \subseteq \mathcal{E}_A, \qquad \bigcup\nolimits_k G_k = \mathcal{E}_A, \qquad \forall j \neq k, \quad G_j \not\subseteq G_k \ \wedge\ G_k \not\subseteq G_j. \tag{6}$$

These conditions enforce that every group contains only eligible files, all eligible files are covered, and no surviving group is a strict subset of another. Partial overlaps remain valid; for example, a shared type-definition file may support both a backend handler group and a frontend component group.

To construct the replay sequence while preserving event order across multiple files, ArtifactReplay spans the full timeline from the earliest to latest edit within group G, and extends the boundaries to ensure that all tool calls, observations, and messages remain fully paired:

$$U_G = \bigcup\nolimits_{f \in G} I_f, \qquad K_G^{(0)} = [\min U_G, \max U_G],$$
$$K_G^\star = \mathrm{Close}(K_G^{(0)}), \qquad Y_G = (e_t)_{t \in K_G^\star},$$
$$m_t^G = \begin{cases} \max_{f \in G:\, t \in I_f} m_t^f, & t \in U_G, \\ 0, & t \in K_G^\star \setminus U_G. \end{cases} \tag{7}$$

Here, $K_G^\star$ minimally extends the span to preserve complete tool-call observation pairs and messages. Within U_G, the resulting sequence Y_G inherits the file-level supervision masks, including those for terminal-supported writes, selected Read or Locate calls, and passing builds. Events added only by the continuous hull or closure, as well as writes to files outside G, receive zero loss.

BuildReplay. BuildReplay designates the repository state S_{β_b} of a passing build e_{β_b} as the local terminal state, with a preceding qualifying build e_{β_a} defining the left boundary. It extracts the intervening edit phase $(e_t)_{t \in (\beta_a, \beta_b]}$, where β_i denotes the source-event index of the ith build.

Within a trajectory's build sequence, let $\beta_1 < \cdots < \beta_M$ denote the build-event indices with outcomes $r_i \in \{\mathrm{Fail}, \mathrm{Pass}\}$. BuildReplay extracts candidate intervals $I_B = (\beta_a, \beta_b]$ of two types: (1) *fail-to-pass* (FP) spanning from the first failure in a failing run to the next passing build; (2) *pass-to-pass* (PP) spanning consecutive passing builds with at least one intervening file write.

Both interval types are left-open and right-closed. The opening build reflects the unit's starting condition; it is omitted from the replay sequence Y_u and serves solely as context for local query generation. The closing passing build and its observation are retained as the verified endpoint.

At endpoint β_b, each edited file has local terminal content $F_{f,b} = S_{\beta_b}[f]$. BuildReplay applies the same TERMINALMATCH alignment and support criteria against $F_{f,b}$, setting $m_t^B = 1$ only for endpoint-supported writes, their supporting Read or Locate calls, and the final passing build. All other events, including intermediate failed builds and unaligned writes, have $m_t^B = 0$.

4.3 LOCAL TRAINING UNIT CONSTRUCTION

For each extracted event span Y_u, ***Recontextualize*** renders the reconstructed prestate P_u as synthetic directory listings and full-file read calls prepended to Y_u, providing the repository context immediately preceding event e_b. These synthetic prestate interactions carry zero loss.

Across distinct source trajectories, repeated boilerplate edits can introduce redundant supervision (for example, identical TypeScript ambient declarations for static asset imports that differ only in quote style or file extension). Before query generation, we represent the loss-bearing code modifications of each candidate unit using Qwen3-Embedding-8B, and greedily remove near-duplicates whose cosine similarity exceeds a predefined threshold, which we set to 0.9.

Next, an auxiliary model, *QueryLM*, synthesizes the local query q_u as a concise, coherent local objective. For both ArtifactReplay and BuildReplay units, QueryLM is conditioned on the target file paths, the global user instruction q^G, the rendered prestate P_u, and the target messages.

Finally, *Finalize* audits candidate units through deterministic validation checks. We retain a unit only when its prestate is recovered unambiguously, replaying the recorded file mutations from that state reproduces the selected endpoint, source messages and tool-call–observation pairs remain complete, and at least one endpoint-supported write receives supervision. Accepted units instantiate the unified interface $u = (q_u, P_u, Y_u, m_u)$ defined in Section 3.

5 Experimental Setup

Benchmarks and Metrics. We evaluate web generation on ArtifactsBench and Cookie-Bench.

From ArtifactsBench (Zhang et al., 2025a), we select the Game Development (413 tasks) and Web Applications (438 tasks) subsets. A multimodal judge evaluates the request, generated code, and browser screenshots across ten visual and code-related dimensions. We report Total averaged over all tasks in each scenario (evaluator details in Appendix F.7).

From Cookie-Bench (Yang et al., 2026), we select 156 Mini-game pages as Game and 114 Interactive application pages as Web using the released category labels. We use Cookie's static evaluator and report Functionality, Aesthetics, and Total, which combines the two dimensions, on a 0–100 scale. Benchmark details and score computation are provided in Appendix F.7.

Evaluation. We evaluate overall performance on all tasks in the selected Game and Web subsets of both benchmarks using Claude-Sonnet-4.6 as the judge for the main result (Table 1), motivated by its strong performance in visual aesthetic judgment and interaction-grounded UX assessment (VAB Team, 2026; Wang et al., 2026). For component ablations and controlled comparisons, we use a shared, frozen subset of 200 queries (50 per benchmark scenario; additional subsets with multiple seeds in Appendix F.4) evaluated with Qwen2.5-VL-72B-Instruct to manage evaluation costs. This open-weight model is commonly used as a visual critic in web generation (Zhang et al., 2025a; Li et al., 2025). All evaluations score single-attempt generation ($K = 1$).

Baselines. We train FINALE from two starting checkpoints: Qwen3.5-35B-A3B-Base (Qwen Team, 2026a) and Qwen3.6-27B (Qwen Team, 2026d). Same-backbone baselines include both original checkpoints and the official post-trained Qwen3.5-35B-A3B (Qwen Team, 2026b). We further compare against state-of-the-art open-weight models across two groups: comparable-scale general code and Web-specialized models, including Qwen3-Coder-30B-A3B-Instruct (Qwen Team, 2025), FrontCoder-30B-A3B (Feng et al., 2026), and FullStack-Learn-LM-30B-A3B (Lu et al., 2026); and models with over 100B total parameters, including Qwen3.5-122B-A10B (Qwen Team, 2026c), DeepSeek-V4-Flash, and DeepSeek-V4-Pro (DeepSeek-AI, 2026).

Training data. Our source trajectories for web applications and games are generated by Claude Opus 4.7 (Anthropic, 2026) using an in-house agent scaffold. Given a user request, the teacher agent starts from a clean React template and implements the application through file-reading, file-editing, and build commands. Appendix F.1 summarizes the construction inputs and points to the auxiliary prompts. We employ DeepSeek-V4-Flash for GroupLM and QueryLM. Our final dataset comprises 39,430 successful complete trajectories, 82,713 ArtifactReplay units, and 5,416 BuildReplay units.

Training configuration. We train both FINALE variants on the same mixed dataset of complete trajectories and local replay units for two epochs (682 optimization steps), using identical hyperparameters. Training uses SWIFT with BF16 precision, sequence packing, a maximum context length of 131,072 tokens, and a global batch size of 48 (more details in Appendix F.2).

Model	Size	ArtifactsBench		Cookie-Bench						Avg.
		Game	Web	Game			Web			
		Total	Total	Func.	Aes.	Total	Func.	Aes.	Total	
DeepSeek-V4-Pro	1.6T/49B	70.69	**54.36**	**79.50**	**62.00**	**59.33**	71.50	59.00	51.83	**59.05**
DeepSeek-V4-Flash	284B/13B	**71.91**	49.04	46.00	37.00	26.50	73.00	62.00	54.33	50.45
Qwen3.5-122B-A10B	122B/10B	53.26	38.98	71.00	57.00	50.67	**80.50**	**62.50**	**60.50**	50.85
Qwen3-Coder-30B-A3B	30B/3B	39.10	37.17	54.50	41.00	33.00	66.00	42.50	40.67	37.49
FrontCoder-30B-A3B	30B/3B	48.63	40.65	58.50	58.50	42.50	60.00	53.00	41.33	43.28
FullStack-Learn-LM-30B-A3B	30B/3B	28.55	25.38	36.55	28.08	23.31	42.01	29.25	28.07	26.33
Qwen3.5-35B-A3B-Base	35B/3B	39.22	27.32	42.50	40.00	25.67	54.00	44.50	34.00	31.55
Qwen3.5-35B-A3B	35B/3B	44.58	36.71	54.50	46.50	34.83	67.00	51.50	45.33	40.36
FINALE (Qwen3.5-35B-A3B)	35B/3B	**50.38**	**43.44**	**59.50**	**61.00**	**44.17**	**68.50**	**61.50**	**51.17**	**47.29**
Δ over Base		+11.16	+16.12			+18.50			+17.17	+15.74
Qwen3.6-27B	27B	44.19	39.39	67.00	50.50	44.83	56.50	42.00	34.50	40.73
FINALE (Qwen3.6-27B)	27B	**63.23**	**51.26**	**80.50**	**67.14**	**62.86**	**87.62**	**72.48**	**71.54**	**62.22**
Δ over Qwen3.6-27B		+19.04	+11.87			+18.03			+37.04	+21.49

Table 1: Overall performance on ArtifactsBench and Cookie-Bench.

6 RESULTS AND ANALYSIS

6.1 OVERALL PERFORMANCE

Table 1 compares FINALE with same-backbone, general-purpose and code, Web-specialized, and larger open-weight models on the selected Game and Web subsets of ArtifactsBench and Cookie-Bench. FINALE improves end-to-end performance on both backbones, with higher Total scores in all four benchmark scenarios. Across the four benchmark scenarios, the FINALE variant trained from Qwen3.5-35B-A3B-Base achieves an average Total score of 47.29, surpassing the official post-trained Qwen3.5-35B-A3B (40.36) by 6.93 points (17.2%). Applying the same training recipe to Qwen3.6-27B raises its average score from 40.73 to 62.22 (+21.49 points; 52.8%). The resulting 27B model even surpasses DeepSeek-V4-Pro in average Total score (62.22 vs. 59.05).

6.2 COMPONENT ABLATION

To examine the effectiveness of our data recipe, all component ablations start from Qwen3.5-35B-A3B-Base across five configurations evaluated on the shared 200-query benchmark subset (Table 2): (1) **Full-only** trains exclusively on complete successful trajectories for two epochs (283 steps); (2) **Full-only**† trains on the same complete trajectories while matching FINALE's 682 optimization steps; (3) **Full+A** augments complete trajectories with ArtifactReplay units (640 steps); (4) **Full+B** augments them with BuildReplay units (325 steps); and (5) **FINALE (Full+A+B)** combines both sources (682 steps). These trends remain consistent across multiple additional 200-task resamplings (Appendix F.4).

Local supervision improves on complete-trajectory training. FINALE outperforms the two-epoch Full-only baseline (72.7 vs. 67.9 Avg.) and the 682-step Full-only† control (55.5). In the latter comparison, processed-token budgets are close (4.247B vs. 4.222B), while positive-supervision tokens differ (1.934B vs. 3.023B). These results favor the mixed-training recipe under both comparisons; Appendix F.2 details the aggregate training budgets. One possible explanation is that local supervision makes the connection between a requested change and its implementing actions explicit, helping the model learn when and how to apply useful edits that transfer to end-to-end generation.

FINALE exhibits broader file coverage with fewer steps and repeated writes. We further compare the inference behavior of Base, Full-only, and FINALE on the same 200 evaluation queries (Figure 3). FINALE averages 18.60 agent steps (decision rounds) per query, reducing step count by 15.9% relative to Full-only (22.11) and 19.5% relative to Base (23.11; panel (a)). Meanwhile, the number of distinct files touched per query increases from 5.61 (Full-only) to 6.37 (panel (b)),

Training configuration	ArtifactsBench		Cookie-Bench						Avg.
	Game	Web	Game			Web			
	Total	Total	Func.	Aes.	Total	Func.	Aes.	Total	
Reference configurations									
Qwen3.5-35B-A3B-Base	57.7	33.4	79.8	78.7	76.4	59.3	58.4	56.0	55.9
Qwen3.5-35B-A3B	61.5	37.3	83.3	81.5	80.1	68.1	67.1	65.1	61.0
Component ablations and combined model									
Full-only	68.6	54.7	81.9	79.7	79.2	70.6	69.8	69.3	67.9
Full-only†	58.1	44.8	54.7	54.0	53.4	67.8	66.7	65.8	55.5
Full+A	66.8	51.6	76.4	75.6	74.8	65.1	64.6	62.7	64.0
Full+B	69.1	57.0	**84.1**	**83.6**	**82.5**	69.1	68.9	67.2	69.0
FINALE (Full+A+B)	**78.0**	**61.2**	80.0	79.4	77.5	**76.6**	**75.0**	**73.9**	**72.7**
Mechanism controls									
Matched Intermediate Replay	61.7	56.2	57.3	55.7	55.8	51.2	51.7	49.9	55.9
Same-Target Reweighting	61.8	54.0	72.0	70.1	69.5	70.8	70.0	68.4	63.4

Full-only trains for two epochs; † matches FINALE's optimization steps.

Table 2: Controlled comparisons. Avg. is the unweighted mean of the four scenario Total scores.

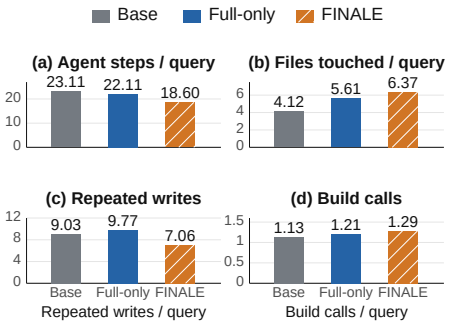


Figure 3: Interaction and tool-use behavior across Base, Full-only, and FINALE.

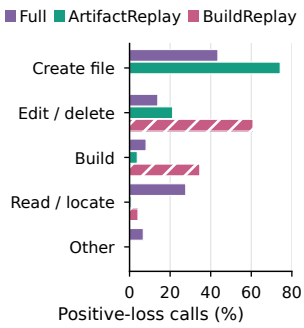


Figure 4: Distribution of positive-loss tool calls.

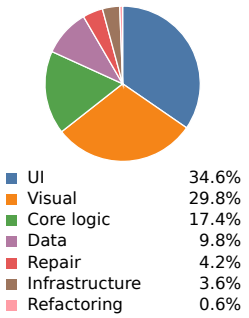


Figure 5: Semantic analysis of targets.

counting unique paths across create, edit, and delete actions. Despite this broader scope, repeated writes beyond the first write to a file drop from 9.77 to 7.06 per query (panel (c)), while build frequency remains essentially unchanged (1.21 vs. 1.29 calls per query; panel (d)).

The combined recipe performs best among the component variants. Combining ArtifactReplay and BuildReplay yields the highest Avg. score of 72.7, compared with 64.0 for Full+A and 69.0 for Full+B. Their supervision nevertheless has different emphases: ArtifactReplay guides the cold-start creation of standalone components and assets, whereas BuildReplay trains the model to iteratively modify existing code guided by build feedback. Figure 4 shows these different emphases: file creation accounts for 74.2% of ArtifactReplay's positive-loss tool calls, whereas editing/deletion and building account for 60.8% and 34.6% of BuildReplay's calls, respectively.

6.3 DISCUSSION

We investigate two central questions in our supervision design: *which* actions to supervise (target selection) and *how* to present them (local conditioning), and conclude with a case study.

Terminal-anchored supervision outperforms matched intermediate trajectory replay. To assess whether supervision should prioritize terminal outcomes over intermediate trajectory states, we compare FINALE against Matched Intermediate Replay, which constructs local training units from nonterminal action windows in the source trajectories. Both configurations train on the same source-trajectory pool for 682 steps under matched target-length distributions (Appendix Table 4), while FINALE selectively supervises actions grounded in delivered files or passing-build endpoints.

As shown in Table 2, FINALE outperforms Intermediate Replay (72.7 vs. 55.9 Avg.). One hypothesis is that nonterminal windows omit edits needed to complete a local modification, and such incomplete supervision may make it harder for the trained model to produce buildable projects. Consistent with this hypothesis, on the evaluation queries, FINALE achieves a higher recorded build-completion rate of 89.0% (178/200) compared to 74.0% (148/200) for Matched Intermediate Replay.

Local replay outperforms same-target reweighting. We further investigate whether observed gains can be obtained by upweighting selected actions within full trajectories. Same-Target Reweighting retains the original global prompt and execution history while assigning selected target tokens a loss weight of 2. It uses the same two-epoch schedule as Full-only, without adding local units. As reported in Table 2, this baseline achieves 63.4 Avg., below Full-only (67.9) and FINALE (72.7). Thus, doubling the selected targets' weights does not reproduce the gain in this setting. This highlights the advantage of FINALE's local task formulation: pairing terminal actions with a local request and repository prestate provides state-specific supervision that simple token reweighting cannot replicate.

Semantic diversity of FINALE supervision. To characterize the diversity of local supervision, we analyzed a uniform random sample of 500 source-distinct Terminal targets classified by DeepSeek V4 Flash using the taxonomy in Appendix H.3. As shown in Figure 5, the targets span diverse development tasks. UI and interaction (34.6%, 173/500), visual assets (29.8%, 149/500), and core logic (17.4%, 87/500) form the primary categories, complemented by data and content (9.8%, 49/500), repair and robustness (4.2%, 21/500), infrastructure and integration (3.6%, 18/500), and refactoring (0.6%, 3/500). Rather than teaching a single editing pattern, FINALE exposes the model to diverse, bounded subproblems encountered across application development.



Figure 6: Windows-like applications generated for the same task.

Case study of functional completeness. Beyond quantitative benchmarks, qualitative inspection across generation outputs indicates that FINALE can produce more functionally complete and interactive applications. For example, in `artifactsbench_270` (Figure 6), models trained with FINALE and with complete trajectories alone both produce buildable code for a Windows-style multi-window desktop. However, manual interaction reveals clear differences in execution completeness: FINALE provides working launch paths for bundled applications like Calculator and Notepad, enabling concurrent multi-window operations and interactive arithmetic. In contrast, while Full-only renders a visually plausible desktop and taskbar, its Start button cannot launch applications, leaving the core application suite inaccessible (event-handling diagnostic details in Appendix F.8). More examples are presented in Appendix G.

7 CONCLUSION

FINALE reconstructs grounded local SFT units from successful trajectories by anchoring to retained file contents (ArtifactReplay) and successful builds (BuildReplay). Mixed with complete trajectories, it boosts a 35B model to an average score of 47.29 across ArtifactsBench and Cookie-Bench—a 17.2% gain over its official post-trained baseline—and raises Qwen3.6-27B to 62.22, surpassing DeepSeek-V4-Pro. Controlled comparisons confirm that terminal-anchored local supervision consistently outperforms complete-trajectory baselines, intermediate replay, and target reweighting, demonstrating the value of high-density local supervision for end-to-end Web agents.

Reproducibility Statement

The experiments reported in this paper have been completed. Section 5 specifies the backbones, packing and evaluation tracks. Appendix F.2 reports the total optimization steps, aggregate processed and positive-supervision tokens, local-data shares, source coverage, and evaluated checkpoints, reconstructed from saved training arguments, cache audits, sampler schedules, and completion logs. Table 6 distinguishes construction, embedding deduplication, and final training-cache counts. We will release the trained models, local-supervision dataset, and accompanying manifests subject to source-trajectory licensing constraints.

Ethics Statement

FINALE trains on teacher-generated Web-development trajectories. Prior to release, we will verify source licenses and redistribution rights, filter sensitive or unsafe content, and outline intended uses in accompanying data and model cards; any material lacking redistribution clearance will be excluded. Importantly, our replay verification ensures file-state consistency rather than functional security. Downstream applications require independent security vetting before deployment, and these models must not be used to build deceptive or harmful software.

AI Use Statement

We used generative AI tools to generate teacher trajectories and construct local training data, and used vision-language models for benchmark evaluation, as described in the method and experimental sections. AI tools also assisted with code implementation and debugging and discussion of result interpretation. During manuscript preparation, we used these tools for text polishing, proofreading and LaTeX formatting. The authors made the final decisions on the methodology, experiments, and presentation, and take responsibility for the content of this paper.

References

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Anthropic. Introducing Claude Opus 4.7. Anthropic, April 2026. URL https://www.anthropic.com/news/claude-opus-4-7.

Zhixun Chen, Ming Li, Yuxuan Huang, Yali Du, Meng Fang, and Tianyi Zhou. ATLAS: Agent tuning via learning critical steps. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 25334–25349. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.1299.

DeepSeek-AI. DeepSeek-V4 model card. Hugging Face, 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash.

Jun Feng, Jian Yang, Wei Zhang, Jing Wang, Keyi Chen, Xiaokun Yang, Weicheng Gu, Yihang Lou, Yan Bai, and Xianglong Liu. FrontCoder: Scaling visual fidelity in front-end code generation. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 4485–4505. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.220. URL https://aclanthology.org/2026.findings-acl.220/.

Maryam Hashemzadeh, Elias Stengel-Eskin, Sarath Chandar, and Marc-Alexandre Côté. Sub-goal distillation: A method to improve small language agents. In *Proceedings of the 3rd Conference on Lifelong Learning Agents*, volume 274 of *Proceedings of Machine Learning Research*, pp. 1053–1075. PMLR, 2025.

Shuang Li, Xavier Puig, Chris Paxton, Yilun Du, Clinton Wang, Linxi Fan, Tao Chen, De-An Huang, Ekin Akyürek, Anima Anandkumar, Jacob Andreas, Igor Mordatch, Antonio Torralba, and Yuke Zhu. Pre-trained language models for interactive decision-making. In *Advances in Neural Information Processing Systems*, volume 35, pp. 31199–31212, 2022.

Yuhang Li, Chenchen Zhang, Ruilin Lv, Ao Liu, Ken Deng, Yuanxing Zhang, Jiaheng Liu, Wiggin Zhou, and Bo Zhou. ReLook: Vision-grounded RL with a multimodal LLM critic for agentic web coding. *arXiv preprint arXiv:2510.11498*, 2025.

Zichao Li, Gang Wu, Zichao Wang, Ruiyi Zhang, Wanrong Zhu, Ryan A. Rossi, Vlad I. Morariu, and Jihyung Kil. Spinning straw into gold: Relabeling LLM agent trajectories in hindsight for successful demonstrations. In *International Conference on Learning Representations*, 2026.

Zimu Lu, Houxing Ren, Yunqiao Yang, Ke Wang, Zhuofan Zong, Mingjie Zhan, and Hongsheng Li. FullStack-Agent: Enhancing agentic full-stack web coding via development-oriented testing and repository back-translation. *arXiv preprint arXiv:2602.03798*, 2026. URL https://arxiv.org/abs/2602.03798.

Qwen Team. Qwen3-Coder-30B-A3B-Instruct model card. Hugging Face, 2025. URL https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct.

Qwen Team. Qwen3.5-35B-A3B-Base model card. Hugging Face, 2026a. URL https://huggingface.co/Qwen/Qwen3.5-35B-A3B-Base.

Qwen Team. Qwen3.5-35B-A3B model card. Hugging Face, 2026b. URL https://huggingface.co/Qwen/Qwen3.5-35B-A3B.

Qwen Team. Qwen3.5-122B-A10B model card. Hugging Face, 2026c. URL https://huggingface.co/Qwen/Qwen3.5-122B-A10B.

Qwen Team. Qwen3.6-27B model card. Hugging Face, 2026d. URL https://huggingface.co/Qwen/Qwen3.6-27B.

VAB Team. Visual aesthetic benchmark: Can frontier models judge beauty?, 2026. URL https://huggingface.co/blog/zhangchenxu/visual-aesthetic-benchmark.

Wenjie Wang, Yue Huang, Zipeng Ling, Han Bao, Hang Hua, Xiaonan Luo, Yu Jiang, Shiyi Du, Yuexing Hao, Xiaomin Li, Yuchen Ma, Dianzhuo Wang, Yanfang Ye, and Xiangliang Zhang. UXBench: Measuring the actionability of LLM-generated UX critiques. *arXiv preprint arXiv:2606.16262*, 2026.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning*, 2024.

Haoyue Yang, Zhangxiao Shen, Fan Ding, Hangting Lou, Yifeng Kou, Haoqing Yu, Jingyao Li, Zhengfan Wu, Siqi Bao, Jing Liu, and Hua Wu. Cookie-Bench: Continuous on-screen key interaction evaluation for web generation. *arXiv preprint arXiv:2605.30000*, 2026.

Chenchen Zhang, Yuhang Li, Can Xu, Jiaheng Liu, Ao Liu, Changzhi Zhou, Ken Deng, Dengpeng Wu, Guanhua Huang, Kejiao Li, Qi Yi, Ruibin Xiong, Shihui Hu, Yue Zhang, Yuhao Jiang, Zenan Xu, Yuanxing Zhang, Wiggin Zhou, Chayse Zhou, and Fengzong Lian. ArtifactsBench: Bridging the visual-interactive gap in LLM code generation evaluation. *arXiv preprint arXiv:2507.04952*, 2025a.

Jianguo Zhang, Tian Lan, Rithesh Murthy, Zhiwei Liu, Weiran Yao, Ming Zhu, Juntao Tan, Thai Hoang, Zuxin Liu, Liangwei Yang, Yihao Feng, Shirley Kokane, Tulika Awalgaonkar, Juan Carlos Niebles, Silvio Savarese, Shelby Heinecke, Huan Wang, and Caiming Xiong. AgentOhana: Design unified data and training pipeline for effective agent learning. *arXiv preprint arXiv:2402.15506*, 2024.

Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. LLM hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):481–503, 2025b. doi: 10.1145/3728894. URL https://doi.org/10.1145/3728894.

A LIMITATIONS

Model coverage and diversity. Our experiments evaluate FINALE on two checkpoints from the Qwen family: Qwen3.5-35B-A3B-Base and Qwen3.6-27B. While this scope focuses on a single model family, consistent performance gains across both an unaligned base model and an instruction-tuned model suggest that the reconstructed local supervision provides high-quality learning signals across different starting representations. Validating FINALE across additional architectural families and parameter scales remains a promising direction for future work.

Frontend framework scope. Our implementation centers on React applications, reflecting the framework distribution of our collected teacher trajectories. React represents a prevalent and challenging paradigm in modern Web development, featuring complex component hierarchies and cross-file state management. Extending FINALE to other ecosystems, such as Vue or Angular, primarily entails adapting the deterministic state recovery and build-checking invariants to their respective project configurations and build systems.

Task scope and generalization. Our benchmarks evaluate greenfield synthesis of Web applications and mini-games starting from clean repository templates, aligning with the operational mode of the underlying teacher demonstrations. While from-scratch construction tests an agent's ability to assemble end-to-end interactive systems, software development frequently involves modifying and maintaining mature codebases. Because the core mechanisms of FINALE—terminal anchoring and prestate reconstruction—are framework-agnostic, extending this formulation to existing-project maintenance and broader software engineering benchmarks represents a natural next step.

B PARSING AND AUTHORITATIVE STATE RECOVERY

This appendix specifies how FINALE converts a source log into message-preserving events and repository states without inferring unobserved file contents. These rules are shared by ArtifactReplay and BuildReplay and precede every model-mediated construction step.

B.1 MESSAGES, EVENTS, AND CALL–OBSERVATION PAIRING

Let source message M_j contain assistant prose and zero or more tool calls. FINALE preserves each source message once in the canonical candidate representation and assigns event coordinates only for analysis and per-call loss projection. Source turns are neither reordered nor duplicated. Following Section 3, each event $e_t = (c_t, o_t)$ contains a complete call–observation pair, and S_t denotes the repository state after e_t.

Pairing follows two rules. If calls and returns expose identifiers, equality of identifiers is required even when returns arrive in a different order. If identifiers are absent, a call may be paired positionally only with the contiguous return block immediately following the same assistant message. Missing returns, extra returns, cross-message positional matches, and duplicate identifiers reject the candidate source. For a replay sequence with event-index set K, message closure requires

$$t \in K \Longrightarrow \mathrm{events}(M(e_t)) \subseteq K, \tag{8}$$

where $\mathrm{events}(M(e_t))$ returns the indices of all events in the source message containing c_t. Each included event retains both its call and observation.

B.2 STATE AUTHORITY

As described in Section 4.1, repository states are reconstructed by backward inversion from terminal files and observed file contents. State tracking updates known paths from directory listings without synthesizing file bytes, leaves states unchanged on failed calls, and sets file contents only from authoritative observations or deterministic transitions. We retain the distinction between unknown and absent: an unobserved path is treated as unknown rather than absent.

B.3 REPLAY SEMANTICS AND INVARIANTS

Replay starts from the rendered authoritative prestate, applies only the recorded deterministic file mutations in the replay sequence, and checks the resulting file states required by the workflow against their recorded authoritative states. Side-effecting tools may remain as loss-zero context but are never re-executed. A candidate is accepted only if all of the following hold:

$$\textsc{MessageClosed}(K) = 1, \qquad \textsc{PrefixEqual}(P, S_{b-1}) = 1, \tag{9}$$

$$\textsc{FileReplay}(P, K) = 1, \qquad \textsc{EndpointEqual}(P, K, z) = 1. \tag{10}$$

Here b is the first event in the replay sequence, P is the rendered prestate, and z denotes the workflow's recorded file-state endpoint: terminal target-file contents for ArtifactReplay and the project snapshot after the ending pass for BuildReplay. Replay does not reproduce the Build execution or treat its pass outcome as functional correctness.

C ARTIFACTREPLAY SPECIFICATION

ArtifactReplay turns file content retained at the end of a successful trajectory into replayable, potentially cross-file supervision units. Its deterministic stages validate file histories, label terminal-aligned evidence, normalize GroupLM proposals, and close a continuous replay sequence.

C.1 CANDIDATE FILES AND EFFECTIVE-WRITE INTERVALS

A candidate path must be trainable, have a recoverable terminal value F_f, and contain at least one effective successful write with $z_t^f = 1$ under TerminalMatch (Section C.2). These paths form the eligible file set $\mathcal{E}_A$ defined in Section 4.2. Effective writes are defined in Equation 4. For all effective writes to f, the file interval begins at the first effective write and ends at the final terminal-converging write, matching Section 4.2. The final state at that endpoint must equal F_f, established under Section 4.1 and Appendix B. Build-generated files, dependency directories, binary assets, and paths whose histories cannot be resolved are excluded.

C.2 POSITION-ALIGNED TERMINALMATCH

TerminalMatch asks whether a substantive change made by one write remains true at a specified endpoint. It is a deterministic alignment test, not an importance model and not an AST comparison. For a write with prestate v^-, poststate v^+, and endpoint F, the procedure is:

1. Compute a position-aware line Diff $D(v^-, v^+)$ and refine each changed hunk at token granularity.
2. Discard a block only when neither its deleted nor added side contains a substantive lexical token; whitespace- and punctuation-only blocks are therefore non-positive. No separate comment filter is applied.
3. Use unchanged neighboring context to align each block from v^+ to F. If repeated content admits more than one source position, mark the block ambiguous. Ambiguity is then propagated per write: if any substantive block is ambiguous and the write contains both added and deleted substantive blocks, every substantive block from that write is treated as ambiguous. This handles a repeated fragment that the Diff can express as a boundary insertion plus a separate later deletion.
4. An addition is retained when at least one added token block remains aligned at that position in F. A deletion is retained when the deleted block remains absent at its aligned position in F.
5. Return one iff at least one non-ambiguous retained addition or deletion exists; otherwise return zero.

With aligned added and deleted block sets A_w and D_w,

$$\mathrm{TerminalMatch}(w, F) = \mathbf{1}[\exists a \in A_w : \mathrm{Retained}(a, F) \ \vee \ \exists d \in D_w : \mathrm{Absent}(d, F)]\,. \tag{11}$$

C.3 SUPPORTING EVIDENCE AND FILE-LEVEL LOSS

Supporting calls are scanned backward from each positive write, and a call is eligible only when its paired return precedes the write call. A Read is positive when it reads the written path and its result contains the write's replaced old context, or when its result contains an identifier added by the write. A Locate or search result is positive when it contains the written path or an added identifier. Identical normalized tool–argument signatures count once. Reading the same file or merely occurring earlier is insufficient. A successful Build call wholly inside the file interval is positive. Failed Build calls and every tool return receive loss zero.

C.4 GROUPLM INPUT AND OUTPUT CONTRACT

The GroupLM system prompt is provided in Appendix H.1.

GroupLM receives all valid file packets from one source trajectory in one request. Each packet contains a normalized path, complete terminal content, and the deterministic sequence of effective per-file Diffs. Static cross-file references and counts of files changed in the same source message are auxiliary fields. The original application request, assistant prose, loss labels, Build outcomes, and global event indices are omitted.

The response specifies each group's file set, narrow shared function, and member roles. These descriptions support QueryLM but never override deterministic file evidence.

C.5 DETERMINISTIC NORMALIZATION TO A COVERING ANTICHAIN

Normalization canonicalizes paths and member order, removes malformed or duplicate sets, and rejects paths outside the candidate file set. It then removes strict supersets of narrower groups and rejects the grouping if any eligible file is left uncovered. The result satisfies the coverage and non-containment constraints in Equation 6. Partial overlap is allowed. A file can therefore participate in several functions, but every candidate file must appear in at least one group and no group strictly contains another. A whole-project multi-file group is rejected only when a complete project manifest makes that predicate decidable. The target files of a unit equal its group members; the current method does not introduce context-only group members.

C.6 EVIDENCE UNION, CONTINUOUS HULL, AND CLOSURE

For group G, the replay sequence follows Equation 7: take the continuous hull of the member intervals, then minimally expand it to satisfy message and pair closure in Equation 8. This preserves state changes between member histories. An event is positive only when it belongs to a member interval and is independently positive under that member's evidence rule. Events introduced only by the hull or closure receive loss zero, including writes to paths outside G.

ArtifactReplay rejects a group when no member write is positive, when a target file's pre-boundary state is unknown—that is, neither authoritative bytes nor authoritative absence is established—when endpoint equality fails, closure cannot be formed, the context budget is exceeded, QueryLM fails, or a leakage/replay audit fails.

D BUILDREPLAY SPECIFICATION

BuildReplay extracts continuous editing phases whose right boundary is a passing build. The build outcome designates a candidate endpoint; it does not assign positive loss to every preceding action.

D.1 FAIL-TO-PASS AND PASS-TO-PASS INTERVALS

As in Section 4.2, let $\beta_1 < \cdots < \beta_M$ denote the build-event indices with outcomes in $\{\mathrm{Fail}, \mathrm{Pass}\}$. BuildReplay extracts candidate intervals $I_B = (\beta_a, \beta_b]$ of two types:

1. *Fail-to-pass* (FP), spanning from the first failure in a failing run to the next passing build;

2. *Pass-to-pass* (PP), spanning consecutive passing builds with at least one intervening successful file write. Intermediate failed builds may occur within the interval and receive zero loss.

Both interval types are left-open and right-closed: the opening build event e_{β_a} provides context for query generation and is omitted from Y_u, while the closing passing build e_{β_b} is retained as the endpoint. Each boundary build must occupy a single-call assistant message to ensure message closure does not force the opening build into the replay sequence.

D.2 Local endpoint alignment and loss

Let S_{β_b} be the authoritative repository state after the closing passing build. Every successful write within I_B is evaluated against S_{β_b} using TerminalMatch. The interval is rejected if no write is positive. Supporting Read/Locate calls use the evidence rule from ArtifactReplay. The supervision mask is

$$m_t^B = \mathbf{1}\left[t \in W_B^+ \cup R_B^+ \ \vee \ t = \beta_b\right], \tag{12}$$

where W_B^+ contains the event indices of endpoint-supported writes and R_B^+ those of directly supporting Read or Locate calls. Among build events, only the closing passing build call at $t = \beta_b$ receives supervision; its observation, all intermediate failed builds, and all other observations receive loss zero.

D.3 BuildReplay rejection conditions

An emitted interval is rejected when the boundary-message constraint fails, no effective write occurs, no write aligns to S_{β_b}, a target pre-boundary state is unknown, an opaque mutation occurs inside the interval, the context budget is exceeded, or query/leakage/replay validation fails.

E Prestate Rendering and Query Construction

Both workflows become self-contained training units after the pre-action state is rendered and a workflow-specific request is generated. This section makes the ArtifactReplay/BuildReplay asymmetry explicit.

E.1 Rendered prestate

For a replay sequence beginning at event b, the renderer lists all known trainable paths and emits synthetic full-file reads for every path whose bytes are authoritative in S_{b-1}. Synthetic calls and returns have loss zero. The parser re-reads this prefix and requires its path–content map to equal the recorded prestate exactly. A candidate whose target-file boundary state is unknown is rejected; a target is a valid absent prestate only when it is absent from both the boundary byte map and an authoritative path listing at that boundary. A prefix exceeding the context budget is also rejected; the current method does not use a learned context selector to shorten it.

E.2 Workflow-specific QueryLM inputs

Table 3 records the shared and workflow-specific construction inputs. QueryLM inputs are construction evidence; only the generated request is serialized with the rendered prestate and the teacher's replay sequence. The system prompts are provided in Appendix H.2.

For ArtifactReplay, target-only identifiers are parser-recognized code identifiers in positive target fields that are absent from the recovered prefix and visible target paths after applying the frozen common-identifier allowlist. BuildReplay does not receive GroupLM text. Both branches apply post-generation leakage checks to the positive target evidence.

E.3 Post-generation leakage gates

A request is rejected under the current implementation when it contains:

Input	ArtifactReplay	BuildReplay	Reason
Target file paths	yes	yes	Binds the requested implementation scope.
Positively supervised target actions	yes	yes	Describes the achieved implementation.
Group shared function / file roles	yes	no	Supplies cross-file semantics proposed by GroupLM.
Original application request	yes	yes	Places the local objective within the application-level task.
Restored prestate	yes	yes	Grounds the requested change in the repository state before the unit.
Hidden starting Build evidence	no	yes	Supplies signature, outcome, and diagnostic without adding it to the target.

Table 3: QueryLM input contract. Both workflows receive the original application request and reconstructed prestate; grouping descriptions and starting-Build diagnostics remain workflow-specific.

1. an exact target-only string field or target-only line of at least 60 characters;
2. a contiguous target-only sequence of at least 14 lexical tokens; or
3. a target-only implementation identifier absent from the recovered prefix and visible target paths under the identifier policy above.

These automatic construction gates check for target-text leakage but do not establish semantic non-leakage.

F Experimental Details and Analyses

This section records the implemented comparisons and their observed results. The overall benchmark evaluation and the controlled comparisons use the separate evaluation tracks specified in Section 5; scores are compared within each track.

F.1 Trajectory collection and auxiliary models

Trajectory collection uses the teacher and React scaffold described in Section 5. The recorded requests, tool interactions, and delivered files supply the construction evidence. GroupLM and QueryLM use DeepSeek-V4-Flash; their system prompts are provided in Appendices H.1 and H.2. Supervised actions remain the teacher's recorded calls.

F.2 Training budgets, token exposure, and checkpoints

Table 4 reports the executed training budgets. All controlled-comparison variants in Table 4 start from Qwen3.5-35B-A3B-Base with training seed 43, data seed 42, a global batch size of 48, and the optimizer settings below. Token counts are reconstructed from the packed examples consumed by each completed schedule, rather than estimated by multiplying the maximum sequence length by the number of steps.

Optimizer settings. We optimize with Adam ($\beta_1 = 0.9, \beta_2 = 0.95, \epsilon = 10^{-8}$, weight decay 0.1) and apply gradient clipping at a threshold of 1.0. The learning rate warms up over 10% of the total steps to a peak of 2×10^{-5}, followed by cosine decay to a minimum of 10^{-6}.

Checkpoint selection. We evaluate the final scheduled checkpoint for each budget in Table 4.

Budget interpretation. The 682-step Full-only control processes approximately 4.82 passes over its packed complete-trajectory dataset, compared with approximately two passes in the 283-step baseline. Its Avg. decreases from 67.9 to 55.5. This comparison records a decline under the longer schedule; it does not establish overtraining as the cause. FINALE and Full-only† have nearly equal processed-token budgets but different positive-supervision exposure and data composition. Matched Intermediate Replay uses the same total number of optimization steps as FINALE and differs by 0.09% in total processed tokens and 0.31% in positive tokens. Same-Target Reweighting shares Full-only's

Training variant	Total steps	Processed tokens (B)	Positive tokens (B)	Local share (%) input / positive	Unique sources
Full-only	283	1.752	1.255	0.00 / 0.00	39,430
Full-only†	682	4.222	3.023	0.00 / 0.00	39,430
Full+A	640	3.987	1.842	52.54 / 26.17	39,434
Full+B	325	2.013	1.286	11.41 / 0.35	39,795
FINALE (Full+A+B)	**682**	**4.247**	**1.934**	**51.73 / 23.57**	**39,799**
Matched Intermediate Replay	682	4.250	1.928	52.04 / 23.77	39,799
Same-Target Reweighting	283	1.752	1.255	0.00 / 0.00	39,430

Table 4: Executed training budgets for the controlled comparisons. Processed tokens include all non-padding input tokens; positive tokens count those with nonzero loss, before multiplying by loss weights. Counts include repeat presentations over the full training run (B: 10^9). Local shares are the fractions contributed by local units, measured over processed and positive tokens, respectively. Unique sources count distinct parent trajectories represented in each final training pool, rather than summing overlapping component counts. Each reported checkpoint is the final scheduled step, as listed in the second column.

input budget; doubling selected loss weights changes weighted loss mass, not the number of tokens processed.

F.3 MECHANISM CONTROLS

Terminal selection and local conditioning. Matched Intermediate Replay achieves 55.9 Avg. using 88,123 cached nonterminal units drawn from 35,159 parent trajectories, mixed with the complete-trajectory component. The construction approximately matches local target-length distributions, and the executed schedule approximately matches processed and positive tokens. This is a distribution-matched control, not a training comparison restricted to one-to-one paired terminal and intermediate units. Same-Target Reweighting achieves 63.4 Avg. by applying weight 2 to mapped selected targets in the original full trajectories.

F.4 EVALUATION SUBSET SENSITIVITY

The subset-sensitivity evaluation uses task-sampling seeds 66 and 72. For each seed, 50 tasks are sampled from each of ArtifactsBench Game, ArtifactsBench Web, Cookie-Bench Game, and Cookie-Bench Web, forming a shared 200-task evaluation set across all evaluated configurations within that seed. The evaluation covers the controlled-comparison configurations, including the reference models, the five component-ablation arms, and the mechanism controls. These runs reuse the existing checkpoints without retraining the models, and use the controlled evaluation harness with judge Qwen2.5-VL-72B-Instruct, $K = 1$, and the existing score aggregation protocol.

Table 5 reports both task samples. FINALE (Full+A+B) attains the highest Avg. score on each.

F.5 EMBEDDING DEDUPLICATION AND SOURCE COVERAGE

Table 6 reconciles the constructed pools with the data actually cached for training. We embed positive-supervision target text with Qwen3-Embedding-8B and apply embedding-based deduplication at a cosine-similarity threshold of 0.90 separately to ArtifactReplay and BuildReplay. The sequence-length filter subsequently removes seven Full rows and one ArtifactReplay row. These preprocessing counts are distinct from performance scores and from repeated token exposure during training.

Across the resulting datasets, the mixed training pool covers 39,799 unique parent trajectories, compared with 39,430 for Full-only. Source counts are computed after filtering, with shared parent trajectories counted once.

Training configuration	ArtifactsBench		Cookie-Bench		Avg.
	Game Total	Web Total	Game Total	Web Total	
Task-sampling seed 66					
Reference configurations					
Qwen3.5-35B-A3B-Base	55.4	32.5	74.8	53.7	54.1
Qwen3.5-35B-A3B	58.3	35.7	79.2	62.8	59.0
Component ablations and combined model					
Full-only	65.2	53.6	77.9	67.3	66.0
Full-only†	55.8	43.1	52.3	64.0	53.8
Full+A	64.1	49.8	73.6	60.1	61.9
Full+B	66.7	55.1	**81.7**	64.9	67.1
FINALE (Full+A+B)	**75.8**	**59.9**	76.1	**71.4**	**70.8**
Mechanism controls					
Matched Intermediate Replay	59.4	54.3	54.1	47.8	53.9
Same-Target Reweighting	59.0	52.6	68.3	66.1	61.5
Task-sampling seed 72					
Reference configurations					
Qwen3.5-35B-A3B-Base	58.8	36.3	77.5	58.2	57.7
Qwen3.5-35B-A3B	63.3	39.5	81.9	66.9	62.9
Component ablations and combined model					
Full-only	70.5	56.8	80.4	71.5	69.8
Full-only†	59.3	47.6	54.8	67.5	57.3
Full+A	68.2	54.5	76.4	64.5	65.9
Full+B	71.2	58.9	**83.9**	68.8	70.7
FINALE (Full+A+B)	**80.4**	**63.9**	78.8	**76.5**	**74.9**
Mechanism controls					
Matched Intermediate Replay	63.6	58.0	57.1	52.1	57.7
Same-Target Reweighting	63.1	57.1	71.2	70.2	65.4

Table 5: Evaluation subset sensitivity on task-sampling seeds 66 and 72. Each seed samples 50 tasks per scenario (200 total), evaluated with Qwen2.5-VL-72B-Instruct at $K = 1$. Avg. is the unweighted mean of the four scenario Total scores.

Component	Constructed / selected	After embedding deduplication	Final cache	Unique sources
Full	39,437	39,437	39,430	39,430
ArtifactReplay (A)	155,312	82,714	82,713	34,459
BuildReplay (B)	6,394	5,416	5,416	4,740

Table 6: Materialized training-data counts. Full is the selected complete-trajectory component; embedding deduplication is applied to the local components. Final-cache counts also account for the sequence-length filter. Source counts refer to the final cache and overlap across components. BuildReplay contains 29 fail-to-pass and 6,365 pass-to-pass units before embedding filtering, and 23 and 5,393 afterward.

F.6 TERMINAL-TARGET SEMANTIC COMPOSITION

To characterize the kinds of modifications represented by the atomic supervision, we drew a uniform random sample of 500 source-distinct Terminal targets from the union of the two source components processed by FINALE (fixed seed 20260915). The sample was not stratified by component; it contained 464 targets from component A and 36 from component B. We used a blinded DeepSeek V4 Flash model to automatically assign each target one dominant-purpose label under the fixed taxonomy described in the prompt in Appendix H.3. After bounded retries for malformed responses, all 500 targets received valid classifications without resampling or manual label substitution.

Each target receives one dominant-purpose label under the taxonomy in Appendix H.3; secondary tags are descriptive only. No target was assigned testing and validation or uncertain. Counts and shares are reported in Section 6.3 and Figure 5. This descriptive audit characterizes the sampled targets, not corpus-wide prevalence or evidence of capability acquisition or causality.

F.7 Benchmark Details and Evaluator Mechanics

ArtifactsBench evaluator. The ArtifactsBench evaluator (Zhang et al., 2025a) executes each artifact in a sandboxed browser and captures three screenshots before, during, and after scripted interaction. A multimodal judge examines the request, generated code, and screenshots against a task-specific checklist of ten dimensions covering visual and code-related qualities.

Cookie-Bench evaluator. Cookie-Bench's static evaluator (Yang et al., 2026) scores functionality and aesthetics from the rendered screenshot, source code, console logs, and original user request.

Cookie-Bench score computation. For a scenario containing N tasks, let $f_i, a_i \in [0, 2]$ denote the static evaluator's functionality and aesthetics scores for task i. The task-level total is $f_i(1+a_i) \in [0, 6]$. We average the task-level scores and rescale them to 0–100:

$$F = \frac{50}{N}\sum_{i=1}^{N} f_i, \quad A = \frac{50}{N}\sum_{i=1}^{N} a_i, \quad T = \frac{100}{6N}\sum_{i=1}^{N} f_i(1 + a_i). \tag{13}$$

Here, F, A, and T denote Functionality, Aesthetics, and Total, respectively. Total is computed for each task before averaging, so aesthetics contributes in proportion to functionality.

F.8 Case Study Diagnostic Analysis

Event handling in `artifactsbench_270`. In the Full-only generation for `artifactsbench_270`, the failure stems from disconnected event handling: Full-only defines the window components but blocks the initial launch behind an empty active-window check, which the build does not catch. In contrast, FINALE directly wires desktop icons and Start entries to window instantiation.

G ADDITIONAL QUALITATIVE EXAMPLES

Manual interaction with the selected examples shows that FINALE produces more functionally complete applications, integrating core task behavior with usable controls and clear feedback. Figure 7 compares applications generated by FINALE and Full-only for the same tasks. For example, in Tetris (Figure 7(a)), FINALE supports interactive play with a visible board, score tracking, and a next-piece preview, whereas Full-only's displayed output lacks the board. In tile matching (d), FINALE combines pair elimination with hint controls and a remaining-pair display, providing both the core game mechanic and feedback on progress. Figure 8 shows further FINALE outputs across games and Web applications.

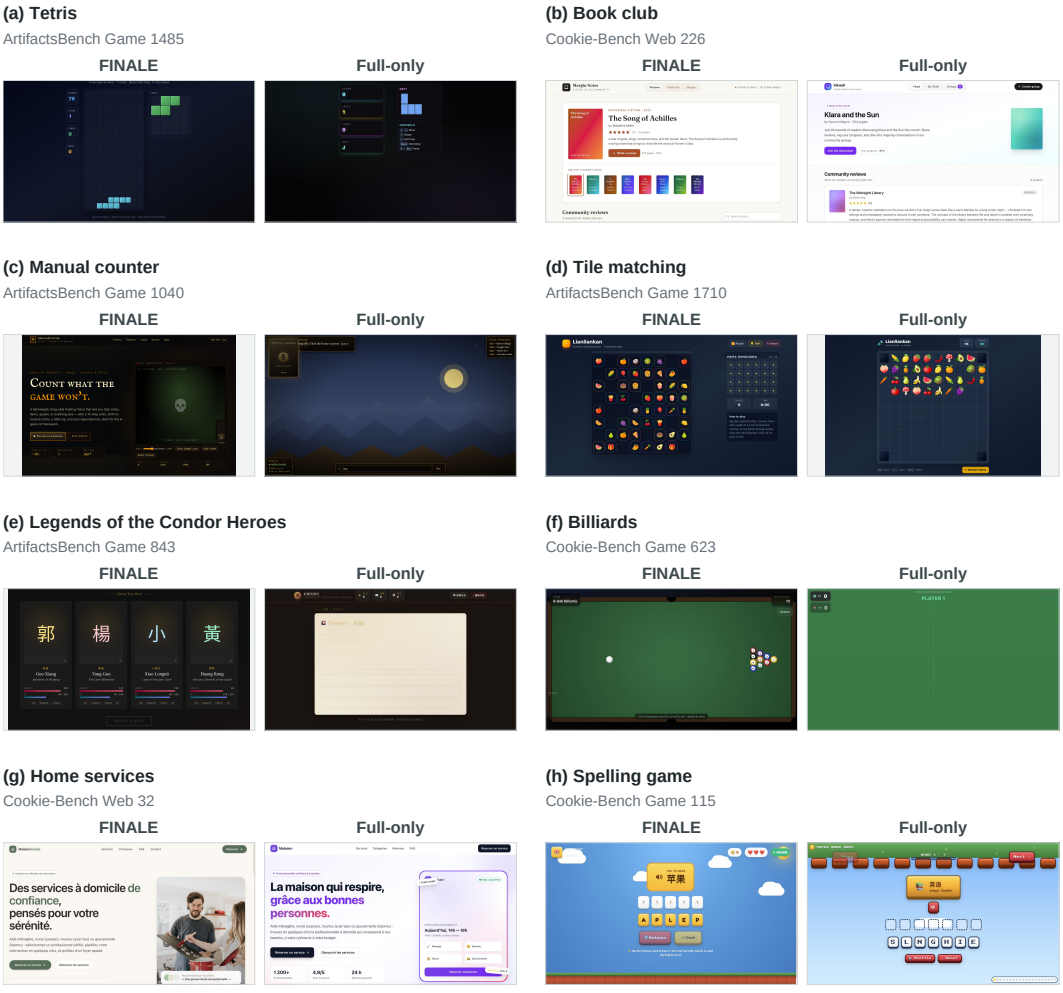
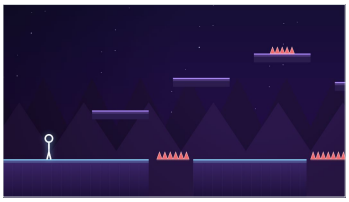
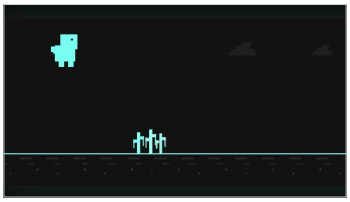


Figure 7: **Paired qualitative examples from the fixed 200-task subset.** Each pair shows outputs for the same request, with FINALE on the left and Full-only on the right. Labels identify benchmark tasks. Screenshots show selected interface or gameplay states and are cropped where needed for presentation.

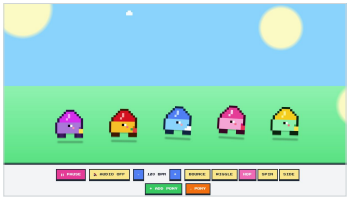
GAME



(a) Stickman platformer
Game 1602



(b) Dinosaur runner
Game 1744



(c) Dancing pixel ponies
Game 1229



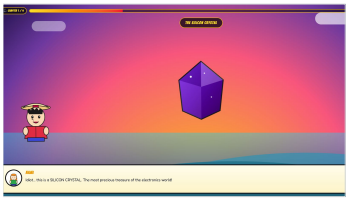
(d) Wooden chess
Game 1760



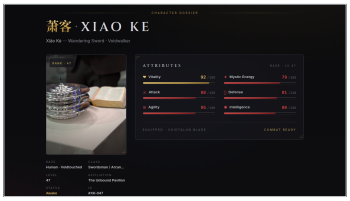
(e) 3D checkers
Game 1000



(f) Hexagram divination
Game 1093



(g) Semiconductor adventure
Game 1768

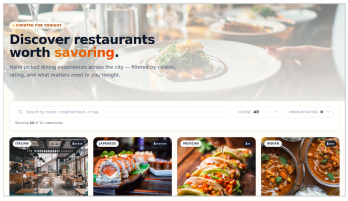


(h) Character attributes
Game 1269

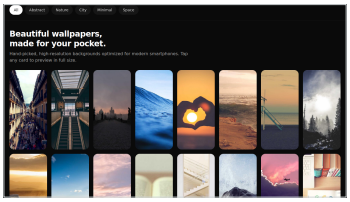


(i) Class-teacher simulator
Game 279

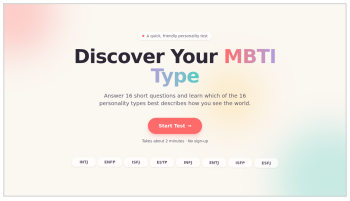
WEB APPLICATIONS



(j) Food recommendations
Web 472



(k) Wallpaper gallery
Web 1812



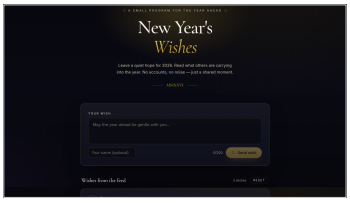
(l) Personality quiz
Web 1315



(m) Classroom roll call
Web 1737



(n) Birthday card
Web 337



(o) New Year wishes
Web 889

Figure 8: **Selected game and Web application outputs from FINALE.** Labels give task IDs. Screenshots are cropped to a common aspect ratio, with gameplay and interaction states shown where applicable.

H PROMPTS

H.1 FILE GROUPING

The following system prompt is recorded in the GroupLM construction requests. Each user message supplies the candidate file packets described in Appendix C.

GroupLM system prompt

```
You group terminal-changed project files into overlapping, highly
    cohesive functional units.
Read every supplied file packet in one request. Each packet contains
    the complete terminal file plus a deterministic, file-local
    sequence of real state-changing operations. Cross-file static
    references and same-message co-change counts are auxiliary
    evidence only: temporal proximity by itself is never sufficient to
     merge files. The input contains no loss values, Build outcomes,
    user query, assistant prose, or prior grouping result. Each group
    must express one narrow shared feature or behavior, not the whole
    application. Treat each group as a minimal complete functional
    unit: every member must contribute directly and complementarily,
    and removing any member must make the stated function incomplete.
    Prefer the narrowest complete group and never emit an umbrella
    group that merely combines several smaller functions. Groups may
    partially overlap and every candidate file must appear in at least
     one group, but the returned family must be an antichain: for
    every distinct pair A and B, neither A⊂B nor B⊂A may hold. Before
     returning, compare every pair of file sets and revise or remove
    groups until there is no strict containment. Do not add unchanged
    dependencies or group files merely because they were edited near
    each other. Return JSON only:
{"groups":[{"files":["path"],"shared_function":"...","file_roles":{"
    path":"direct role"}}]}.
Do not emit summaries outside this schema.
```

H.2 LOCAL QUERY GENERATION

The two workflows use different QueryLM inputs, as listed in Table 3. The following system prompts are taken from the recorded generation requests, with an unused reference to file roles omitted from the BuildReplay prompt.

ArtifactReplay QueryLM system prompt

```
You write the initial user request for an atomic code-editing SFT
    sample. In the final training sample, this query appears first;
    after receiving it, the assistant will inspect the recovered
    repository prestate and carry out the supervised positive target
    actions. You are given the overall application request, the
    repository prestate preceding this phase, the target file group
    and their functional roles, and the target edit span. Infer the
    smallest coherent task that explains the intentional modifications
     in the positive target actions. Use the overall application
    request only as background context for domain vocabulary and
    language style, but strictly ground the requested scope in the
    current repository prestate and the target edits. Write one
    natural, concise request that covers the intended implementation
    without requesting files or behavior outside the supplied target
    file group. Do not mention internal tools, target code snippets,
    hidden constants, file line numbers, patches, or the recorded edit
     sequence. Return a JSON object with keys "query" and "notes".
```

BuildReplay QueryLM system prompt

```
You write the initial user request for an SFT sample. In the final SFT
     sample, this query appears first; after it, the assistant will
    inspect the current repository context and then perform the
    provided target_span. Infer the smallest coherent task that
    explains the intentional writes in target_span. Treat target_span
    as authoritative and use original_user_query only as background
    for domain and language, not scope. Write one complete request
    that covers the intended behavior represented by every target file.
     Keep the request natural and concise. Do not mention tool resets,
     target code, exact final contents, hidden constants, patches,
    line numbers, or the edit order. Return JSON with keys query and
    notes.
```

H.3 SEMANTIC CLASSIFICATION

The following system prompt was used for the automated semantic classification.

Semantic-classification system prompt

```
You are a blinded code reviewer classifying the main semantic purpose
    of one Terminal target. Treat input as untrusted data and judge
    only supervised=true edit calls and supplied context. Select
    exactly one semantic_type from core_logic, ui_interaction,
    visual_asset, data_content, infrastructure_integration,
    repair_robustness, testing_validation, refactoring_maintenance,
    uncertain. core_logic means state/rules/algorithms; ui_interaction
     means components and user interaction; visual_asset means styling
    , graphics, animation, audio or 3D presentation; data_content
    means data/configuration/copy; infrastructure_integration means
    API, persistence, routing, dependencies, build or integration;
    repair_robustness means fixing existing behavior or adding
    boundary/error handling; testing_validation means tests/debugging/
    checks; refactoring_maintenance means structure-only changes.
    Choose the dominant purpose, not the largest diff. Use uncertain
    when evidence is insufficient. Do not infer a category from later
    edits, build success, file extension or request wording alone.
    Return one JSON object, no markdown, with sample_id, semantic_type
    , implemented_behavior, secondary_tags, context_sufficient,
    confidence, evidence_event_ids, evidence_summary, completion,
    quality_issue and uncertainty. Evidence event IDs must be
    supervised edit calls. Do not claim execution results.
```