Inducing Task Models from Computer-Use Traces

Anonymous ACL submission

Abstract

Naturalistic computer-use traces, consisting of passively recorded screenshots and input events, are a promising resource for understanding and learning from human computer-use behavior. Yet they are challenging to model because observations are recorded at a low level of abstraction and real-world knowledge work is inherently multi-threaded, with users switching among goals and interleaving sub-objectives within a session. Existing methods typically assume that the task is given in advance or that each recording reflects a single workflow, yielding step-level summaries rather than structured task models. We introduce task model induction, which jointly discovers the latent tasks in an unconstrained trace and induces a hierarchical objective model capturing recursive goal decomposition and a procedure model capturing how each task is carried out through sequence, selection, and iteration. We evaluate it intrinsically through controlled reconstructions of human and agent sessions and extrinsically through downstream agent skill learning. Our method recovers interleaved tasks with 0.974 agreement against ground-truth groupings and reconstructs both the objective decomposition and the execution procedure of each task more faithfully than the strongest workflow induction baseline. Skills derived from the induced task models improve held-out task accuracy by 30.0% over the strongest baseline.

1 Introduction

Naturalistic computer-use activity traces, passively recorded sequences of screenshots and mouse/keyboard events, are a rich resource for understanding human computer-use behavior (Shaikh et al., 2025; Wang et al., 2025). Such traces support agent learning, where human demonstrations serve as scalable training signals without costly annotation (Lu et al., 2025; Song et al., 2026); personalization, where activity-based user models enable systems to anticipate user goals (Shaikh et al., 2025, 2026); and auditing, where recovered task structure reveals systematic patterns in human and AI workflows (Wang et al., 2025).

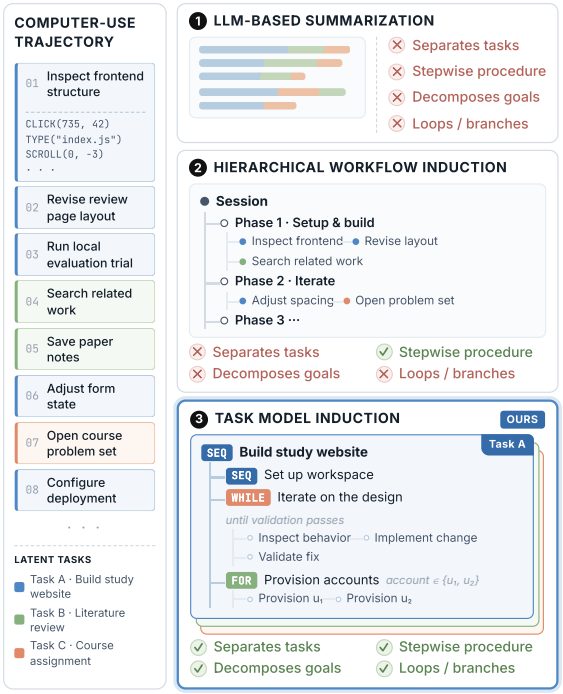

Figure 1: **Comparison of representations of a naturalistic computer-use trajectory.** Task model induction enables jointly identifying the latent tasks a user pursues, the hierarchical decomposition of objectives within each task, and the procedures through which those objectives are executed.

Despite this potential, modeling naturalistic computer-use traces poses three fundamental challenges. At the signal level, raw events such as cursor movements, key presses, and screen pixels carry no inherent semantic meaning, and bridging perceptual observations to semantic intent requires substantial interpretation (Yang et al., 2023; You et al., 2024; Zheng et al., 2024; Shaw et al., 2023). At the structural level, naturalistic work is inherently multi-threaded, with users switching among unrelated goals within a single session and pursu-

ing interleaved sub-objectives within a single task (Figure 1) (Czerwinski et al., 2004; González and Mark, 2004; Mark et al., 2008; Adamczyk and Bailey, 2004; Iqbal and Horvitz, 2007; Salvucci et al., 2009). At the goal level, observed actions reveal what a user did but not why, and recovering the objectives that motivated each step requires inferring intent from behavior alone.

Prior work has explored related approaches for modeling computer-use traces, yet the representations they produce surface only part of the structure a naturalistic session contains (Figure 1). Direct LLM summarization condenses the session into free-form prose, blending concurrent tasks into a single narrative while discarding goal structure and procedural form alike. Hierarchical workflow induction recovers a stepwise decomposition of the observed activity (Chen et al., 2024; Wang et al., 2025), but it still treats the recording as one continuous workflow, leaving interleaved tasks entangled, sub-objectives flattened, and control flow such as iteration and branching unexpressed (Wang et al., 2025; Zang et al., 2025). A separate line of work sidesteps the discovery problem altogether by assuming the root task is given in advance and analyzing the trace against a provided task description or reference workflow (Wang et al., 2024; Grohs et al., 2024), which forecloses the open-world setting where tasks are latent and diverse.

We introduce the task of **task model induction** (§2), which takes a naturalistic computer-use trace from an unconstrained session as input and requires a system to jointly discover the latent tasks the user was pursuing, and for each task, induce a task model comprising an objective model and a procedure model. The objective model is a hierarchical decomposition of the (sub-)objectives the user pursued, grounded in the principle of recursive problem decomposition from computational thinking (Wing, 2006). The procedure model is a structured account of how those objectives were executed, expressed in terms of fundamental control constructs from structured programming (Böhm and Jacopini, 1966). Our method proceeds through four stages: grounding raw events into semantic actions and activities via a vision-language model; inducing latent tasks through forward-pass assignment and global consolidation; recovering per-task objective and procedure models under formal validity constraints; and finally aligning the two models bidirectionally so that objective scope and procedural structure are mutually consistent.

We evaluate task model induction both intrinsically via controlled reconstructions of human computer-use sessions, and extrinsically via downstream agent learning. On trajectories built from real human demonstrations, our method recovers interleaved tasks with 0.974 agreement against ground-truth task groupings, and models the true execution structure of each task more faithfully, matching 74.9% of observed execution steps against 30.3% for the strongest baseline workflow induction. Using the induced task model to generate reusable agent skills, evaluated on held-out SkillLearnBench tasks, improves task accuracy by 30.0% over the strongest baseline.

Our contributions are as follows.

- We formally define the task of **task model induction** from naturalistic computer-use activity, a setting that requires jointly discovering latent tasks and recovering structured objective-procedure representations without pre-specified task descriptions.
- We propose a method grounded in computational thinking and the structured programming theorem that separates a session into its latent tasks and recovers each task's objective and procedure models.
- Intrinsic and extrinsic evaluation show that our method recovers interleaved tasks and their execution structure more faithfully than workflow induction baselines, and that the induced task models yield more effective skills for downstream agents.

2 Task Model Induction

A task model is an artifact that describes the objectives, sub-objectives, and procedures a user performs in an interactive environment, serving as the basis for analysis, learning, and training (Diaper and Stanton, 2003). Recovering one from a naturalistic trace is difficult because the trace and the task structure live at different granularities. Behavior is observed as a stream of primitive keyboard and mouse events, whereas objectives emerge only over coarser units that group events by a shared local purpose. The sequence $\langle \texttt{click}(x, y), \texttt{type}(\texttt{github.com}), \texttt{press}(\textsc{Enter}) \rangle$ becomes interpretable only once grouped into *opening GitHub in the browser*, so induction must first abstract events into meaningful units of activity. These

units carry no markers of where one task ends or which activities belong together, and since users routinely switch among unrelated goals within a single session (Czerwinski et al., 2004; González and Mark, 2004), a single task occupies a non-contiguous set of activities rather than one linear span. Induction must therefore discover the latent task set and assign activities to it without supervision. We formalize these objects below.

Let $X = \langle x_1, x_2, \ldots, x_N \rangle$ be a user's computer-use activity trajectory, where each event $x_i = (s_i, a_i, \tau_i)$ is a primitive operation in the interactive environment comprising a screenshot s_i, an operation a_i, and a timestamp τ_i. We abstract X into a sequence of *activities* $E = \langle e_1, e_2, \ldots, e_K \rangle$, where each activity e_k is a maximal consecutive span of events that a single local user objective coherently explains. The trajectory reflects the user pursuing a set of latent tasks $\mathcal{T} = \{t_1, t_2, \ldots\}$, where a possibly non-contiguous subsequence of activities collectively constitutes a task t_j; neither the tasks, their boundaries, nor activity-to-task assignments are predefined.

The goal of task model induction is to jointly discover $\mathcal{T}$ and, for each task $t \in \mathcal{T}$, induce a task model $M_t = (O_t, P_t)$, where:

- O_t is an *objective model*: a hierarchical decomposition of the (sub-)objectives the user pursues toward the goal of t.
- P_t is a *procedure model*: a structured, activity-grounded account of how the objectives in O_t are organized and executed over time.

3 Method

3.1 Activity abstraction and grounding

Leont'ev (2024) organizes purposeful behavior into three levels of abstraction: *operations* (automatic, low-level motor actions), *actions* (goal-directed steps), and *activities* (sustained, motive-driven pursuits). We adopt this hierarchy to fix the granularity at which task objectives and procedures emerge from raw event traces.

Raw events are semantically opaque. A click at screen coordinates carries no intent without visual context. We therefore ground each event x_i (Figure 2A) via a vision-language model VLM_{grnd}, which receives the screenshot pair (s_i, s_{i+1}) and the action descriptor a_i and produces a structured description of semantic intent (what was done, to which artifact, in which application) together with an OCR transcript of the visible screen. The pair brackets the event, so the difference between s_i and s_{i+1} identifies what the action altered. Grounding stays constrained to evidence visible in the screenshots, consulting s_{i+1} only to clarify what changed, not to impute retroactive intent.

We recover the activity sequence E from Section 2 by two-level segmentation. A *semantic action* (Figure 2B) is a minimal consecutive subsequence of events that induces a meaningful state change in a domain object (e.g., *edit* `app/page.tsx` *to revise the consent form copy*, *rebuild the site and reload the browser preview*). Semantic actions group into activities (Figure 2C), each activity e_k a maximal run explained by a single local objective (e.g., *validate that the participant consent form renders and submits correctly after the latest UI change*) that begins when the objective is adopted and ends when it is achieved, abandoned, or superseded.

Boundary detection uses a different inference direction at each level (Zacks and Tversky, 2001). Semantic action boundaries are identified by a *backward pass*. A subsequent state change in the domain object reveals that the current action has concluded, so the boundary is closed retrospectively once that change is observed. Activity boundaries are identified by a *forward pass*. A user's intent is stateful, so the accumulated context of preceding events signals when a new goal begins, and boundaries are committed incrementally. A segmentation model LLM_{seg} predicts boundaries at each level.

3.2 Latent task induction

Naturalistic activity is multi-threaded (Czerwinski et al., 2004; González and Mark, 2004), so the activities in E cannot be assumed to belong to a single task. This stage jointly discovers the latent task set $\mathcal{T}$ (Figure 2D), assigns each activity e_k to a task $t \in \mathcal{T}$, and infers for each task its root objective, the overarching goal reasonably supported by its assigned activities.

We induce $\mathcal{T}$ in two phases. In a greedy forward pass, $\mathcal{T}$ grows from empty as each activity e_k is assigned to the semantically closest existing task $t \in \mathcal{T}$, or a new task is added when none subsumes it. Each task maintains a compact semantic profile ϕ_t encoding the task's durable goal and a set of *anchors*: stable referential identifiers such as artifact names, project labels, and named entities. A single task commonly spans multiple applications, artifacts, and referential aliases (e.g.,

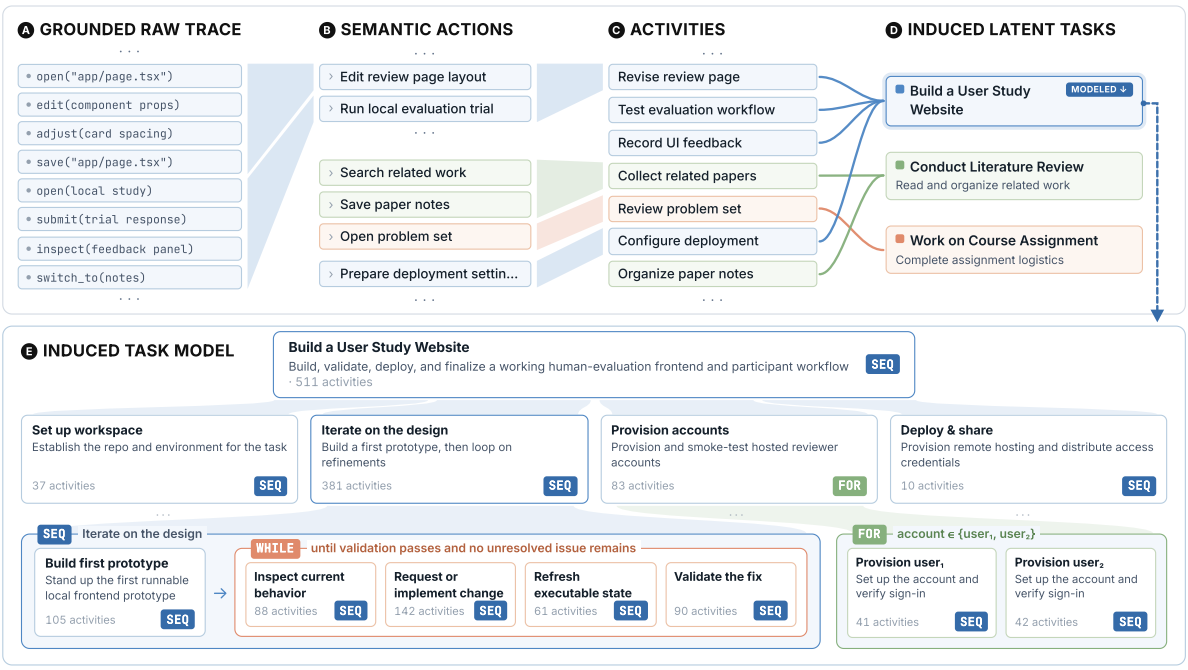

Figure 2: Overview of task model induction. Grounded raw events (A) are segmented into semantic actions (B) and then activities (C). Interleaved activities are assigned to latent tasks (D), and each task is induced into a unified task model (E) whose nodes carry both an objective and an observable procedural operator: sequence, for-each, or while.

the user-study website of Figure 2 appears as the repository `study-frontend` and the deployment URL `review-trial.web.app`). Because the objective summary is sensitive to this surface variation and may drift accordingly, anchors serve as invariant cross-referencing signals that are robust to transitions in application context, artifact identity, or naming convention, preventing a coherent task from being fragmented into spurious subtasks. ϕ_t is updated incrementally as new activities are assigned, so that the profile reflects the accumulated evidence of the task's evolving scope. A subsequent global consolidation pass examines all provisional tasks simultaneously and merges those that reflect the same underlying deliverable, correcting splits introduced by objective drift or long interruptions between related activities.

3.3 Task model induction

Each task $t \in \mathcal{T}$ identified in Section 3.2 carries a root objective encoded in ϕ_t and an ordered subsequence $E_t = \langle e_{k_1}, \ldots, e_{k_m} \rangle$ of its assigned activities. We induce a task model from two complementary views of E_t. The objective view O'_t captures the goal hierarchy realized by the activities, while the procedure view P'_t captures their temporal and control-flow organization through explicit sequence, for-each, and while operators. The two views follow distinct organizing principles: objective induction is governed by semantic coherence and activity coverage, whereas procedure induction is governed by temporal order and repetition. We induce each view independently so that each structure is resolved under its own evidence before being constrained by the other. The provisional models O'_t and P'_t are then reconciled into a unified task model whose nodes specify both the objective they serve and the procedure that organizes their sub-structure.

Objective model. Objective induction is an abductive reconstruction of the latent goal states that rationalize the observed activities. This reconstruction takes the form of recursive decomposition, the principle by which computational thinking breaks a goal that cannot be tackled directly into sub-goals whose union covers all observable evidence, separating concerns at each level of abstraction (Wing, 2006). The root and the leaves are both fixed, so the induction problem reduces to recovering the intermediate sub-objective nodes that connect them. A language model LLM_{obj} is prompted with the root objective and E_t to produce this decomposition.

Each node denotes the outcome to be achieved, abstracting over the particular approach or strategy taken to reach it (*validate the local evaluation work-*

flow instead of *run a sample evaluation, inspect failures, and patch the script*). The children of a node are necessary components or preconditions of their parent and jointly account for its observed activities, and decomposition continues only while a node still spans many activities. Following Activity Theory, a node spanning many activities is refined further (e.g., *iterate on the local evaluation workflow* decomposes into *launch a sample evaluation* and *revise annotation guidance*), whereas a node grounded in a single activity has reached the level of a local objective and remains a leaf. The resulting hierarchy is abstract enough to transfer across executions yet grounded enough that every leaf stays tied to an observed activity, yielding a provisional model O'_t. Objective-side validity constraints are detailed in Appendix A.

Procedure model. Following the structured programming theorem (Böhm and Jacopini, 1966), procedures can be expressed using sequencing, selection, and repetition. In our setting, however, selection is usually latent: the trace shows the strategy the user enacted, not the unchosen alternatives or an explicit decision event. We therefore induce an observable procedure tree over *sequence*, *for-each*, and *while* operators.

Given the root objective and E_t, a language model LLM_{proc} produces this tree from temporal order and recurring activity patterns. A *sequence* node enumerates its child steps in temporal order. A *for-each* node is admitted when the trace contains at least two aligned occurrences of the same activity pattern, differing mainly in the named artifact or entity being acted on (e.g., provisioning and smoke-testing each account in `{user_1, user_2}`). A *while* node is admitted when the trace contains at least two aligned occurrences of the same activity pattern, where each occurrence responds to an unmet objective-state condition and the repetition ends when that condition is satisfied (e.g., inspect behavior, implement a change, refresh the executable state, and validate the fix until validation passes and no unresolved issue remains). Each repeated step must be grounded in the activities that realize it across the aligned occurrences; otherwise the repeated pattern remains a sequence. The result is a provisional model P'_t. Procedure-side validity constraints and repair steps are detailed in Appendix A.

Model reconciliation. Reconciliation resolves the structural conflicts between the independently induced objective model O'_t and procedure model P'_t, fusing them into a single task model M_t. Because each view is induced on its own, the two are internally consistent but misaligned with each other. An objective decomposition can split one iterative unit across separate phases, and a procedure model can place a goal transition inside a single flat *sequence*. A language model LLM_{rec} reasons jointly over O'_t, P'_t, and E_t to resolve these mismatches.

Reconciliation builds M_t top-down, one layer at a time. Its root is the root objective of O'_t, which governs all of E_t. Expanding a node divides its governed subsequence among the node's children, and each division point is a *boundary*, the position in E_t where one child's subsequence ends and the next begins. To place these boundaries, LLM_{rec} compares how O'_t and P'_t split the subsequence already assigned to the node. The chosen boundaries fix the children and the subsequence each one governs, and reconciliation recurses into every child.

Boundaries follow the agreement between O'_t and P'_t. Where both place a boundary at the same point, M_t retains it. Where P'_t reads a subsequence as a *while* or *for-each* construct but O'_t splits it into several sub-objectives, M_t keeps the subsequence as one iterative phase and nests those sub-objectives as semantic refinements within it. Where P'_t reads it as a flat *sequence*, M_t keeps the temporal order and adopts the child boundaries from O'_t. Where neither view exposes internal structure yet the activities show aligned repetitions over named artifacts or repeated attempts under an unmet objective-state condition, LLM_{rec} infers a *for-each* or *while* operator from the trace. In each case LLM_{rec} splits, merges, or re-parents nodes until objective scope and control flow agree. The result is a unified task model M_t (Figure 2E) whose every node carries both an objective interpretation and a procedure operator, and whose every leaf is grounded in the activities of E_t.

Implementation. LLM_{seg}, LLM_{obj}, LLM_{proc}, and LLM_{rec} use `gpt-5.4` at temperature 1.0, and evaluations that rely on an LLM judge use `gpt-5.5` at temperature 1.0.

4 Intrinsic Evaluation

Datasets. We use two base datasets across all three evaluations. The first is the human computer-use dataset from Wang et al. (2025), comprising 38 recorded sessions spanning 15 tasks across five professional domains (data analysis, engineering,

computation, writing, and design), with multiple human users per task and each session labeled with its ground-truth task identity. The second is SkillsBench (Li et al., 2026), comprising 86 software engineering tasks executed by coding agents across three harnesses (Claude Code, Codex, and Gemini CLI) and three skill conditions (Appendix C), from which we select the 15 tasks with the highest average action count, yielding 195 runs. Each evaluation adapts these sources as described below.

4.1 Latent Task Induction Robustness

Real sessions interleave actions from many concurrent tasks, and a usable task model must recover each one without conflation. We measure how reliably induction separates interleaved actions into the correct tasks as task multiplicity and interleaving density grow.

Evaluation Protocol. To test robustness under varying degrees of task multiplicity and interleaving density, we construct synthetic multi-task trajectories from the Wang et al. (2025) sessions. For each task count $K \in \{2, 3, \ldots, 15\}$ and interleaving difficulty $d \in \{2, 3, 4\}$ (segments per session), we sample K tasks and merge their interleaved segments into a single composite trajectory, with three independent draws per (K, d) condition. Full construction details are in Appendix B.

Each trajectory is processed through the full activity abstraction and grounding stage (§3.1) followed by latent task induction (§3.2). The system produces a predicted task count $\hat{K}$ and an assignment of activities to predicted tasks. Task assignment quality is assessed via the Adjusted Rand Index (Hubert and Arabie, 1985) (ARI), which measures pairwise agreement between predicted and ground-truth partitions of activities after correcting for chance (range $[-1, 1]$, higher is better). Task count accuracy is measured by mean absolute error MAE $= |\hat{K} - K|$, where lower values indicate more precise task enumeration. Both metrics are averaged across all trajectories per (K, d) condition.

Induction stays robust under heavy interleaving. Table 1 reports ARI and MAE aggregated over all task counts $K \in \{2, \ldots, 15\}$ for each interleaving difficulty level. The system reaches 0.974 overall ARI, recovering non-contiguous task partitions even when activities are densely interwoven. MAE stays below 1 across all conditions, so the predicted task count tracks the true count as the number of concurrent tasks grows from 2 to 15. Appendix B.1 presents the error analysis.

Table 1: Latent task induction performance by interleaving difficulty d (number of segments per session), averaged over $K \in \{2, \ldots, 15\}$ and three trajectories per condition. The bottom row reports the overall mean.

d	ARI ↑	MAE ↓
2	0.980 ± 0.024	0.29 ± 0.34
3	0.975 ± 0.032	0.50 ± 0.60
4	0.968 ± 0.029	0.64 ± 0.61
Overall	0.974 ± 0.028	0.48 ± 0.54

4.2 Latent Task Induction Stability

The induced latent task set should reflect the actual task being performed and remain stable across different execution strategies. Since induction never observes the task label, agreement with the assigned task indicates that induced tasks capture task identity rather than execution idiosyncrasies.

Evaluation Protocol. We evaluate on both base datasets, each containing tasks executed across multiple runs by different human users or LLM agents that vary in capability and strategy. From Wang et al. (2025) we induce a task model from each of the 38 sessions. On SkillsBench, we evaluate sessions whose task outcome is marked successful, spanning different agent harnesses and skill conditions: no skill, curated skill, or self-generated skill supplied as a scaffold to the task agent.

For each run i, task model induction observes only the trajectory and produces an induced latent task set $\mathcal{T}_i$. The dataset's ground-truth assigned task y_i is withheld from induction and used only for evaluation. We give $\mathcal{T}_i$ and y_i to the LLM judge, which returns a binary label indicating whether the induced task set correctly captures the assigned task. We report the average match rate across runs.

Induced tasks track task identity, not execution idiosyncrasies. Table 2 reports match rates for both datasets. On human sessions, the induced task set matches the assigned task in 94.74% of runs, indicating that induction recovers the task identity without observing the dataset label. On successful SkillsBench runs, the overall match rate is 93.24%, with similar rates across most agent harnesses and skill conditions. The induced latent task set changes with the underlying assigned task rather than merely reflecting idiosyncratic execution traces.

Table 2: Latent task identity match rate. For each run, an LLM judge (gpt-5.5) receives the induced latent task set and the ground-truth assigned task, which is hidden during induction, and returns a binary match label. SkillsBench rows include only runs whose task outcome is marked successful.

Dataset	Subset	N	Match Rate ↑
Wang et al. (2025)	All	38	94.74
SkillsBench	All	74	93.24
	Agent: Claude Code	23	95.65
	Agent: Codex	19	84.21
	Agent: Gemini CLI	32	96.88
	Skill: none	21	95.24
	Skill: self-gen.	13	84.62
	Skill: curated	40	95.00

Table 3: Task model fidelity. Task is latent task recovery; Cov. is objective coverage; Coh. is objective coherence; P-Ch. is parent-child consistency; Desc. is step description accuracy; Op. is operator correctness. Task and Cov. are 5-point Likert scores (↑); remaining columns are binary pass rates (%,↑). N/A: dimension not applicable.

		Objective			Procedure	
Model	Task	Cov.	Coh.	P-Ch.	Desc.	Op.
Wang et al. (2025)	2.87	3.24	65.7	62.2	30.3	N/A
Direct gen.	3.63	4.00	87.0	97.3	23.4	52.7
Ours	**3.71**	4.34	85.7	92.6	**74.9**	**88.5**
Ours (joint)	3.66	**4.40**	**99.1**	**99.1**	56.0	70.3

4.3 Task Model Fidelity

Downstream use depends on whether sub-goal decompositions capture meaningful task phases and procedure steps faithfully describe the agent's execution. We assess the structural quality of the induced task model against human judgment and two baselines.

Baselines. We compare against two baselines. Workflow-summary uses the workflow summarization toolkit from Wang et al. (2025), which produces a phase-based narrative of each observed session without a formal task model schema or control-flow operators. Direct generation prompts GPT-5.4 with the complete task model schema alongside the observed activity trace and asks the model to produce a conformant task model in one pass. Both baselines use GPT-5.4 for generation. We further include an ablation of our method, Ours (joint), which induces the objective and procedure models in a single joint pass and merges them into a task model, rather than inducing each view independently and reconciling them. This ablation isolates the contribution of staged induction and reconciliation.

Evaluation Protocol and Metrics. We evaluate on all 38 Wang et al. (2025) sessions. For human validation, we randomly sample 20 sessions and assign each to two independent annotators who score the same rubric dimensions. We assess six rubric dimensions across three groups: latent task recovery and objective coverage rated on a 5-point Likert scale, and four per-node binary checks covering objective coherence, parent-child consistency, step description accuracy, and operator correctness. Full rubric definitions are in Appendix D. The LLM judge receives the task instruction, activity trace, and induced task model. The 20 sampled sessions are independently scored by two human annotators on the same rubric.

Task models stay faithful to procedures and observed objective decomposition. Table 3 reports results. Our method leads on both Likert dimensions and achieves substantially higher scores on procedure fidelity, with step description accuracy of 74.9% against 23.4% and 30.3% for the two baselines, and operator correctness of 88.5% against 52.7%. These gains suggest that grounding, constraint checking, and reconciliation between objectives and procedures contribute to better alignment between the induced procedure and the observed execution. Objective decomposition quality is comparable between our method and direct generation, while our method improves task recovery and coverage. The ablation attains better objective structure scores but recovers procedures markedly less faithfully. This validates the hypothesis that inducing the two views simultaneously forces structural misalignment, since the objective model and the procedure model are distinct modelings of the same execution and fitting them in a single pass distorts the procedure.

5 Extrinsic Evaluation

A useful task model should transfer beyond the demonstration it was induced from. We test whether an induced task model is an effective source for generating reusable skills that improve a downstream agent's accuracy on held-out tasks.

Experiment Protocol. We test this by using the induced task model as the source of learning for agent skill generation on SkillLearnBench (Zhong

Table 4: Extrinsic evaluation on SkillLearnBench. Each row is a source of learning fed into the same skill creator. Cov. is skill coverage; Exec. is executability; Safe. is safety; Align. is agent trajectory alignment; Acc. is held-out task accuracy. All values are percentages (↑). No skill produces no generated artifact, so the skill-quality columns are not applicable.

	Skill quality			Execution	
Source of learning	Cov.	Exec.	Safe.	Align.	Acc.
No skill	N/A	N/A	N/A	59.20	8.57
Raw demonstration	52.46	53.49	**92.52**	63.15	11.43
Wang et al. (2025)	54.07	59.35	91.74	66.18	14.29
Ours	**54.95**	**67.65**	90.57	**67.99**	**18.57**

et al., 2026), a benchmark for continual learning methods for agent skill generation on real-world tasks. SkillLearnBench contains 20 task families, each grouping multiple tasks of a similar nature that can be addressed with similar strategies. For each task family we induce a task model from a single successful demonstration of one instance and pass it to the skill creator of Codex,[1] which synthesizes a reusable skill. The skill is scored against the SkillLearnBench rubric and then deployed on held-out instances of the same family.

We compare three sources of learning, each fed into the same skill creator. Raw demonstration passes the grounded trace directly, the hierarchical workflow summary baseline (Wang et al., 2025) passes a phase-based summary of the demonstration, and *Ours* passes the task model. A *No skill* condition bounds performance without any generated skill. All skills are generated and all held-out instances executed with GPT-5-mini, so performance differences isolate the source of learning. We report five SkillLearnBench metrics: skill coverage, executability, safety, agent trajectory alignment, and held-out task accuracy.

The task model is an effective representation for skill transfer. Skills generated from our task models are more executable and transfer better to held-out tasks than skills generated from raw demonstrations or workflow summaries. As shown in Table 4, our model improves executability from 59.35 to 67.65 and held-out accuracy from 14.29 to 18.57 over the strongest baseline, a 30% relative accuracy gain.

[1]The skill creation component of the Codex CLI agent, https://github.com/openai/codex.

6 Related Works

Inducing representation from behavior Plan recognition infers goals from observed actions but presupposes a plan library or domain theory (Kautz et al., 1986), while process mining discovers procedural models from event logs that already contain typed activities and case identifiers (Van Der Aalst et al., 2011). Recent computer-use trace work grounds pixels into semantic operations (Shaw et al., 2023; Yang et al., 2023; You et al., 2024; Zheng et al., 2024), extracts action sequences or instructional steps (Chen et al., 2024; Zang et al., 2025), learns persistent user models (Shaikh et al., 2025, 2026), pretrains agents on demonstrations (Lu et al., 2025; Song et al., 2026), or analyzes trajectories under a known root task (Wang et al., 2024, 2025). We drop these assumptions, inducing semantic actions, latent tasks, objective hierarchies, and procedures jointly from raw screen-and-input traces.

Multitasking and interleaving Field and cognitive studies show that knowledge workers continuously interleave goals across working spheres and incur measurable costs when switching (Czerwinski et al., 2004; González and Mark, 2004; Mark et al., 2008; Iqbal and Horvitz, 2007; Salvucci et al., 2009), and event segmentation theory frames activity as hierarchically organized boundaries inferred from goal and state changes (Zacks and Tversky, 2001). Synthetic-trace studies further argue that latent intent must be modeled to make sense of interleaved activity (Yang et al., 2026). These findings ground our choice to treat task identity as latent and to allow non-contiguous task spans, so that interleaved activity is resolved during task induction rather than carried into the procedure model.

7 Conclusion

We introduced task model induction, which turns naturalistic computer-use traces into latent tasks with structured objective and procedure models. The pipeline abstracts raw events into activities, induces latent tasks, and reconciles separately induced objective and procedure models into a unified task model. Experiments show that this representation recovers interleaved execution faithfully and transfers better to downstream agent skills than raw traces or workflow summaries.

Limitations

Task model induction operates on naturalistic computer-use traces that may contain personally identifiable information. Future deployments that apply this method to uncurated traces should consider privacy redaction of screenshots and input events before induction, so that sensitive content does not propagate into induced artifacts distributed for downstream uses such as skill learning. Studying the effect of such redaction on induction quality is left to future work.

Ethics Statement

This work uses three publicly available datasets, which are all released for research use and contain no personally identifiable information. The synthetic multi-task trajectories constructed for the robustness evaluation are assembled by merging segments of these existing public sessions; no new data collection involving human subjects was conducted. Skill generation and agent evaluation are performed on benchmark tasks with no access to private user data. We do not foresee direct harms from this work.

References

Piotr D Adamczyk and Brian P Bailey. 2004. If not now, when? the effects of interruption at different moments within task execution. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 271–278.

Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371.

Yanting Chen, Yi Ren, Xiaoting Qin, Jue Zhang, Kehong Yuan, Lu Han, Qingwei Lin, Dongmei Zhang, Saravan Rajmohan, and Qi Zhang. 2024. Sharingan: Extract user action sequence from desktop recordings. *Preprint*, arXiv:2411.08768.

Mary Czerwinski, Eric Horvitz, and Susan Wilhite. 2004. A diary study of task switching and interruptions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 175–182.

Dan Diaper and Neville Stanton. 2003. The handbook of task analysis for human-computer interaction.

Victor M González and Gloria Mark. 2004. " constant, constant, multi-tasking craziness" managing multiple working spheres. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 113–120.

Michael Grohs, Han van der Aa, and Jana-Rebecca Rehse. 2024. Beyond log and model moves in conformance checking: discovering process-level deviation patterns. In *International Conference on Business Process Management*, pages 381–399. Springer.

Lawrence Hubert and Phipps Arabie. 1985. Comparing partitions. *Journal of classification*, 2(1):193–218.

Shamsi T. Iqbal and Eric Horvitz. 2007. Disruption and recovery of computing tasks: field study, analysis, and directions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, page 677–686, New York, NY, USA. Association for Computing Machinery.

Henry A Kautz, James F Allen, and 1 others. 1986. Generalized plan recognition. In *AAAI*, volume 86, page 5. Philadelphia, PA.

Aleksei N Leont'ev. 2024. The problem of activity in psychology. In *The concept of activity in Soviet psychology*, pages 37–71. Routledge.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, and 1 others. 2026. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*.

Dunjie Lu, Yiheng Xu, Junli Wang, Haoyuan Wu, Xinyuan Wang, Zekun Wang, Junlin Yang, Hongjin Su, Jixuan Chen, Junda Chen, Yuchen Mao, Jingren Zhou, Junyang Lin, Binyuan Hui, and Tao Yu. 2025. Videoagenttrek: Computer use pretraining from unlabeled videos. *Preprint*, arXiv:2510.19488.

Gloria Mark, Daniela Gudith, and Ulrich Klocke. 2008. The cost of interrupted work: more speed and stress. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 107–110.

Dario D. Salvucci, Niels A. Taatgen, and Jelmer P. Borst. 2009. Toward a unified theory of the multitasking continuum: from concurrent performance to task switching, interruption, and resumption. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, page 1819–1828, New York, NY, USA. Association for Computing Machinery.

Omar Shaikh, Shardul Sapkota, Shan Rizvi, Eric Horvitz, Joon Sung Park, Diyi Yang, and Michael S. Bernstein. 2025. Creating general user models from computer use. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*, UIST '25, New York, NY, USA. Association for Computing Machinery.

Omar Shaikh, Valentin Teutschbein, Kanishk Gandhi, Yikun Chi, Nick Haber, Thomas Robinson, Nilam Ram, Byron Reeves, Sherry Yang, Michael S Bernstein, and 1 others. 2026. Learning next action predictors from human-computer interaction. *arXiv preprint arXiv:2603.05923*.

Peter Shaw, Mandar Joshi, James Cohan, Jonathan Berant, Panupong Pasupat, Hexiang Hu, Urvashi Khandelwal, Kenton Lee, and Kristina N Toutanova. 2023. From pixels to ui actions: Learning to follow instructions via graphical user interfaces. *Advances in Neural Information Processing Systems*, 36:34354–34370.

Chan Hee Song, Yiwen Song, Palash Goyal, Yu Su, Oriana Riva, Hamid Palangi, and Tomas Pfister. 2026. Watch and learn: Learning to use computers from online videos. *Preprint*, arXiv:2510.04673.

Wil Van Der Aalst, Arya Adriansyah, Ana Karla Alves De Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose, Peter Van Den Brand, Ronald Brandtjen, Joos Buijs, and 1 others. 2011. Process mining manifesto. In *International conference on business process management*, pages 169–194. Springer.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2024. Agent workflow memory. *arXiv preprint arXiv:2409.07429*.

Zora Zhiruo Wang, Yijia Shao, Omar Shaikh, Daniel Fried, Graham Neubig, and Diyi Yang. 2025. How do ai agents do human work? comparing ai and human workflows across diverse occupations. *arXiv preprint arXiv:2510.22780*.

Jeannette M. Wing. 2006. Computational thinking. *Commun. ACM*, 49(3):33–35.

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. 2023. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v. *arXiv preprint arXiv:2310.11441*.

Ruozhen Yang, Yucheng Jiang, Yueqi Jiang, Priyanka Kargupta, Yunyi Zhang, and Jiawei Han. 2026. Grounding agent memory in contextual intent. *arXiv preprint arXiv:2601.10702*.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. 2024. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *European Conference on Computer Vision*, pages 240–255. Springer.

Jeffrey M Zacks and Barbara Tversky. 2001. Event structure in perception and conception. *Psychological bulletin*, 127(1):3.

Yuan Zang, Hao Tan, Seunghyun Yoon, Franck Dernoncourt, Jiuxiang Gu, Kushal Kafle, Chen Sun, and Trung Bui. 2025. Ms4ui: A dataset for multi-modal summarization of user interface instructional videos. *arXiv preprint arXiv:2506.12623*.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. 2024. Gpt-4v (ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*.

Shanshan Zhong, Yi Lu, Jingjie Ning, Yibing Wan, Lihan Feng, Yuyi Ao, Leonardo FR Ribeiro, Markus Dreyer, Sean Ammirati, and Chenyan Xiong. 2026. Skilllearnbench: Benchmarking continual learning methods for agent skill generation on real-world tasks. *arXiv preprint arXiv:2604.20087*.

A Structural Validity Constraints

Both O'_t and P'_t are subject to formal validity constraints that induction must satisfy. For the objective tree, every activity in E_t must appear under exactly one leaf node (completeness and non-overlap); a node spanning a single activity has reached the level of a local objective and must remain a leaf; and every internal node must state a desired outcome rather than an interface action. For the procedure tree, every activity in E_t must appear in at least one procedure node's reference set, every operator must belong to the closed primitive set, every for-each node must bind its iteration variable to an explicitly enumerated collection, and every while node must state an objective-state exit condition. A loop body is an abstract template whose every step maps to the activity episodes it covers across all repetitions, so a loop is admitted only when grounded in recurring evidence rather than asserted. Constraint violations are identified by a deterministic validator and fed back as structured feedback, prompting targeted repair, after which a recovery pass re-examines flat sequences for repeated bodies that should have been folded into a for-each or while node.

B Latent Task Induction Robustness

Dataset statistics. The 38 recorded sessions in Wang et al. (2025) average 1,282 raw keyboard and mouse actions and 1.01 hours of active computer use per session. Session duration is computed after removing idle intervals exceeding ten minutes between consecutive actions, treating such gaps as disengagement rather than active task work.

Construction. Synthetic interleaving trajectories are constructed from these 38 sessions. For each task count $K \in \{2, 3, \ldots, 15\}$ and interleaving difficulty $d \in \{2, 3, 4\}$, we sample K distinct tasks without replacement and assign each a single randomly selected session. The session for each task is then partitioned into d contiguous segments by sampling $d - 1$ split points uniformly at random from positions satisfying the constraint that every resulting segment contains at least ten activities. This constraint prevents configurations in which a task occupies only a trivially short span, which would not reflect realistic interleaving patterns in naturalistic computer use. The $K \times d$ segments are then randomly permuted to form the composite trajectory; within each segment, the original event ordering is preserved. Three independent trajectories are generated per (K, d) condition for a total of $14 \times 3 \times 3 = 126$ synthetic trajectories across all conditions. Table 5 reports the full per-condition results.

B.1 Error Analysis

We manually sample and code 1,000 predicted decision boundaries produced across all 126 synthetic trajectories. Table 6 reports the distribution of label-level failure patterns; 87.1% of labels are correctly assigned or outside the scope of any named failure type. The remaining 12.9% split across four structural causes described below, along with potential mitigations.

Subgoal promotion (6.8%). A coherent phase within a benchmark task, such as data cleaning before analysis or asset export before presentation, is predicted as a separate root task, inflating $\hat{K}$. Among the 68 affected labels, 33.8% involve real user navigation to personal activities that are unrelated to the assigned benchmark task; these cases are in principle unresolvable because the trajectory contains no task-specific context for the off-task behavior.

Recorder-induced clustering (2.5%). The screen-recording tool used in the original data collection generates setup, screenshot, and teardown operations that are interleaved across multiple benchmark tasks and carry no task-specific context. These activities accumulate into spurious clusters that the system treats as independent tasks.

Cross-task workspace sharing (2.4%). The dataset spans 15 tasks across five professional domains, so multiple tasks within the same domain share the same tools and output artifacts, for example two data analysis tasks both editing the same Jupyter notebook, or two design tasks both working in Figma. When such tasks are interleaved, their activity streams are very similar by tool or interface alone. The primary mitigation is to attend more to artifact-level unique identifiers and screen content, using both textual and visual signals to distinguish tasks that share an interaction surface.

C SkillsBench Trajectory Conversion

Skill conditions. A *skill* in SkillsBench is a reusable, human-authored procedural document that an agent may consult when approaching a task. Runs are executed under one of three conditions:

no supplementary skill (*no-skill*), a human-curated skill (*skill*), or a skill the agent generates for itself (*self-generation*).

Trajectory format. Coding agent trajectories from SkillsBench differ structurally from the low-level keyboard and mouse streams found in human computer-use recordings. Each step consists of an assistant message that issues one or more tool calls (e.g., `bash`, file reads, or web searches) followed by corresponding tool results. These operations are already semantic rather than raw input events, so they do not pass through the activity abstraction stage of our pipeline. Instead, we map each (tool call, tool result) pair directly to one activity in our schema, bypassing the low-level action grounding step.

Pairing strategy. Consecutive assistant turns can issue multiple tool calls in a single burst before any results arrive, a pattern common when Claude Code issues parallel independent commands. We apply a burst-aware last-M assignment rule in which a burst of N tool calls followed by only $M \leq N$ results maps the first $N - M$ calls to empty output (capturing write-style commands that produce no meaningful stdout) and the remaining M calls are paired with results in FIFO order. When $M = N$ all calls are matched one-to-one. User-role messages and assistant text-only reasoning steps carry no tool calls and are dropped; orphaned tool-result steps with no preceding burst are also discarded.

Dataset statistics. Applying this conversion to the 195 selected SkillsBench runs (15 tasks, 13 runs per task) yields 9,459 activities in total, averaging 48.5 activities per run (median 41; range 8–237). The wide range reflects task heterogeneity across tasks spanning security patching, performance debugging, scientific computation, and multimedia processing.

D Task Model Fidelity Rubric

The LLM judge and human annotators use the same three rubrics below. Each evaluation call receives the task instruction, activity trajectory, and induced task model, then returns scores for one rubric group. The judge is instructed to use only evidence from the provided inputs and to return structured JSON matching the schema.

Rubric 1 | Latent Task Recovery (scored 1–5, ↑)

Evaluates whether the root objective of the induced task model correctly identifies the latent task the agent was pursuing.

- **5** Correctly identifies the task intent, deliverable, and abstraction level. Specific enough to distinguish the task but not tied to incidental tools or local actions.
- **4** Mostly correct, with a minor issue in specificity, abstraction, or a secondary deliverable.
- **3** Partially correct. Captures the broad goal or domain but misses an important part of the task intent, deliverable, or scope.
- **2** Weak match. Describes only a local phase, a tool action, or a vague activity rather than the full latent task.
- **1** Incorrect or unsupported. Describes the wrong task, hallucinates intent, or is not grounded in the trace.

Rubric 2 | Subgoal Decomposition Quality

Coverage score (1–5, ↑): evaluates whether the subgoals span all major phases of the observed task.

- **5** All major phases of the task are represented by subgoals.
- **4** Most major phases are represented; only minor phases are missing or merged.
- **3** Some important phases are represented, but at least one major phase is missing or poorly covered.
- **2** Decomposition captures only a small part of the task or uses very coarse/noisy phases.
- **1** Subgoals do not meaningfully cover the trace.

Binary checks (per non-root objective node, ↑):

Coherent subgoals	Each subgoal groups activities serving one recognizable intermediate objective.
Parent-child consistent	Child subgoals collectively advance the parent objective.

Rubric 3 | Procedure Fidelity

Binary checks (per procedure node):

Operator correct (↑)	The control-flow operator is supported by the trace.
Step descriptions accurate (↑)	Step descriptions faithfully summarize what occurred.

Operator decision rules:

`SEQ`	Steps occur in the stated temporal order.
`FOR`	The trace shows the same operation applied to multiple items.
`WHILE`	The trace shows repeated attempts or checks until a stopping condition.
`CHOICE`	The trace shows selecting one option among alternatives.

If an operator is plausible but not directly evidenced, mark *operator correct* as false.

E Implementation Details and Pipeline Prompt Templates

This appendix lists the fixed instruction templates used by each LLM call in the pipeline. The two action-grounding prompts (Listing 3) are used by VLM_{grnd} described in Section 3.1. The backward semantic-action segmentation prompt (Listing 4) and the activity segmentation prompt (Listing 5) are both used by LLM_{seg} from the same section. The root-thread discovery and consolidation prompts (Listings 6 and 7) are used by the two-phase latent task induction in Section 3.2. The objective-model, procedure-model, and reconciliation prompts (Listings 8, 9, and 10) are used by LLM_{obj}, LLM_{proc}, and LLM_{rec} from the task model induction subsection.

Table 5: Full per-condition results for latent task induction robustness. Each cell reports mean ± std over three trajectories.

K	d	ARI ↑	MAE ↓
2	2	1.000±0.000	0.00±0.00
	3	1.000±0.000	0.00±0.00
	4	0.994±0.009	0.33±0.47
3	2	1.000±0.000	0.00±0.00
	3	1.000±0.000	0.00±0.00
	4	0.902±0.046	0.67±0.47
4	2	0.983±0.024	0.67±0.94
	3	0.998±0.003	0.33±0.47
	4	0.997±0.004	0.00±0.00
5	2	1.000±0.000	0.00±0.00
	3	1.000±0.000	0.00±0.00
	4	0.972±0.040	0.33±0.47
6	2	0.998±0.003	0.00±0.00
	3	0.987±0.018	0.33±0.47
	4	0.934±0.093	1.00±1.41
7	2	1.000±0.000	0.00±0.00
	3	1.000±0.000	0.67±0.47
	4	1.000±0.000	0.00±0.00
8	2	0.920±0.113	0.00±0.00
	3	0.945±0.077	0.33±0.47
	4	0.984±0.011	1.00±0.82
9	2	0.972±0.040	0.00±0.00
	3	0.997±0.005	0.33±0.47
	4	0.948±0.059	1.00±0.00
10	2	0.951±0.065	0.33±0.47
	3	0.938±0.076	0.67±0.47
	4	0.947±0.074	0.00±0.00
11	2	0.960±0.056	0.33±0.47
	3	0.997±0.004	0.33±0.47
	4	0.966±0.013	2.00±1.41
12	2	0.980±0.024	1.00±1.41
	3	0.991±0.013	0.33±0.47
	4	0.972±0.028	1.00±0.82
13	2	0.990±0.010	0.67±0.47
	3	0.923±0.044	2.33±1.25
	4	0.999±0.002	0.00±0.00
14	2	0.999±0.001	0.33±0.47
	3	0.962±0.045	0.33±0.47
	4	0.984±0.007	1.33±0.47
15	2	0.962±0.033	0.67±0.47
	3	0.917±0.081	1.00±0.82
	4	0.952±0.035	0.33±0.47

Table 6: Label-level failure distribution across 1,107 predicted task labels from 126 synthetic trajectories.

Failure pattern	Labels	%
Correct / not failure-relevant	871	87.1
Subgoal promotion	68	6.8
Recorder-induced clustering	25	2.5
Cross-task workspace sharing	24	2.4
Others	12	1.2

```
GOAL_SYSTEM_PROMPT = """
Infer the immediate intent of one computer action.

Use the action string, the screenshot captured at the action moment, and the optional after screenshot. The before/action
screenshot is primary evidence; use the after screenshot only to disambiguate what changed.

Return one concise sentence for the `goal` field.

Rules:
- Describe the local UI operation, not the user's broader task.
- Prefer concrete visible targets: button names, menu items, fields, files, tabs, cells, links, commands, or text snippets.
- Include the action verb when it matters, such as click, drag, type, select, open, close, scroll, or submit.
- Do not invent hidden motivations or off-screen content.
- If the visible evidence is insufficient, return the best grounded statement and mark uncertain details as "not sure".
"""

CONTEXT_SYSTEM_PROMPT = """
Ground one computer action in visible UI context.

Use the action string, the screenshot captured at the action moment, any zoomed-in crops, and the optional after screenshot.
Zoom crops are centered on the action coordinates; a red outline or marker indicates the likely target region.

Return:
- `active_application`: application name plus visible window, page, document, file, or tab title when readable.
- `visual_content`: the specific visible artifact the action is aimed at or the user's eyes are likely focused on.

Rules:
- Do not output the goal; only output application/context fields.
- For `active_application`, prefer formats like "Google Chrome - Page title", "VS Code - filename.py", "Terminal - shell
session", or "not sure".
- For `visual_content`, name the exact visible control/content region when possible: button, menu item, field, selected text,
file row, cell, chart, code line, terminal command, tab, or document section.
- Ground every detail in visible text, recognizable UI, the action coordinate, or the before/after change.
- If a field is not clearly visible, return "not sure" for that field.
"""
```

Figure 3: Action grounding prompts for immediate intent and visible UI context.

```
BACKWARD_SEMANTIC_ACTION_SEGMENT_PROMPT = """
You are analyzing a user's computer workflow by looking at actions in REVERSE order from the end of the session backward.

KEY INSIGHT: later outcomes help explain earlier low-level actions.

Each segment should be one candidate atom semantic action.

{semantic_action_definition}

=== WHAT HAPPENS AFTER THESE ACTIONS ===
{future_context}

=== ACTIONS TO ANALYZE (chronological order, index 0 = earliest) ===
{actions_list}

=== TASK ===
Segment these low-level actions into atom semantic actions.

For each group output:
1. semantic_action: one concise sentence describing the intentional operation

Semantic-action rules:
- Use semantic, operation-level language.
- Prefer the immediate operation over the broader task objective.
- Keep concrete apps, clicks, typing, commands, URLs, files, and navigation out of semantic_action unless essential.
- Avoid semantic actions that start with purely mechanical verbs like click, scroll, focus, move, hover, drag, or wait unless
that operation is itself the meaningful user action.
- Do not skip failed attempts or corrections; include them with the operation they are trying to complete when intent is
unchanged.

Coverage rules:
- Groups must be consecutive indices with no gaps, overlaps, or reorder.
- Every index from 0 to {max_idx} must appear exactly once.
- Split when the range contains multiple intentional operations.
- Keep together repeated low-level interaction needed to finish the same operation.

Output ONLY valid JSON:
{
  "groups": [
    {"start_idx": <int>, "end_idx": <int>, "semantic_action": "<atom semantic action>"}
  ]
}

List groups in REVERSE chronological order, latest group first. start_idx and end_idx are inclusive.
"""
```

Figure 4: Backward semantic-action segmentation prompt.

```
SEGMENTATION_PROMPT = """
You segment chronological atom semantic actions into activities.

{definition}

=== WHAT HAPPENED BEFORE THIS BATCH ===
{prior_context}

=== SEMANTIC ACTIONS TO SEGMENT (chronological order, index 0 = earliest in this batch) ===
{actions_list}

=== TASK ===
Partition the current batch into contiguous activities.

For each segment output:
1. start_idx and end_idx, inclusive, using the batch-local indices.
2. objective: one concise, self-contained phrase/sentence naming the intended local outcome or intermediate state. Include the
concrete target, artifact, person, project, channel, file, URL, or app needed to interpret the activity. Do not list the
procedure.
3. additional_context: one to three concise sentences with the observed procedure and concrete evidence needed to understand
that objective.

Coverage rules:
- Every index from 0 to {max_idx} must appear exactly once.
- Segments must be consecutive with no gaps, overlaps, or reordering.
- Keep scaffolding actions with the objective they enable when evidence supports it.
- If an activity appears to continue across a batch boundary, produce the best segment inside this batch; a later merge pass
will join adjacent segments.

Output ONLY valid JSON:
{
  "segments": [
    {"start_idx": <int>, "end_idx": <int>, "objective": "<activity>", "additional_context": "<concise evidence-grounded context
    >"}
  ]
}
"""
```

Figure 5: Activity segmentation prompt over atom semantic actions.

```
ROOT_THREAD_DISCOVERY_PROMPT = """
You are building a task thread forest from chronologically ordered LEAF latent tasks.

Each leaf is already a local task. Your job is to attach each leaf to a durable ROOT thread.

Core mental model:
- Ask: "Which long-running top-level objective is this leaf advancing right now?"
- Roots represent durable objectives / deliverables, not contiguous time blocks.
- Leaves may interleave across roots.
- A root can pause and later resume.

Critical rules:
- Objective continuity beats adjacency.
- Two adjacent leaves can belong to different roots.
- Two distant leaves can belong to the same root.
- Different apps do NOT imply different roots.
- Interruptions do NOT imply different roots.
- Create a NEW root only when a genuinely new durable objective appears.
- Prefer a small number of strong roots over many near-duplicates.
- Communication leaves belong to the root defined by the SUBJECT of the message, not to a generic communication root.
- Setup, debugging, repo inspection, and environment preparation should stay under the same root as the later deliverable if
they are clearly in service of that deliverable.

Available existing roots:
{existing_roots}

Most recent assigned leaves before this batch:
{recent_context}

Current leaves to assign:
{leaf_batch}

Task:
1. Reuse an existing root whenever the leaf advances the same durable objective / deliverable.
2. Create a new root only when needed.
3. New roots created inside this batch can be referenced by later leaves in the same batch.
4. After assigning leaves, update each touched root's label/objective/summary/last_update/anchor so future batches can judge
fit.

Output ONLY valid JSON with `new_roots`, `assignments`, and `root_updates`.
"""
```

Figure 6: Latent task/root-thread discovery prompt.

```
ROOT_THREAD_CONSOLIDATION_PROMPT = """
You are consolidating provisional durable root threads into the final task thread forest.

Each provisional root was discovered from chronological leaves. Some provisional roots may actually belong to the SAME durable
objective and should be merged.

Core rules:
- Merge provisional roots if they advance the same long-running deliverable / objective, even if they are far apart in time,
use different apps, or are interrupted.
- Keep roots separate if they represent genuinely different durable objectives.
- Objective continuity beats adjacency.
- Tiny opportunistic one-off roots may be absorbed into a nearby substantive root if they do not establish an independent
durable objective.
- Prefer a compact set of strong canonical roots.
- Early setup/debugging/investigation roots should be merged into the later product root when they clearly enable that same
deliverable.
- Communication-heavy provisional roots should be merged based on what the messages are ABOUT, not merely because they happen
in the same messaging tool.

Provisional roots:
{provisional_roots}

Output ONLY valid JSON:
{
  "canonical_roots": [
    {
      "canonical_root_id": "C1",
      "label": "<short human-readable root label>",
      "objective": "<durable top-level objective>",
      "deliverable": "<artifact/state this root advances>",
      "success_criteria": "<observable completion criteria>",
      "member_root_ids": ["R001", "R004"]
    }
  ]
}
"""
```

Figure 7: Latent task/root-thread consolidation prompt.

```
GENERATION_PROMPT_TEXT = """
You induce a hierarchical objective model from task activity observations using computational thinking and recursive
decomposition.

The input contains activity segments describing WHAT A USER DID. Your job is to abstract over those actions and recover the
hierarchy of SUB-GOALS they were pursuing -- the recursive decomposition of the task into what needs to be accomplished at each
 level.
Return only a valid JSON object.

Objectives follow the computational thinking paradigm of recursive decomposition:
- Each node states a SUB-GOAL: what needs to be accomplished at this level to advance the parent goal.
- Write as a goal to be achieved -- not as a low-level procedure and not as a passive state predicate.
- Objectives must be TOOL-AGNOSTIC and USER-AGNOSTIC.
- Save procedural and evidential details for the `summary` field, NOT the `objective` field.

Granularity rules:
- A child node must represent a sub-outcome that is a necessary precondition or component of the parent outcome.
- If the input already represents one coherent atomic success state, use "decomposition": [] instead of inventing procedural
children.
- A node covering exactly one activity/subgoal segment must not have decomposition.

Required output schema, recursively:
{
  "id": "<stable hierarchical id, e.g. C1 or C1.1>",
  "objective": "<sub-goal: what needs to be accomplished at this level, tool-agnostic and user-agnostic>",
  "summary": "<brief evidence-grounded summary; may reference specific tools, files, or actions observed>",
  "subgoal_segments": ["<single integer id such as 16 or closed integer range string such as 16-23>"],
  "decomposition": [<child nodes with the same schema> or <empty list if no further decomposition is needed>]
}
"""
```

Figure 8: Objective-model induction prompt.

```
PROCEDURE_GENERATION_PROMPT_TEXT = """
You induce a procedure model from a task-thread objective JSON by applying the Structured Programming Theorem (Bohm-Jacopini,
1966).
Return only a valid JSON object.

The Structured Programming Theorem proves that any computable procedure can be expressed using exactly three control constructs
:
1. Sequence -- steps executed one after another in order.
2. Selection -- a choice between mutually exclusive alternative paths.
3. Iteration -- a body repeated either over a named collection (for-each) or until a condition holds (while).

Your task is to decompose the observed activity trace into a tree built from these constructs. Every branching, repetition, or
iteration in the trace must be captured by the appropriate construct -- not collapsed into a flat sequence.

The four operators are the complete and closed vocabulary:
- `SEQ`: steps in fixed order with no repetition and no branching.
- `FOR`: the same procedure body is applied to each member of a named, enumerable collection.
- `WHILE`: a body is repeated until an observable objective-state condition is satisfied.
- `CHOICE`: mutually exclusive alternative paths to the same goal; only one branch is observed.

FOR and WHILE bodies are ABSTRACT TEMPLATES -- they describe what happens per item / per pass using named steps with `name`, `
description`, and `activity_refs`. Do NOT place activity_id leaves inside a FOR or WHILE body.

Coverage rules:
- Every activity episode in the input must appear in at least one node's `activity_refs` or as an inline `activity_id` leaf.
- Composite nodes cover the union of their children's episodes.
- Prefer one primary owning node per episode.

Required output schema:
{procedure_output_schema_text()}
"""
```

Figure 9: Procedure-model induction prompt.

```
RECONCILIATION_GENERATION_PROMPT = """
You produce a unified task model by reconciling independently-induced objective and procedure models for the same activity
trace. Return only a valid JSON object.

What you produce:
A unified tree where each node has two layers:
- Objective layer: a domain-specific program in natural language. Captures domain invariants: required outcomes, correctness
constraints, and orderings that hold regardless of who executes the task or what specific inputs are used.
- Procedure layer: the faithful record of how the work was actually carried out in the observed trace, including failures and
corrections.

Both layers are determined jointly from both input models. Neither is authoritative alone.

Inputs:
- `source`: task-thread JSON with an `activities` list.
- `objective_model`: hierarchical objective model.
- `procedure_model`: control-flow procedure model.

Reconciliation:
- Both models draw the same boundary: honor it.
- Procedure shows WHILE or FOR across a range: strong structural signal; that range is one iterative phase.
- Procedure shows only a flat SEQ: weak structural signal; defer to the objective model's semantic clusters.
- Objective shows a clear semantic phase shift: supports a new boundary even when the procedure model draws a continuous SEQ.
- Neither model shows structure for a range: scan the source `activities` for FOR or WHILE patterns.

Coverage and ID rules:
- Every source activity must appear in at least one node's `activity_refs`.
- A node's `activity_refs` is the union of its children's `activity_refs` or its body-step `activity_refs`.
- Use compact ranges: `activity_NNNN` or `activity_NNNN-activity_MMMM`.
- Root id is the task-thread id; children are numbered sequentially.

Required output schema:
{unified_schema_text()}
"""
```

Figure 10: Bidirectional objective/procedure reconciliation prompt.