INFERRING HIDDEN AGENTIC LOGIC FROM TRACES FOR ONLINE FAILURE ATTRIBUTION AND WORKFLOW OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent execution consists not only of explicit actions at each step, but also of implicit structural shifts in hidden states. Yet the behavioral logic that governs these states and their transitions remains largely opaque. We introduce HALT, a framework for recovering this hidden structure and its cross-run topology from execution traces using hidden Markov models (HMMs). HALT models observable atomic actions as emissions from latent workflow states, while state transitions capture the underlying logical structure of agent behavior. Fitted on historical traces, these HMMs provide a unified view for understanding, evaluating, and improving agents. We evaluate HALT on two downstream applications. For failure attribution, HALT trains separate HMMs on successful and failed traces, and attributes each step by contrasting its likelihood under the two models. This enables real-time, step-level failure localization without requiring access to the complete trace, and naturally supports early stopping. On Who&When, HALT achieves 37.0% step-level attribution accuracy, outperforming a fine-tuned LLM-based attributor that reads the entire trace (32.1%), while requiring no LLM calls and observing only about 27% of the trace. For workflow optimization, existing methods typically rely on manually specified building blocks to construct and refine workflows during search. HALT instead extracts reusable workflow operators from HMMs fitted to historical successful traces, turning latent structural patterns into automatically discovered building blocks for optimization. Workflows found by searching over these HMM-derived operators outperform both hand-designed and automatically searched baselines across all four tasks. On MBPP Pro with GLM-5.1, HALT reaches 86.2%, compared with 73.6% for the best baseline, while also incurring lower total cost. Together, these results position HALT as a lightweight structural primitive for inferring hidden agentic logic from traces and leveraging that structure for both online failure attribution and workflow optimization.

1 INTRODUCTION

Large language model (LLM) agents have demonstrated strong capabilities in complex tasks, including mathematical problem solving (Gou et al., 2024; Hendrycks et al., 2021), software engineering (Yang et al., 2024; 2026; Yu et al., 2025), and web navigation (He et al., 2024). Their execution may interleave reasoning and action, as in ReAct (Yao et al., 2023), or separate planning from execution, as in plan-and-execute architectures (Kim et al., 2024). Each execution is captured as a trace, the ordered sequence of an agent's actions and the feedback they elicit from its environment. Underlying these explicit traces is a higher-level behavioral structure, namely the states an agent traverses and the transitions between them. For example, a coding agent (Yang et al., 2024) may cycle through Planning → Implementation → Verification (Yang et al., 2026; Cho et al., 2026).

However, this structure remains largely implicit (Chang et al., 2026; Cho et al., 2026), since observable actions alone do not uniquely identify their associated states. A single state can produce many different actions, and the same action can be produced by different states. Reading a file, for instance, may serve to explore the codebase in one phase or to verify a previously implemented fix in another. Uncovering these behavioral patterns across runs is a key challenge, yet a valuable one, since they provide a basis for understanding, evaluating, and improving agents (Cho et al., 2026).

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.

To address this challenge, we introduce HALT (**H**idden **A**gentic **L**ogic from **T**races), a framework that models agent execution from a hidden Markov perspective (Rabiner, 1989; Dai et al., 2025; Chang et al., 2026). In this formulation, each latent state is a behavioral phase such as planning or verification, and the emission distribution of a state records which atomic actions, such as tool calls, that phase tends to produce. State transitions record how phases follow one another and depend only on the preceding state. Learning these distributions from execution traces lets HALT construct a shared behavioral topology across runs, in which emissions tell what each phase does and transitions tell how control flows between phases. This modeling framework provides a unified basis for analyzing and improving agent behavior. In this work, we demonstrate two downstream applications, namely online failure attribution (Zhang et al., 2025b; 2026; Shen et al., 2026) and automated workflow optimization (Hu et al., 2025; Zhang et al., 2025a; Zhao et al., 2026a).

The first application is *failure attribution*, which identifies the decisive error step and the responsible agent in a failing execution. Existing approaches rely on prompted or fine-tuned LLMs to analyze completed traces or judge successive steps (Zhang et al., 2025b; 2026). Full-trace analysis delays feedback until execution ends, while stepwise judging requires repeated LLM inference. HALT instead captures failed and successful execution dynamics in its learned transition and emission distributions. At inference time, it scores each incoming step by contrasting its likelihood under these dynamics, conditioned only on the observed prefix. This enables online attribution and real-time alarms without LLM calls or access to the complete trace.

The second application is *automated workflow optimization*, which searches for a workflow that performs well on a task, guided by reusable building blocks called *operators*, such as self-refine or ensembling. Existing methods either depend on hand-written operators, as in AFlow (Zhang et al., 2025a), or rely on an LLM meta-agent that iteratively generates agent programs informed by an archive of previous designs, as in ADAS (Hu et al., 2025). The former is not fully automated, and the latter repeats LLM synthesis at every search iteration. HALT instead recovers operators from successful traces. Each operator corresponds to a high-probability subgraph of the learned behavioral topology, whose emission signature provides the evidence for an LLM to synthesize its prompt once, offline. The resulting operator bank is reused at each search expansion, eliminating the need for hand-written operators or repeated operator synthesis during search.

HALT is both accurate and efficient. For failure attribution, it runs online without LLM calls, yet surpasses AgenTracer (Zhang et al., 2026), a fine-tuned LLM attributor that reads the complete trace, on Who&When (Zhang et al., 2025b) (70.1% vs. 63.8% agent-level and 37.0% vs. 32.1% step-level accuracy) after observing only about 27% of a trace on average. For workflow optimization, HALT-induced workflows outperform AutoGen, ADAS, and AFlow (Wu et al., 2024; Hu et al., 2025; Zhang et al., 2025a) on all four tasks with both GLM-5.1 (GLM-5-Team, 2026) and DeepSeek-V4-Pro (DeepSeek-AI, 2026), and with GLM-5.1 they beat the strongest baseline by 12.6 points on MBPP Pro and 5.0 points on HumanEval Pro (Yu et al., 2025), showing that operators learned from traces can beat hand-written ones. Together, these results show that a learned behavioral topology is an effective basis for both online failure diagnosis and automated workflow construction.

Our contributions.

1. To the best of our knowledge, HALT is the first to model agent execution traces with hidden Markov models for online failure attribution and for automatic operator discovery in workflow optimization, recovering shared behavioral logic across runs.
2. Leveraging this structure, HALT enables both *online failure attribution*, without complete traces or LLM calls at inference time, and *automated workflow optimization*, with operators automatically derived from traces rather than drawn from a hand-designed operator bank.
3. Extensive experiments on two failure attribution benchmarks and four workflow optimization benchmarks demonstrate HALT's effectiveness and efficiency, showing consistent improvements over LLM-based attribution baselines and efficient discovery of high-performing workflows.

2 PRELIMINARIES: HIDDEN MARKOV MODELS

A hidden Markov model (HMM) describes an observation sequence $o_{1:T}$ generated by latent states $S_{1:T}$ (Rabiner, 1989; Dai et al., 2025). The states follow a Markov chain, meaning that states preceding the current state have no influence on the future except through the current state. We consider time-homogeneous HMMs with a finite state space, $\mathcal{S} = \{1, \ldots, M\}$, and a finite observation space, $\mathcal{O} = \{1, \ldots, L\}$. Such a model is fully specified by the parameter triple $\theta = (\pi, A, B)$, comprising the initial distribution $\pi \in \mathbb{R}^M$, where $\pi_j = \Pr(S_1 = j)$; the transition matrix $A \in \mathbb{R}^{M \times M}$, where $A_{ij} = \Pr(S_t = j \mid S_{t-1} = i)$; and the emission matrix $B \in \mathbb{R}^{M \times L}$, where $B_{ju} = \Pr(o_t = u \mid S_t = j)$.

3 METHODOLOGY

3.1 HIDDEN MARKOV MODELING OF AGENT TRACES

An agent's trace is a time-ordered record of the actions emitted during execution. We view this sequence as the visible surface of a latent control process: at each step, the agent is in an unobserved workflow phase, and that phase emits one observable action atom. Transitions between hidden phases capture how the agent moves through its workflow, while emissions connect each phase to the actions likely to appear in the trace. This makes agent execution naturally suited to an HMM formulation, as illustrated in Figure 1. To model a trace as an HMM, we first define the observations and hidden states, and then estimate the model parameters.

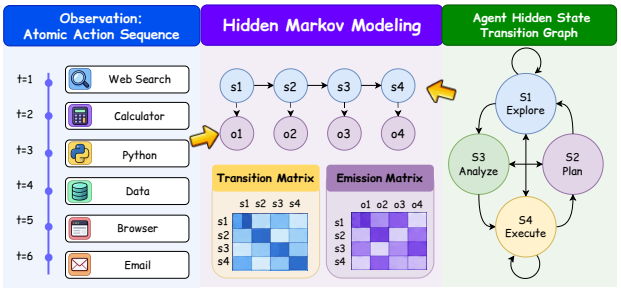


Figure 1: Overview of HALT. We model agent execution at two levels, namely explicit actions and the hidden behavioral state transitions that organize them.

Observation (Atomic Actions). The observation of step t is a single atom $o_t \in \mathcal{O}$, representing the primary atomic action emitted by that step, such as a tool call or function invocation. The vocabulary $\mathcal{O} = \{1, \ldots, L\}$ contains the L atoms observed in the training corpus. Agent traces come in heterogeneous formats, so we apply three extraction rules, in priority order, to normalize the primary action atom, namely (1) *tool calls*, where the action is the function name if a message contains a tool-call field; (2) *action tags*, where the action is the action label if the content contains an `[ACTION]` description; and (3) *command extraction*, for agents using code blocks, where we take the first command token and map it to a semantic category. If no rule matches, the action defaults to `role:content_type` (e.g., `assistant:text`). This extraction is deterministic and format-specific. Appendix C details the mapping for each dataset.

Hidden Workflow State. Each latent state $S_t \in \mathcal{S}$ represents an unobserved workflow phase that governs which actions are likely to be emitted at time t. The transition matrix A encodes the hidden logic of phase-to-phase control, while the emission matrix B captures the observable actions characteristic of each phase. Since each step emits one observation atom, the emission probability under state s is simply $p_B(o_t \mid s) = B_{s,o_t}$.

Estimation of Parameters. Given an observation sequence $o_{1:T}$ and the fixed number M of latent workflow states, we learn the HMM parameters $\theta = (\pi, A, B)$ via the Baum-Welch algorithm (Baum et al., 1970; Rabiner, 1989), an expectation-maximization (EM) procedure that alternates two steps (Algorithm 3 in Appendix B). The E-step runs the forward-backward algorithm to obtain the state posteriors and pairwise transition posteriors. The M-step then updates the initial distribution, transition matrix, and emission matrix from expected counts: π from the posterior state occupancy at the first step, A from expected transitions between latent states, and B from expected emissions of each single observation atom.

3.2 Online Failure Attribution

Preliminary. Failure attribution aims to systematically determine which agent is at fault and when. Prior work is mainly post-hoc, analyzing the whole trace only after a failure has manifested, and relies on LLM prompting (Zhang et al., 2025b) or fine-tuning (Zhang et al., 2026). We instead aim to develop a method that operates in real time.

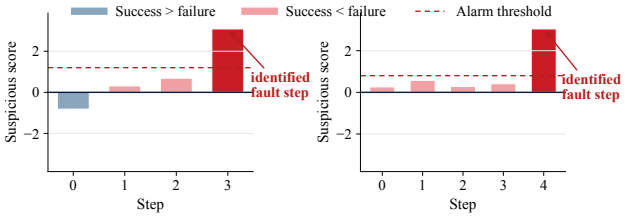


Figure 2: Illustration of HALT's online failure attribution on Who&When (left) and TracerTraj (right).

Suspicion score. Our principle is simple. A step is suspicious to the extent that its behavior conforms to failure patterns and deviates from correct ones. We therefore first fit two HMMs on same-domain historical traces, θ^- on failed traces and θ^+ on successful traces, which gives us a model of what failing and what succeeding each look like.

Given these two models, we need the probability that a given step was generated by each. Scoring the full-trace likelihood from the first step is unsuitable, since it would dilute a single anomalous step's evidence across an otherwise long, ordinary trace. We instead use a k-step conditional probability that only looks at the window $o_{b_t:t-1}$ of at most k preceding steps, with $b_t = \max(1, t-k)$, which matches the online setting and keeps the score local to the step being evaluated. For $t > 1$, a forward pass over the non-empty window, started from π as if the window began a trace, gives its likelihood

$$p_\theta(o_{b_t:t-1}) = \sum_{s\in\mathcal{S}} \alpha_{t-1}[s], \qquad \alpha_i[s] = p_B(o_i \mid s) \sum_{r\in\mathcal{S}} A_{r,s}\, \alpha_{i-1}[r], \quad i = b_t+1, \ldots, t-1, \tag{1}$$

with $\alpha_{b_t}[s] = \pi_s\, p_B(o_{b_t} \mid s)$. Normalizing α_{t-1} gives the filtered posterior $q_{t-1}[r] = \alpha_{t-1}[r]/p_\theta(o_{b_t:t-1})$ over the hidden state of the last window step, and the conditional probability of the current step follows as

$$p_\theta(o_t \mid o_{b_t:t-1}) = \sum_{s\in\mathcal{S}} p_B(o_t \mid s) \sum_{r\in\mathcal{S}} A_{r,s}\, q_{t-1}[r]. \tag{2}$$

For $t = 1$, the preceding window is empty and the conditional probability reduces to $\sum_{s\in\mathcal{S}} \pi_s p_B(o_1 \mid s)$. Algorithm 4 in Appendix B gives this computation for an arbitrary preceding window.

The suspicion score of step t contrasts the two models,

$$h_t = \log p_{\theta^-}(o_t \mid o_{b_t:t-1}) - \log p_{\theta^+}(o_t \mid o_{b_t:t-1}), \tag{3}$$

which is large when the observed step is more likely under the failure HMM θ^- than under the successful HMM θ^+ given the same recent window. HALT flags step t as the decisive step as soon as $h_t \geq \tau_h$, the alarm threshold illustrated in Figure 2. Algorithm 1 summarizes the online procedure.

Agent-level attribution is a special case of the same procedure in which only the window changes. To attribute failure to a specific agent, the context window is restricted to that agent's own steps, starting from the point at which the corresponding sub-agent began acting. Let φ_t be the agent acting at step t and κ_t the step at which its current run began. The window $o_{b_t:t-1}$ is replaced by $o^{\varphi_t}_{\kappa_t:t-1}$, the steps of agent φ_t from κ_t to $t-1$, which gives the agent-level score h^{φ}_t, and the first agent whose score reaches τ_h is reported.

Algorithm 1 Online Failure Attribution

Input: (Partial) execution trace $o_{1:T}$, fitted HMMs θ^- for failure traces and θ^+ for success traces, window size k, alarm threshold τ_h
Output: Flagged step $\hat{t}$

for $t = 1, \ldots, T$ **do** ▷ Process each step as it arrives
  $b_t \leftarrow \max(1, t-k)$. ▷ Use at most k preceding steps
  $h_t \leftarrow \log p_{\theta^-}(o_t \mid o_{b_t:t-1}) - \log p_{\theta^+}(o_t \mid o_{b_t:t-1})$ by Algorithm 4. ▷ Suspicion score
  if $h_t \geq \tau_h$ **then**
    return t. ▷ Stop at the first threshold crossing
  end if
end for ▷ No alarm in the observed trajectory

Algorithm 2 Automated Operator Acquisition and Workflow Search

Input: Fitted HMM $\theta = (\pi, A, B)$; successful historical traces $\mathcal{D}_{\text{succ}}$, with $\mathcal{X}_i, \mathcal{X}_\omega \subseteq \mathcal{D}_{\text{succ}}$ the trace segments aligned to state i and motif ω
Input: Edge-pruning threshold ρ, maximum extracted sequence length $\text{len}_{\max}$, LLM synthesizer Λ
Input: Evaluator G, validation split $\mathcal{D}_V$, maximum iterations $N_{\max}$
Output: Optimized workflow $W^\star$

Phase 1: Operator Acquisition
Construct F_ρ from A, retaining edges with $A_{ij} \geq \rho$ and $i \neq j$.
$v_i \leftarrow \Lambda(B_i, \mathcal{X}_i)$ for each hidden state i. ▷ Synthesize prompts, tools, and I/O formats
$\Omega \leftarrow \textsc{ExtractLoops}(F_\rho)$. ▷ Extract loops first
$\Omega \leftarrow \Omega \cup \textsc{ExtractBranches}(F_\rho) \cup \textsc{ExtractEnsembles}(F_\rho)$. ▷ Branch and ensemble motifs
$F_{\text{rem}} \leftarrow F_\rho$ with edges $\bigcup_{\omega \in \Omega} E(\omega)$ removed.
$\Omega \leftarrow \Omega \cup \textsc{ExtractSequences}(F_{\text{rem}}, \text{len}_{\max})$.
$Op_{\text{auto}} \leftarrow \{\Lambda(\omega; \{v_i\}_{i \in V(\omega)}, \mathcal{X}_\omega) : \omega \in \Omega\}$. ▷ Compile motifs into operators
Phase 2: Workflow Search
Initialize W_0 from $v_{\arg\max_i \pi_i}$; $W^\star \leftarrow W_0$.
tree $\leftarrow \textsc{InitializeTree}(W_0, \textsc{Evaluate}(W_0, G, \mathcal{D}_V))$.
for $n = 1, \ldots, N_{\max}$ **do**
  $W \leftarrow \textsc{SelectUCT}(\text{tree})$. ▷ UCT selection
  $W' \leftarrow \textsc{Expand}(W, Op_{\text{auto}})$.
  $\sigma \leftarrow \textsc{Evaluate}(W', G, \mathcal{D}_V)$. ▷ Evaluate on the validation set
  Add W' as a child of W; $\textsc{Backpropagate}(W', \sigma)$.
  Update $W^\star$ if W' improves the best evaluation score.
  break if convergence criteria are met.
end for
return $W^\star$

3.3 Automated Workflow Optimization

Preliminary. Agent workflow design has moved from hand-crafted pipelines to automated search and adaptive operator discovery (Hu et al., 2025; Zhang et al., 2025a; Zhao et al., 2026a). A hand-designed workflow encodes one expert's fixed choice of steps and lacks flexibility, whereas automated search explores a space of candidate workflows and returns the one that performs best on a validation split $\mathcal{D}_V$. This shift from static templates toward dynamically optimized runtime graphs is also emphasized in recent workflow-optimization surveys (Yue et al., 2026). We follow AFlow's formulation (Zhang et al., 2025a), in which a workflow is a graph whose nodes invoke an LLM, each specified by a prompt, tools, and an input/output format, and whose edges encode control flow between nodes. The set of all such graphs forms the base search space $\mathcal{W}_{\text{base}}$.

Because $\mathcal{W}_{\text{base}}$ is too large to enumerate, prior work searches it with Monte Carlo Tree Search (MCTS) (Kocsis & Szepesvári, 2006; Zhang et al., 2025a). AFlow further restricts the node choice at each search step to a small library of hand-defined *operators* Op (e.g., self-refine, multi-agent debate), which narrows the search and makes it tractable. However, since Op is authored by hand, the resulting pipeline is only partially auto-

mated. A2Flow also shows that replacing manually predefined operators with self-adaptive abstraction operators can improve generalization and resource efficiency (Zhao et al., 2026a).

Automated Operator Acquisition and Workflow Search. Figure 3 illustrates the pipeline and Algorithm 2 gives the full procedure.

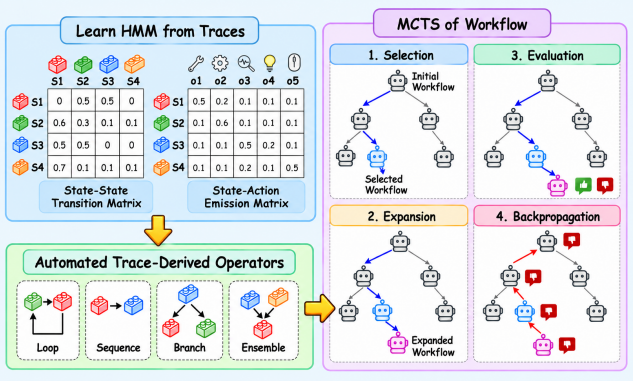


Figure 3: Workflow optimization by HALT.

Phase 1, operator acquisition. This phase acquires Op automatically from an HMM $\theta = (\pi, A, B)$ fitted on historical successful traces, treating every recurring high-weight substructure in the learned transition graph A as a candidate operator. We first prune A at edge-pruning threshold ρ to obtain a sparse graph F_ρ, then extract loops, branches, and ensembles, and chunk the remaining states into linear sequences of length at most $\mathrm{len}_{\max}$. Each resulting motif ω, with node set $V(\omega)$ and edge set $E(\omega)$, is a candidate operator.

To turn a candidate motif into an executable workflow node, an LLM synthesizer Λ, grounded in the emission distributions B_i of its states and the retrieved historical trace segments $\mathcal{X}_\omega$, compiles ω into an operator with a concrete prompt, interface, and output format. The operators derived this way form the induced operator bank Op_{auto}, which restricts the base search space to

$$\mathcal{W}_{\text{HALT}} = \mathcal{W}_{\text{base}}\big[Op_{\text{auto}}\big] \subseteq \mathcal{W}_{\text{base}}, \tag{4}$$

where $\mathcal{W}_{\text{base}}[Op_{\text{auto}}]$ denotes the search subspace induced by the automatically discovered operators Op_{auto}, which serve as structural anchors for workflow search.

Phase 2, workflow search. The optimized workflow maximizes the evaluator score on the validation split over the induced search space,

$$W^\star = \arg\max_{W \in \mathcal{W}_{\text{HALT}}} G\big(W; \mathcal{D}_V\big), \tag{5}$$

where G is the task's evaluator (its pass rate, solve rate, or accuracy). We approximate it with the Monte Carlo Tree Search procedure, run until the maximum number of iterations $N_{\max}$ is reached or the convergence criteria are met. Each iteration has four steps (Zhang et al., 2025a). *Selection* picks a workflow from the search tree to improve next using the UCT (Upper Confidence Bounds for Trees). *Expansion* uses an LLM as an optimizer to modify the selected workflow, drawing on operators from Op_{auto}. *Evaluation* runs the modified workflow on $\mathcal{D}_V$ to obtain feedback. *Backpropagation* then records (1) the workflow's performance, (2) the optimizer's modification relative to its parent workflow, and (3) whether the modification improved on the parent. This information is stored as experience and propagated back to the parent workflow, while the performance score is added to the global record used for selection.

4 Experimental Setup

4.1 Datasets and Splits

Failure Attribution. We evaluate on two benchmarks. Who&When (Zhang et al., 2025b) contains 184 failed multi-agent traces, 126 Algorithm-Generated and 58 Hand-Crafted, each annotated with the responsible agent and the decisive error step. TracerTraj-public v1.0.0 (Zhang et al., 2026) covers Code, Math, and Agentic traces, and we use its official test split of 127, 63, and 600 traces. Because ground-truth solutions are rarely available in practice, the main experiments give no method access to them, and the ground-truth-assisted setting used by prior benchmarks is reported in Appendix G.

Workflow Optimization. We use four benchmarks. HumanEval Pro and MBPP Pro (Yu et al., 2025) extend HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) with self-invoking problems

for code generation, MATH Lv5 consists of the hardest level of MATH (Hendrycks et al., 2021), and SimpleQA Verified (Haas et al., 2025) is a curated version of SimpleQA (Wei et al., 2024) for short-form factual question answering. Each benchmark is split 20/80. The 20% split is the MCTS validation set used by the *evaluate* step to score candidate workflows during search, and the remaining 80% is the test set on which Table 4 is reported, which gives 131, 302, 1,059, and 800 test instances for HumanEval Pro, MBPP Pro, MATH Lv5, and SimpleQA Verified. Statistics of all six benchmarks are given in Appendix E.

HMM Training Data. All HMMs are trained on same-domain traces that never include the evaluation instances. The trace sources of each benchmark are described in Appendix F.

4.2 Hyperparameters

Unless otherwise stated, we use $M = 5$ hidden states for Baum-Welch fitting, running until the log-likelihood changes by less than the convergence tolerance $\epsilon = 10^{-3}$. The suspicion-score window is $k = 3$ steps. The alarm threshold is fixed per dataset, at $\tau_h = 1.2$ on Who&When and $\tau_h = 0.8$ on TracerTraj. For workflow optimization, the edge-pruning threshold is $\rho = 0.08$, the maximum extracted sequence length is $\mathrm{len}_{\max} = 4$, and the search runs for at most $N_{\max} = 20$ rounds, stopping early when the average score of the top three workflows has not improved for five consecutive rounds.

4.3 Language Models

Failure attribution and workflow optimization backbones differ by benchmark. On Who&When, the GPT-4o (OpenAI, 2024) offline baselines are taken from the source paper (Zhang et al., 2025b), while we run DeepSeek-V4-Pro ourselves; the numbers of AgenTracer, a fine-tuned LLM attributor (Zhang et al., 2026), are likewise taken from its paper. On TracerTraj-public, we run both GLM-5.1 (GLM-5-Team, 2026) and DeepSeek-V4-Pro (DeepSeek-AI, 2026) as offline backbones. Workflow optimization is evaluated with GLM-5.1 and DeepSeek-V4-Pro, and the operator synthesizer Λ is DeepSeek-V4-Pro in all experiments. For every model we run ourselves, we enable thinking mode and set reasoning effort to `max`, leaving all other parameters at default.

4.4 Baselines

Failure Attribution. We compare HALT against three baseline families. The offline family covers three zero-shot prompting protocols (Zhang et al., 2025b), run over the backbone models above. All-at-Once gives the LLM the entire trace at once and asks it to name the responsible agent and the decisive step in a single pass. Step-by-Step lets the LLM read the trace one step at a time and make a binary judgment at each step on whether it is the decisive error. Binary Search repeatedly asks the LLM which half of the current segment contains the error and narrows the trace until a single step remains. AgenTracer (Zhang et al., 2026) is a fine-tuned LLM attributor, which we report on Who&When only because its model weights are not released for TracerTraj-public. The online family covers ProMAS (Zhao et al., 2026b) and MASC (Shen et al., 2026), which attribute failure without seeing the full trace in advance.

Workflow Optimization. We compare against AutoGen (Wu et al., 2024), ADAS (Hu et al., 2025), and AFlow (Zhang et al., 2025a). These differ in automation level, i.e., how much of the workflow and operator design is delegated from a human expert to the search procedure itself.

4.5 Metrics

Failure Attribution. We report agent-level and step-level attribution accuracy, which are the fractions of traces whose predicted responsible agent and predicted decisive step match the annotation, together with the average token usage per trace and the number of trajectory steps examined before conclusion.

Workflow Optimization. We report benchmark-specific task performance: pass@1 for HumanEval Pro and MBPP Pro, where evaluation uses the added self-invoking problems; solve rate for MATH Lv5; and accuracy for SimpleQA Verified. We also report the total dollar cost of running the final workflow on the test set. Workflow results are the mean of three runs.

5 RESULTS

5.1 FAILURE ATTRIBUTION

Table 1 and Table 2 report failure attribution accuracy at the agent and step levels, on TracerTraj-public and Who&When respectively (Zhang et al., 2025b; 2026). N/A denotes unavailable or inapplicable cells. MASC's prototype-guided next-execution reconstruction scores each step for anomaly but performs no agent identification (Shen et al., 2026). TracerTraj Agentic agent-level cells are N/A because the Agentic split is a single-agent setting. AgenTracer, a fine-tuned LLM attributor, reports TracerTraj results on a private split; no public-release results or model parameters are released (Zhang et al., 2026). Results with ground-truth assistance, as used by prior benchmarks, are deferred to Appendix G.

Table 1: Performance comparison on different subsets of TracerTraj-public. For each subset, accuracy is reported at the agent/step levels. N/A denotes inapplicable cells.

	Method	Code		Math		Agentic	
		Agent	**Step**	**Agent**	**Step**	**Agent**	**Step**
Offline	*GLM-5.1*						
	All-at-Once	62.68	11.81	61.90	42.38	N/A	15.50
	Step-by-Step	61.10	12.60	57.14	42.38	N/A	17.39
	Binary Search	62.14	9.45	63.49	39.21	N/A	20.50
	DeepSeek-V4-Pro						
	All-at-Once	62.68	14.17	50.79	42.86	N/A	19.17
	Step-by-Step	62.68	13.39	53.97	42.86	N/A	16.83
	Binary Search	48.82	6.30	55.08	39.21	N/A	15.67
Online	ProMAS	39.37	4.72	58.73	42.86	N/A	22.67
	MASC	N/A	5.51	N/A	23.81	N/A	20.67
	HALT	**63.78**	**21.26**	**65.08**	**44.44**	**N/A**	**21.50**

HALT achieves the best accuracy in 8 of the 9 comparable columns across the two tables. On TracerTraj Code, HALT reaches 63.78%/21.26% agent/step accuracy, ahead of the strongest offline baseline (DeepSeek All-at-Once, 62.68%/14.17%). On Who&When, HALT leads on both subsets, reaching 69.84%/42.86% on Algorithm-Generated and 70.69%/24.14% on Hand-Crafted.

Online methods generally trail offline ones, since offline attribution has access to the full trace while online methods must decide with only a truncated trace. However, Table 3 shows why online is more practical, since it costs fewer tokens and does not need to observe the full trace. HALT, despite being online, still outperforms every offline baseline in Table 1 and Table 2, at zero LLM token cost and while observing a smaller fraction of the trace. This indicates that the hidden-state transition structure, not just the surface trace content, is what carries most of the signal for failure attribution.

Table 2: Performance comparison on different subsets of Who&When. For each subset, accuracy is reported at the agent/step levels. N/A denotes inapplicable cells.

	Method	Algorithm-Generated		Hand-Crafted	
		Agent-level	**Step-level**	**Agent-level**	**Step-level**
Offline	*GPT-4o*				
	All-at-Once	51.12	13.53	53.44	3.51
	Step-by-Step	26.02	15.31	32.75	8.77
	Binary Search	30.11	16.59	36.21	6.90
	DeepSeek-V4-Pro				
	All-at-Once	54.76	31.27	44.83	15.52
	Step-by-Step	54.76	29.68	51.72	17.24
	Binary Search	39.68	25.40	60.34	15.52
	AgenTracer	63.73	37.30	63.82	20.68
Online	ProMAS	46.53	24.75	27.66	19.14
	MASC	N/A	21.72	N/A	20.79
	HALT	**69.84**	**42.86**	**70.69**	**24.14**

5.2 WORKFLOW OPTIMIZATION

Table 4 presents performance across four benchmarks for two language models, GLM-5.1 and DeepSeek-V4-Pro. The automation level column reflects the degree of manual effort and expert knowledge required: Manual Workflow relies on a hand-crafted static template; Auto Workflow searches the workflow structure automatically; Auto Workflow + Manual Op. searches the workflow structure automatically but builds it from hand-designed operators; and Auto Workflow + Auto Op. represents full automation, including operator design.

Automation does not help uniformly. With GLM-5.1, ADAS trails AutoGen on three of the four tasks and AFlow trails it on SimpleQA Verified (24.6 vs. 32.2) and HumanEval Pro (78.0 vs. 79.0), and only with DeepSeek-V4-Pro does AFlow beat AutoGen on all four tasks. AFlow's composition of predefined building blocks still relies on a fixed operator library designed by experts, which caps what its search can reach.

Table 3: Inference and information efficiency of failure attribution methods. We measure inference efficiency by LLM tokens per trace and information efficiency by mean observed steps relative to whole trace length (observed/total). Fewer tokens indicate lower LLM inference cost, while fewer observed steps indicate earlier failure detection. N/A denotes unavailable entries.

Method	Who&When		TracerTraj	
	token usage per trace	steps/total	token usage per trace	steps/total
All-at-Once	3,148	22.24/22.24	2,260	10.94/10.94
Step-by-Step	3,154	22.24/22.24	2,273	10.94/10.94
Binary Search	6,214	22.24/22.24	4,605	10.94/10.94
AgenTracer	3,148	22.24/22.24	N/A	N/A
ProMAS	1,263	8.92/22.24	1,411	6.83/10.94
MASC	1,328	9.38/22.24	1,680	8.13/10.94
HALT	**0**	**5.96/22.24**	**0**	**4.06/10.94**

Table 4: Performance comparison of workflow optimization methods across different levels of automation. Each workflow is run three times and we report the average. The robot icon marks automated components.

Model	Method	Automation Level	SimpleQA Verified	MATH Lv5	MBPP Pro	HumanEval Pro
GLM-5.1	AutoGen	Manual Workflow	32.2	79.4	69.4	79.0
	ADAS	Auto Workflow	31.0	77.0	73.6	72.7
	AFlow	Auto Workflow + Manual Op.	24.6	79.6	73.6	78.0
	HALT	Auto Workflow + Auto Op.	38.6	82.0	86.2	84.0
DeepSeek-V4-Pro	AutoGen	Manual Workflow	35.6	69.6	77.4	78.2
	ADAS	Auto Workflow	33.0	76.4	77.8	79.6
	AFlow	Auto Workflow + Manual Op.	37.4	80.0	81.2	78.8
	HALT	Auto Workflow + Auto Op.	39.4	84.6	86.2	80.9

Our method HALT automates both workflow and operator design, and achieves gains across every benchmark. On MBPP Pro, HALT improves over the strongest baseline by 12.6 points with GLM-5.1. The learned operators also outperform AFlow's manually predefined ones on every task.

In terms of cost, Figure 4 shows that HALT workflows have the lowest or tied-lowest total cost on MATH Lv5, MBPP Pro, and HumanEval Pro under both backbones, while on SimpleQA Verified the cost is comparable to the baselines.

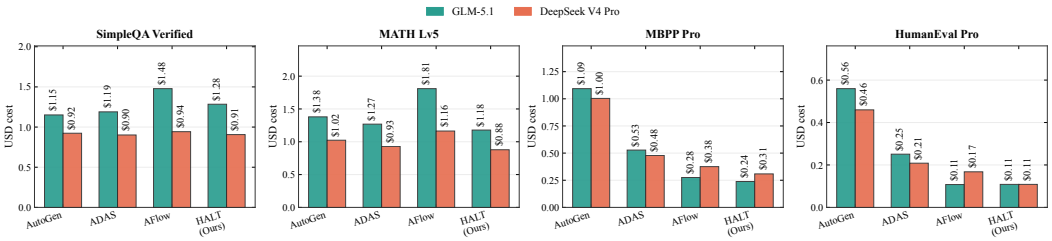


Figure 4: Total USD cost of running each method's final workflow on the test set.

6 Conclusion

We presented HALT, a framework that models agent execution traces as hidden Markov models, treating atomic actions as emissions of latent workflow states and state transitions as the behavioral logic that organizes them. For failure attribution, contrasting step likelihoods under failure and success models localizes the decisive step online without LLM calls, and it surpasses a fine-tuned LLM attributor on Who&When while observing only about 27% of the trace. For workflow optimization, recurring substructures in the transition graph become reusable operators that replace hand-written ones, and the resulting workflows outperform baselines on all four tasks.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970. doi: 10.1214/aoms/1177697196.

Ruidi Chang, Jiawei Zhou, and Hanjie Chen. Hidden Markov modeling of reasoning dynamics in Large Language Models, 2026. URL https://openreview.net/pdf?id=fr9t7r43am. OpenReview ICLR 2026 submission.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374. Version 2.

Seonglae Cho, Franklin Cardenoso Fernandez, Umar Mohammed, Zekun Wu, Kleyton Da Costa, Ilham Wicaksono, and Adriano Koshiyama. Automata from agent traces: Failure and next-step prediction, 2026. URL https://arxiv.org/abs/2608.23670.

Yijia Dai, Zhaolin Gao, Yahya Sattar, Sarah Dean, and Jennifer J. Sun. Pre-trained large language models learn to predict hidden Markov models in-context. In *Advances in Neural Information Processing Systems 38*, 2025. doi: 10.52202/085713-0970. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/29a0ea49a103a233b17c0705cdeccb66-Abstract-Conference.html.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

GLM-5-Team. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. ToRA: A tool-integrated reasoning agent for mathematical problem solving. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Ep0TtjVoap.

Lukas Haas, Gal Yona, Giovanni D'Antonio, Sasha Goldshtein, and Dipanjan Das. SimpleQA Verified: A reliable factuality benchmark to measure parametric knowledge, 2025. URL https://arxiv.org/abs/2509.07968.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6864–6890. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.371. URL https://aclanthology.org/2024.acl-long.371/.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/

paper/2021/hash/be83ab3ecd0db773eb2dc1b0a17836a1-Abstract-round2.html.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 24370–24391. PMLR, 2024. URL https://proceedings.mlr.press/v235/kim24y.html.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pp. 282–293. Springer, 2006. doi: 10.1007/11871842_29.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants, 2023. URL https://arxiv.org/abs/2311.12983.

OpenAI. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989. doi: 10.1109/5.18626.

Xu Shen, Qi Zhang, Song Wang, Zhen Tan, Xinyu Zhao, Laura Yao, Vaishnav Tadiparthi, Hossein Nourkhiz Mahjoub, Ehsan Moradi Pari, Kwonjoon Lee, and Tianlong Chen. Metacognitive self-correction for multi-agent system via prototype-guided next-execution reconstruction. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 23320–23337. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1168. URL https://aclanthology.org/2026.findings-acl.1168/.

Junjie Wang, Yawen Wang, Mengzhuo Chen, Xiaofei Xie, Chunyang Chen, Fangwen Mu, Zhe Liu, and Qing Wang. A survey for LLM agent trajectory analysis: From failure attribution to enhancement. *IEEE Transactions on Software Engineering*, 2026. doi: 10.1109/TSE.2026.3717765. URL https://doi.org/10.1109/TSE.2026.3717765. Accepted/In press.

Jason Wei, Nguyen Karina, Hyung Won Chung, Yunxin Joy Jiao, Spencer Papay, Amelia Glaese, John Schulman, and William Fedus. Measuring short-form factuality in large language models, 2024. URL https://arxiv.org/abs/2411.04368.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-Agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024.

Xu Yang, Jiayuan Zhou, Michael Pacheco, Wenhan Zhu, Pengfei He, Shaowei Wang, Kui Liu, and Ruiqi Pan. Lingxi: Repository-level issue resolution framework enhanced by procedural knowledge guided scaling. In *Proceedings of the 35th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2026. URL https://arxiv.org/abs/2510.11838. arXiv:2510.11838.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. AssistantBench: Can web agents solve realistic and time-consuming tasks? In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 8938–8968, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.505. URL https://aclanthology.org/2024.emnlp-main.505/.

Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. HumanEval Pro and MBPP Pro: Evaluating large language models on self-invoking code generation task. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 13253–13279, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.686. URL https://aclanthology.org/2025.findings-acl.686/.

Ling Yue, Kushal Raj Bhandari, Ching-Yun Ko, Dhaval Patel, Shuxin Lin, Nianjun Zhou, Jianxi Gao, Pin-Yu Chen, and Shaowu Pan. From static templates to dynamic runtime graphs: A survey of workflow optimization for LLM agents, 2026. URL https://arxiv.org/abs/2603.22386.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. AgenTracer: Who is inducing failure in the LLM agentic systems? In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=l05DseqvuD. arXiv:2509.03312v2.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=z5uVAKwmjf.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76583–76599. PMLR, 2025b. URL https://proceedings.mlr.press/v267/zhang25cq.html.

Mingming Zhao, Xiaokang Wei, Yuanqi Shao, Kaiwen Zhou, Lin Yang, Siwei Rao, Junhui Zhan, and Zhitang Chen. A^2Flow: Automating agentic workflow generation via self-adaptive abstraction operators. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29930–29938, 2026a. doi: 10.1609/aaai.v40i35.40240.

Xinkui Zhao, Sai Liu, Yifan Zhang, Qingyu Ma, Guanjie Cheng, Naibo Wang, and Chang Liu. ProMAS: Proactive error forecasting for multi-agent systems using Markov transition dynamics, 2026b. URL https://arxiv.org/abs/2603.20260.

DECLARATION OF AI ASSISTANCE

AI-assisted tools were used solely to polish the language of this manuscript, improving grammar, clarity, and readability.

A RELATED WORK

Agentic workflow optimization. AutoGen provides a static, hand-designed template in which role-specialized agents converse under hand-specified turn-taking (Wu et al., 2024). ADAS automates agent design, with a meta agent that iteratively programs new agents directly in code, guided by an archive of previously discovered designs and their validation performance (Hu et al., 2025). AFlow frames workflow generation as Monte Carlo Tree Search over code-represented workflows built from a fixed set of hand-defined operators, such as `Ensemble`, `Review`, and `Revise`, so the search composes expert-designed operators rather than discovering them (Zhang et al., 2025a). HALT keeps the same outer objective, finding a workflow that maximizes task performance given validation feedback, but derives the reusable operators from recurring substructures of the transition graph learned from successful traces rather than placing them by hand.

Failure attribution in multi-agent systems. Who&When provides failed multi-agent logs annotated with a single responsible `mistake_agent` and `mistake_step` and a natural-language reason, over two subsets, Algorithm-Generated and Hand-Crafted (Zhang et al., 2025b). AgenTracer introduces TracerTraj, a larger train/test attribution dataset spanning agentic, coding, and mathematics domains, and fine-tunes a dedicated LLM attributor, also called AgenTracer, on it (Zhang et al., 2026). Recent trajectory-analysis surveys organize this area around failure taxonomy, failure attribution, monitoring, datasets, and system enhancement (Wang et al., 2026). Published methods for Who&When are predominantly zero-shot LLM prompting strategies, namely All-at-Once, Step-by-Step, and Binary Search, all of which analyze a completed trace. Online methods avoid waiting for the complete trace. MASC detects step-level anomalies by prototype-guided reconstruction of the next execution and uses them for metacognitive self-correction (Shen et al., 2026), and ProMAS forecasts errors proactively using Markov transition dynamics (Zhao et al., 2026b). HALT instead infers latent workflow states with a hidden Markov model and scores each step by contrasting its likelihood under failure and success models, which requires no LLM at inference.

B ALGORITHM DETAILS

B.1 BAUM-WELCH ESTIMATION

Algorithm 3 is the expectation-maximization procedure used to fit every HMM in this paper (Section 3.1). Each iteration alternates two steps. The E-step runs the forward and backward recursions under the current $\theta = (\pi, A, B)$ and normalizes by the sequence likelihood Z, which gives the state posteriors $\gamma_t(s)$ and the transition posteriors $\xi_t(s, r)$. The M-step then re-estimates π, A, and B from expected counts: initial-state occupancy, expected transitions per expected departure, and expected emissions of each observation atom per expected state occupancy. The loop stops once the log-likelihood ℓ changes by less than ϵ.

The algorithm is written for a single sequence for clarity. For a corpus of traces, the expected counts are summed over traces and the stopping test uses the total log-likelihood, with the convergence tolerance from Section 4.2. We initialize π uniformly, A with a bias toward staying in the current state and moving to the next state index, and B at random with a fixed seed. Each iteration costs $O(TM^2)$ per sequence for the recursions.

Algorithm 3 Baum-Welch Algorithm

Input: State space $\mathcal{S} = \{1, \dots, M\}$, observation space $\mathcal{O} = \{1, \dots, L\}$, observation sequence $o_{1:T}$, initial parameters $\theta^{(0)} = (\pi^{(0)}, A^{(0)}, B^{(0)})$, convergence tolerance ϵ
Output: Refined parameters $\theta = (\pi, A, B)$

$(\pi, A, B) \leftarrow (\pi^{(0)}, A^{(0)}, B^{(0)})$, $\ell \leftarrow -\infty$
repeat
  $\ell_{\text{prev}} \leftarrow \ell$
  E-step: compute expected hidden-state statistics under the current θ
  $\alpha_1[s] \leftarrow \pi_s \, p_B(o_1 \mid s)$ for all $s \in \mathcal{S}$ ▷ Forward pass
  $\alpha_t[s] \leftarrow p_B(o_t \mid s) \sum_{r \in \mathcal{S}} A_{r,s} \, \alpha_{t-1}[r]$ for $t = 2, \dots, T$, $s \in \mathcal{S}$
  $Z \leftarrow \sum_{s \in \mathcal{S}} \alpha_T[s]$, $\ell \leftarrow \log Z$ ▷ Sequence likelihood
  $\beta_T[s] \leftarrow 1$ for all $s \in \mathcal{S}$ ▷ Backward pass
  $\beta_t[s] \leftarrow \sum_{r \in \mathcal{S}} A_{s,r} \, p_B(o_{t+1} \mid r) \, \beta_{t+1}[r]$ for $t = T-1, \dots, 1$, $s \in \mathcal{S}$
  $\gamma_t(s) \leftarrow \dfrac{\alpha_t[s] \, \beta_t[s]}{Z}$ for $t = 1, \dots, T$, $s \in \mathcal{S}$ ▷ State posterior
  $\xi_t(s, r) \leftarrow \dfrac{\alpha_t[s] \, A_{s,r} \, p_B(o_{t+1} \mid r) \, \beta_{t+1}[r]}{Z}$ for $t = 1, \dots, T-1$, $s, r \in \mathcal{S}$ ▷ Transition posterior
  M-step: re-estimate θ from the expected counts
  $\pi_s \leftarrow \gamma_1(s)$ for all $s \in \mathcal{S}$ ▷ Initial distribution
  $A_{s,r} \leftarrow \dfrac{\sum_{t=1}^{T-1} \xi_t(s, r)}{\sum_{t=1}^{T-1} \gamma_t(s)}$ for all $s, r \in \mathcal{S}$ ▷ Transition matrix
  $B_{s,u} \leftarrow \dfrac{\sum_{t=1}^{T} \gamma_t(s) \mathbf{1}\{o_t = u\}}{\sum_{t=1}^{T} \gamma_t(s)}$ for all $s \in \mathcal{S}$, $u \in \mathcal{O}$ ▷ Emission matrix
until $|\ell - \ell_{\text{prev}}| < \epsilon$ ▷ Convergence check
return $\theta = (\pi, A, B)$

B.2 k-Step Conditional Probability

Algorithm 4 computes the conditional probability of the current observation given a preceding window, which is the quantity behind the online attribution rule (Section 3.2). The window is at most k preceding steps for step-level attribution and the sub-agent's own steps since it began acting for agent-level attribution, and the routine treats both alike. A forward pass over the window, started from the initial distribution π as if the window began a trace, gives the window likelihood. Normalizing the final forward vector gives the posterior q_m over the hidden state of the last window step, propagating it through A gives the belief $\hat{q}$ over the state of the current step, and mixing the emission probabilities of the current observation under this belief gives its conditional probability. For an empty window the belief is π.

Algorithm 1 calls this routine twice per score, once with θ^- and once with θ^+, and takes the log ratio as the suspicion score. Since a window has at most k steps at the step level, the cost per step is $O(kM^2)$ regardless of trace length, which is what keeps the attribution online.

Algorithm 4 Compute $p_\theta(o \mid o_{\text{pre}})$

Input: Fitted HMM $\theta = (\pi, A, B)$ with state space $\mathcal{S} = \{1, \dots, M\}$, preceding window $o_{\text{pre}} = (o_{(1)}, \dots, o_{(m)})$, current observation o
Output: Conditional probability $p_\theta(o \mid o_{\text{pre}})$

if $m = 0$ **then**
  $\hat{q}[s] \leftarrow \pi_s$ for all $s \in \mathcal{S}$ ▷ Empty window: prior belief
else
  $\alpha_1[s] \leftarrow p_B(o_{(1)} \mid s)\, \pi_s$ for all $s \in \mathcal{S}$ ▷ Initialize at window start
  for $j = 2, \dots, m$ **do**
    $\alpha_j[s] \leftarrow p_B(o_{(j)} \mid s) \sum_{r \in \mathcal{S}} A_{r,s}\, \alpha_{j-1}[r]$ for all $s \in \mathcal{S}$ ▷ Forward recursion
  end for
  $p_\theta(o_{\text{pre}}) \leftarrow \sum_{s \in \mathcal{S}} \alpha_m[s]$ ▷ Window likelihood
  $q_m[s] \leftarrow \alpha_m[s] \,/\, p_\theta(o_{\text{pre}})$ for all $s \in \mathcal{S}$ ▷ Posterior of last window state
  $\hat{q}[s] \leftarrow \sum_{r \in \mathcal{S}} A_{r,s}\, q_m[r]$ for all $s \in \mathcal{S}$ ▷ Predict state of current step
end if
return $\sum_{s \in \mathcal{S}} p_B(o \mid s)\, \hat{q}[s]$ ▷ Predictive probability

C Atomic-Action Mapping per Dataset

Every trace step is mapped deterministically, without any LLM call, to one primary atom by the three rules of Section 3.1, applied in priority order: (1) *tool calls*, where a tool-call field yields the function name; (2) *action tags*, where an `[ACTION]` description yields the action label; and (3) *command extraction*, where the first command token of a code block is mapped to a semantic category. A step that matches none of them receives the default atom `role:content_type`. The observation space $\mathcal{O}$ of a dataset is the set of atoms observed in it. Table 5 lists, for each dataset, representative atoms and the resulting size $L = |\mathcal{O}|$.

Table 5: Atoms per dataset. For each dataset we list representative atoms and the number of distinct atoms $L = |\mathcal{O}|$.

Dataset	Example atoms	L
Who&When	`perform_web_search`, `browser`, `ocr`, `verify`, `assistant:text`	28
TracerTraj-Code	`python`, `calculator`, `code`, `plan`, `verify`	22
TracerTraj-Math	`calculator`, `calculate`, `reason`, `finalize`	18
TracerTraj-Agentic	`web_search`, `browser`, `retrieve`, `verify`	25
HumanEval	`code`, `reason`, `verify`, `select`, `benchmark_verifier`	16
MBPP	`code`, `verify`, `select`, `modify_workflow`, `benchmark_verifier`	16
MATH (excl. Lv5)	`reason`, `calculate`, `verify`, `finalize`, `select`	15
SimpleQA	`reason`, `retrieve`, `verify`, `finalize`, `select`	14

D HALT-Induced Operators

AFlow represents every operator as code, namely a subclass of `Operator` whose `__call__` builds a prompt from a template and fills a typed output node with an LLM call (`_fill_node`). HALT's synthesizer Λ produces operators in this same form, with each operator chaining the LLM nodes of one hidden-state motif and having a concrete prompt, interface, and output format for every node. The comment after each class name gives the motif it was induced from. For each task we show three induced operators; the prompt templates are abridged and the shared imports are listed once.

Shared imports

```
import asyncio
from scripts.operators import Operator, Test, ScEnsemble, MdEnsemble
from scripts.operator_an import GenerateOp, AnswerGenerateOp, FormatOp, ReviewOp, ReviseOp
from scripts.prompts.prompt import (ANSWER_GENERATION_PROMPT, FORMAT_PROMPT,
```

```
                                        REVIEW_PROMPT, REVISE_PROMPT)
```

HumanEval Pro

```
PLAN_PROMPT = ("Restate the requirements, list edge cases, and outline the algorithm.\n"
               "problem: {problem}")

class CodeReasonReview(Operator):  # Generate -> Reason -> Review
    async def __call__(self, problem, entry_point, instruction):
        fill = self._fill_node
        plan = await fill(GenerateOp, PLAN_PROMPT.format(problem=problem), mode="single_fill")
        code = await fill(GenerateOp, instruction + problem + plan["response"],
                          mode="code_fill", function_name=entry_point)
        review = await fill(ReviewOp, REVIEW_PROMPT.format(problem=problem,
                            solution=code["response"]), mode="xml_fill")
        return {"solution": code["response"], "review": review}

class ReviewRegenerateJudge(Operator):  # Review -> Generate -> Judge
    async def __call__(self, problem, solution, entry_point):
        fill = self._fill_node
        review = await fill(ReviewOp, REVIEW_PROMPT.format(problem=problem,
                            solution=solution), mode="xml_fill")
        if not review["review_result"]:
            revised = await fill(ReviseOp, REVISE_PROMPT.format(problem=problem,
                                 solution=solution, feedback=review["feedback"]),
                                 mode="xml_fill")
            solution = revised["solution"]
        return await Test(self.llm)(problem, solution, entry_point)

class JudgeSelfRefine(Operator):  # Judge -> Revise -> Judge
    async def __call__(self, problem, solution, entry_point, rounds: int = 2):
        test = Test(self.llm)
        for _ in range(rounds):
            out = await test(problem, solution, entry_point)
            solution = out["solution"]
            if out["result"]:
                break
        return {"solution": solution, "passed": out["result"]}
```

MBPP Pro

```
TRACE_PROMPT = ("Trace the code on the edge cases of the problem and list every failing "
                "case, or answer 'none'.\nproblem: {problem}\ncode: {solution}")

class PlanReviewGenerate(Operator):  # Reason -> Review -> Generate
    async def __call__(self, problem, entry_point, instruction):
        fill = self._fill_node
        plan = await fill(GenerateOp, PLAN_PROMPT.format(problem=problem), mode="single_fill")
        check = await fill(ReviewOp, REVIEW_PROMPT.format(problem=problem,
                           solution=plan["response"]), mode="xml_fill")
        prompt = instruction + problem + plan["response"] + check["feedback"]
        return await fill(GenerateOp, prompt, mode="code_fill", function_name=entry_point)

class GenerateReasonJudge(Operator):  # Generate -> Reason -> Judge
    async def __call__(self, problem, entry_point, instruction):
        fill = self._fill_node
        code = await fill(GenerateOp, instruction + problem, mode="code_fill",
                          function_name=entry_point)
        solution = code["response"]
        trace = await fill(GenerateOp, TRACE_PROMPT.format(problem=problem,
                           solution=solution), mode="single_fill")
        if "none" not in trace["response"].lower():
            fixed = await fill(ReviseOp, REVISE_PROMPT.format(problem=problem,
                               solution=solution, feedback=trace["response"]),
                               mode="xml_fill")
            solution = fixed["solution"]
        return await Test(self.llm)(problem, solution, entry_point)

class BranchToReview(Operator):  # {Generate, Reason} -> Review
    async def __call__(self, problem, entry_point, instruction):
        fill = self._fill_node
        plan = await fill(GenerateOp, PLAN_PROMPT.format(problem=problem), mode="single_fill")
        direct = await fill(GenerateOp, instruction + problem, mode="code_fill",
                            function_name=entry_point)
        guided = await fill(GenerateOp, instruction + problem + plan["response"],
                            mode="code_fill", function_name=entry_point)
        return await MdEnsemble(self.llm)([direct["response"], guided["response"]], problem)
```

MATH Lv5

```
STEP_PROMPT = ("Stage: {stage}. Continue the solution using the notes so far.\n"
               "problem: {problem}\nnotes: {notes}")
RETRY_PROMPT = ("The final answer below was judged incorrect or malformed. Re-solve the "
```

```
                "problem and put the final answer in \\boxed{{}}.\nproblem: {problem}\n"
                "attempt: {solution}\nfeedback: {feedback}")

class StepwiseSolveJudge(Operator):  # Solve_1 -> Solve_2 -> Solve_3 -> Judge
    async def __call__(self, problem):
        fill = self._fill_node
        notes = ""
        for stage in ("set up", "derive", "verify"):
            step = await fill(GenerateOp, STEP_PROMPT.format(problem=problem, stage=stage,
                              notes=notes), mode="single_fill")
            notes += step["response"] + "\n"
        return await fill(FormatOp, FORMAT_PROMPT.format(problem_description=problem,
                          solution=notes), mode="xml_fill")

class SelfConsistencyMerge(Operator):  # {Solve_a, Solve_b, Solve_c} -> Merge -> Judge
    async def __call__(self, problem, n: int = 3):
        fill = self._fill_node
        prompt = ANSWER_GENERATION_PROMPT.format(input=problem)
        outs = await asyncio.gather(*[fill(AnswerGenerateOp, prompt, mode="xml_fill")
                                      for _ in range(n)])
        texts = [o["thought"] + "\nAnswer: " + o["answer"] for o in outs]
        best = await ScEnsemble(self.llm)(texts, problem)
        return await fill(FormatOp, FORMAT_PROMPT.format(problem_description=problem,
                          solution=best["response"]), mode="xml_fill")

class AnswerMonitor(Operator):  # Judge -> Solve -> Judge
    async def __call__(self, problem, solution, retries: int = 1):
        fill = self._fill_node
        for _ in range(retries + 1):
            ans = await fill(FormatOp, FORMAT_PROMPT.format(problem_description=problem,
                             solution=solution), mode="xml_fill")
            check = await fill(ReviewOp, REVIEW_PROMPT.format(problem=problem,
                               solution=ans["solution"]), mode="xml_fill")
            if check["review_result"]:
                break
            retry = await fill(GenerateOp, RETRY_PROMPT.format(problem=problem,
                               solution=solution, feedback=check["feedback"]),
                               mode="single_fill")
            solution = retry["response"]
        return {"answer": ans["solution"]}
```

SimpleQA Verified

```
DECOMPOSE_PROMPT = ("List the atomic facts needed to answer the question and recall each "
                    "one.\nquestion: {question}")
FACT_REVISE_PROMPT = ("Correct the answer using the reviewer's feedback and reply with the "
                      "short answer only.\nquestion: {question}\nanswer: {answer}\n"
                      "feedback: {feedback}")

class RecallVerify(Operator):  # Solve -> Review -> Revise
    async def __call__(self, question):
        fill = self._fill_node
        draft = await fill(AnswerGenerateOp, ANSWER_GENERATION_PROMPT.format(input=question),
                           mode="xml_fill")
        check = await fill(ReviewOp, REVIEW_PROMPT.format(problem=question,
                           solution=draft["answer"]), mode="xml_fill")
        if check["review_result"]:
            return {"answer": draft["answer"]}
        fixed = await fill(ReviseOp, FACT_REVISE_PROMPT.format(question=question,
                           answer=draft["answer"], feedback=check["feedback"]),
                           mode="xml_fill")
        return {"answer": fixed["solution"]}

class SelfConsistentAnswer(Operator):  # {Solve_a, Solve_b, Solve_c} -> Merge
    async def __call__(self, question, n: int = 3):
        prompt = ANSWER_GENERATION_PROMPT.format(input=question)
        outs = await asyncio.gather(*[self._fill_node(AnswerGenerateOp, prompt, mode="xml_fill")
                                      for _ in range(n)])
        best = await ScEnsemble(self.llm)([o["answer"] for o in outs], question)
        return {"answer": best["response"]}

class DecomposeRecall(Operator):  # Reason -> Solve
    async def __call__(self, question):
        fill = self._fill_node
        facts = await fill(GenerateOp, DECOMPOSE_PROMPT.format(question=question),
                           mode="single_fill")
        draft = await fill(AnswerGenerateOp, ANSWER_GENERATION_PROMPT.format(
                           input=question + "\nKnown facts: " + facts["response"]),
                           mode="xml_fill")
        return {"answer": draft["answer"]}
```

E Dataset Statistics

Table 6 summarizes the two failure attribution benchmarks and Table 7 the four workflow optimization benchmarks; together they are the six evaluation benchmarks used in this paper.

Table 6: Statistics of the failure attribution benchmarks. Steps and agents are averaged per trace (agents are the distinct agent or role names in a trace). TracerTraj follows the official v1.0.0 train/test split and statistics are computed on the test split; Who&When is used for evaluation only.

Benchmark	Subset	Train	Test	Avg. steps	Avg. agents
Who&When	Algorithm-Generated	–	126	8.7	3.6
	Hand-Crafted	–	58	51.6	5.7
	Total	–	184	22.2	4.3
TracerTraj	Code	1,141	127	20.8	2.3
	Math	567	63	6.4	2.8
	Agentic	1,500	600	9.3	1.0

Table 7: Statistics of the workflow optimization benchmarks. Each benchmark is split 20%/80% into an MCTS validation set and a test set (rounded to whole instances). The HMM for each benchmark is trained on the traces listed in the last column.

Benchmark	Task	Instances	MCTS val.	Test	Metric	HMM training traces
HumanEval Pro	Code	164	33	131	pass@1	ordinary HumanEval
MBPP Pro	Code	378	76	302	pass@1	ordinary MBPP
MATH Lv5	Math	1,324	265	1,059	solve rate	MATH excluding Lv5
SimpleQA Verified	Factual QA	1,000	200	800	accuracy	SimpleQA

F HMM Training Data

All HMMs are trained on same-domain traces that never include the evaluation instances.

Failure attribution. The failure HMM θ^- and the success HMM θ^+ are fit on traces of the same domain as the test set. For Who&When, whose traces come from GAIA (Mialon et al., 2023) and AssistantBench (Yoran et al., 2024), the failure traces are taken from the GAIA and AssistantBench portions of the TracerTraj training split (Zhang et al., 2026). The successful traces come from the same two benchmarks and are produced by the same agent framework that generated the failure traces. For TracerTraj-public, each domain is trained on its official training split, with successful traces generated from the same data source and agent framework as the failure traces of that domain.

Workflow optimization. Each HMM is trained on the correct trajectories obtained by running the AFlow agent framework (Zhang et al., 2025a) on the training problems of the same domain. The HumanEval HMM is trained on ordinary HumanEval (Chen et al., 2021), the MBPP HMM on ordinary MBPP (Austin et al., 2021), the MATH Lv5 HMM on MATH problems excluding Level 5 (Hendrycks et al., 2021), and the SimpleQA HMM on SimpleQA (Wei et al., 2024; Haas et al., 2025). Only trajectories that solve their problem are kept.

G With-Ground-Truth Attribution Results

The main text reports failure attribution accuracy in the without-ground-truth (w/o G) setting only, since it requires no oracle solution and no LLM judge and is therefore closest to real deployment. Table 8 and Table 9 report the with-ground-truth (w/G) accuracy for the same methods, for comparison against the w/o G numbers in Table 1 and Table 2. For HALT, the w/o G protocol reports the bare HMM's first alarm. The w/G protocol passes the HMM's alarms, in order, to a bounded LLM judge given the ground-truth solution: if the judge confirms an alarm, HALT reports that step, and if it rejects the alarm, HALT resumes scanning the trace and checks the next alarm. The ground truth is

Table 9: Who&When with ground truth (w/G). Compare against the w/o G numbers in Table 2. N/A denotes unavailable or inapplicable cells.

	Method	Algorithm-Generated		Hand-Crafted	
		Agent-level	Step-level	Agent-level	Step-level
Offline	*GPT-4o*				
	All-at-Once	54.33	12.50	55.17	5.27
	Step-by-Step	35.20	25.51	34.48	7.02
	Binary Search	44.13	23.98	51.72	6.90
	DeepSeek-V4-Pro				
	All-at-Once	57.94	32.06	43.10	17.24
	Step-by-Step	55.56	31.27	46.55	15.52
	Binary Search	42.06	27.78	46.55	6.90
	AgenTracer	69.62	42.86	69.10	20.68
On.	MASC	N/A	19.24	N/A	18.25
	HALT	**70.63**	**45.24**	**72.41**	**24.14**

consulted only after an alarm is raised, so it can filter the HMM's alarms but never proposes a step on its own.

Ground-truth assistance does not help the baselines consistently. On Who&When, it raises accuracy in 13 of the 15 baseline cells on Algorithm-Generated but in only 6 of 15 on Hand-Crafted, and its effect can be large in either direction: GPT-4o Binary Search gains 14.0 and 15.5 agent-level points on the two subsets, while DeepSeek-V4-Pro Binary Search loses 13.8 agent-level points on Hand-Crafted (from 60.34% to 46.55%). On TracerTraj, ground truth lowers more baseline cells than it raises (18 vs. 13), with every change within 5 points. HALT, in contrast, never loses accuracy with ground truth. Its w/G accuracy matches or exceeds its w/o G accuracy in every column, by at most 3.2 points, which is consistent with the protocol above, where the ground truth only filters alarms that the HMM has already raised. HALT remains the best method in every comparable column of both w/G tables, so ground-truth assistance does not change our conclusions. We include the w/G variant to align with the evaluation protocol used by prior benchmarks (Zhang et al., 2025b; 2026).

ProMAS has no w/G setting, so its row is omitted entirely from these tables (Zhao et al., 2026b). MASC's prototype-guided next-execution reconstruction scores each step for anomaly but performs no agent identification (Shen et al., 2026), so its Agent-level cells are N/A rather than a step-derived agent guess. Agentic agent-level cells are N/A because Agentic is a single-agent setting. The public TracerTraj v1.0.0 release includes ground-truth answers for Code and Math; for Agentic we backfill answers from upstream task sources and evaluate the w/G setting on all 600 test traces. We omit an AgenTracer row on TracerTraj because AgenTracer's published TracerTraj results use a private split, and no public-release results or model parameters are released (Zhang et al., 2026).

Table 8: TracerTraj-public with ground truth (w/G). Compare against the w/o G numbers in Table 1. N/A denotes unavailable or inapplicable cells.

	Method	Code		Math		Agentic	
		Agent	Step	Agent	Step	Agent	Step
Offline	*GLM-5.1*						
	All-at-Once	61.89	8.66	58.73	39.21	N/A	18.00
	Step-by-Step	62.68	13.39	60.32	40.79	N/A	20.50
	Binary Search	62.78	7.87	60.32	40.79	N/A	17.33
	DeepSeek-V4-Pro						
	All-at-Once	61.10	11.81	55.56	46.03	N/A	18.00
	Step-by-Step	62.68	12.60	55.56	46.03	N/A	14.50
	Binary Search	51.18	4.72	58.25	37.62	N/A	13.00
On.	MASC	N/A	4.72	N/A	22.22	N/A	20.67
	HALT	**65.35**	**21.26**	**66.67**	**47.62**	**N/A**	**22.50**