# POLICYFORK: TASK SUCCESS CAN HIDE POLICY VIOLATIONS IN AGENT DELEGATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

AI agents use protocols such as Agent2Agent (A2A) to delegate tasks across independently built systems. Task policies govern which agents may use a tool or service and under what conditions. Runtime systems can check submitted operations against these policies. Before that check, the receiving agent chooses an operation using a *handoff*, the task and context sent with the delegation. If the handoff omits a required record, the agent may reject permitted work or attempt a forbidden operation. Agent evaluation must distinguish this information gap from a model error.

To address this problem, we present POLICYFORK, the *first* framework to verify whether agent handoffs determine policy decisions. POLICYFORK adds the fewest available fields needed to determine each candidate action's decision and rechecks the repaired handoff. We also construct *decision-sufficient execution state* (DSES), a shortest fixed-length encoding that communicates these decisions to the recipient. We implement eight software-delivery policies from GitHub, GitLab, Amazon Web Services, Azure, and Google Cloud in local A2A workflows. Our release tests show that repaired handoffs let a rule-based receiver accept 28/28 permitted state–action combinations, up from 16/28, with zero false Permits. In paired trials, adding records raises correct Permits on allowed requests from 0/48 to 48/48 for each of four model configurations. POLICYFORK separates missing information from model errors and exposes policy violations that task-success scores miss. Our data and code will be made available at https://anonymous.4open.science/r/PolicyFork-ICLR27/

## 1 INTRODUCTION

AI agents delegate work to other agents that have the tools and permissions needed to complete a task (Muruaga, 2026). The Agent2Agent (A2A) protocol supports this collaboration across independently built systems. Agents exchange tasks and results without requiring access to one another's internal memory or tools (A2A Project, 2026b). A *handoff* supplies the receiving agent with the task and context, such as policy text and records of earlier operations. Delegated work remains subject to *task policies* governing who may perform an operation, on which resources, and under what conditions.

These policies can depend on what other participants have done. For example, deployment may require a successful code check for the requested revision and independent approval. GitHub lets owners require approval by someone other than the initiator (GitHub Actions, 2026). A handoff may report that the code check passed while omitting whether independent approval was obtained. Without that approval information, the recipient cannot tell whether deployment is allowed. Deploying without approval violates the policy, while rejecting an approved deployment blocks legitimate work. The receiving agent therefore needs both the policy and the records that establish whether its prerequisites hold.

Existing work evaluates policy compliance (Yao et al., 2025; Levy et al., 2026). Runtime systems such as FORGE can block forbidden calls and return corrective feedback (Palumbo et al., 2026). Before submitting a call, however, the recipient must choose among available operations. Different operations can require different records. Developers therefore need to know whether the handoff contains the records needed to decide among those operations.



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

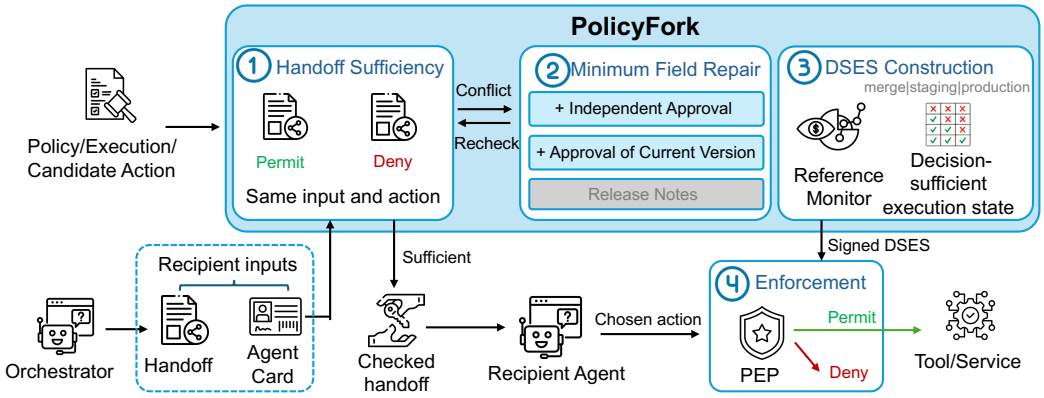


Figure 1: **PolicyFork overview.** The orchestrator prepares the handoff. The checker compares complete recipient inputs, including visible Agent Cards, and rechecks field repairs. Blue plus signs mark the fields selected for addition. DSES columns represent merge, staging, and production. The policy enforcement point (PEP) checks the chosen action using signed DSES.

**Our approach.** To address this problem, we present PolicyFork, the *first* framework to verify whether agent handoffs determine policy decisions (Figure 1). Given an executable policy and a finite set of execution histories, the *model-independent handoff checker* in PolicyFork compares histories with identical complete recipient inputs. It checks every tool call the recipient may choose next, termed a *candidate action*. If the same call requires opposite policy decisions, PolicyFork returns the pair for replay. Otherwise, it verifies sufficiency for the supplied executions and actions.

Our key insight is that a sufficient handoff must distinguish executions requiring different policy decisions for the same action. Our *exact handoff criterion* makes this requirement precise across all candidate actions. We use it to select the fewest available fields that separate conflicting pairs, obtaining a *minimum field repair*. PolicyFork preserves the original handoff content, adds the selected fields, and checks the resulting input across every candidate action. If the original handoff already passes this check, PolicyFork adds no fields. We also construct *decision-sufficient execution state* (DSES), a shortest fixed-length encoding of the candidate-action decisions.

To evaluate handoff checking and repair, we implement eight software-delivery policies from GitHub, GitLab, Amazon Web Services (AWS), Azure, and Google Cloud in local A2A workflows. In our release study, PolicyFork adds a field for independent approval and another for whether approval was recorded for the requested revision, then checks merge, staging, and production (Table 3). On an independently authored GitHub policy, it restores omitted fields and leaves handoffs containing both deployment status and environment unchanged (Section 6.1). We then evaluate how cross-agent records affect models' Permit/Deny answers and task completion.

**Our Contributions.**

1. We introduce a **model-independent handoff checker** within PolicyFork. Its counterexamples prove information gaps for every model receiving the same inputs. When no conflicting pair exists, it verifies sufficiency for the supplied executions and candidate actions.
2. We establish an **exact handoff criterion** and compute **minimum field repairs**. The criterion characterizes sufficient handoffs and yields both the fewest added fields and the shortest fixed-length DSES encoding for the candidate actions.
3. We implement and evaluate **handoff checking and repair** in source-derived A2A workflows. Repaired handoffs let a rule-based receiver accept all 28 permitted state–action combinations, up from 16, and reject all 68 forbidden combinations. Paired evaluations across four model configurations separate missing information from errors in using it, while model-chosen executions expose policy violations that task-success scores miss.

## 2 Related Work

Policy compliance depends on both the agent's choice and the information available when it chooses. Prior work measures compliant behavior, studies decisions under limited information, retrieves missing evidence, and controls tool calls. Table 1 compares these approaches across four design points.

Table 1: **Comparison across four design points:** policy compliance, recipient-input sufficiency (whether the input before action choice determines every candidate action's policy answer), handoff repair, and runtime enforcement.

| Work | Policy compliance | Recipient-input sufficiency | Handoff repair | Runtime enforcement |
|---|---|---|---|---|
| ST-WebAgentBench (Levy et al., 2026) | ● | ○ | ○ | ○ |
| AgentDojo (Debenedetti et al., 2024) | ● | ○ | ○ | ◑ |
| CaMeL (Debenedetti et al., 2026) | ● | ◑ | ○ | ● |
| FORGE (Palumbo et al., 2026) | ● | ○ | ○ | ● |
| Bounded Agents (Muruaga, 2026) | ● | ○ | ○ | ● |
| SMT-TaLLM (Winston et al., 2026) | ● | ○ | ○ | ● |
| AgentFlow (Shivakumar et al., 2026) | ● | ○ | ○ | ● |
| Handover (theory) (Kato & Kato, 2026) | ○ | ● | ○ | ○ |
| Scientific-RAM (Anonymous, 2026) | ○ | ◑ | ● | ○ |
| PolicyFork (ours) | ● | ● | ● | ● |

● Supported ◑ Partially supported ○ Not addressed

**Policy compliance.** Policy-compliance evaluations measure policy following and user–agent coordination (Yao et al., 2025; Barres et al., 2026), policy-valid completion from incomplete observations (Levy et al., 2026), and utility under attack (Debenedetti et al., 2024). A refusal of a permitted deployment means something different when the approval record was never sent than when the agent received and ignored it. The recipient's input shows which failure the evaluation measures.

**Information sufficiency.** Partial-observation control studies what agents can decide from their local views (Rudie & Wonham, 1992; Mostafa & Bonakdarpour, 2015). Handover characterizes records sufficient for later predictions (Kato & Kato, 2026), and CaMeL lets its parsing model flag missing information (Debenedetti et al., 2026). In delegated work, the practical question is whether the recipient's input determines which of its available tool calls the policy permits.

**Handoff repair.** Scientific-RAM checks role-specific handoff packets and retrieves missing evidence for a specified downstream action (Anonymous, 2026). A recipient may choose a different call after the handoff, so fixing one answer can leave another unresolved. A repair should cover the available calls and add nothing when the original input already settles them.

**Runtime enforcement.** Runtime systems check proposed operations. FORGE evaluates executable policies using information supplied by the environment (Palumbo et al., 2026). The Open Agent Passport (OAP) intercepts tool calls and issues signed authorization records (Uchibeke, 2026). AgentBound combines delegated authority, owner rules, and site action contracts into decisions and receipts (Kaul et al., 2026). SMT-TaLLM provides corrective feedback, and AgentFlow verifies bounded paths (Winston et al., 2026; Shivakumar et al., 2026). These approaches evaluate or constrain proposed actions and execution paths. Before proposing a call, the recipient needs the records on which its choice depends.

## 3 Problem Formulation

We define the workflow, trust assumptions, and information needed to determine whether a candidate tool call is permitted.

### 3.1 SYSTEM MODEL

The *orchestrator* coordinates the task and prepares a handoff from earlier agents' results. The *recipient agent* combines the handoff with its local information, selects an action, and submits it as a tool call. The tool or service executes accepted requests and produces *records* identifying the operations, actors, resources, and versions. We call an operation *committed* once its effect takes place and is recorded. The orchestrator can include these records in subsequent handoffs.

An A2A *Agent Card* describes capabilities and authentication requirements (A2A Project, 2026b;a). Credentials identify the caller, and delegated *scope* specifies allowed operations and resources (Microsoft, 2026b; Okta, 2026; South et al., 2025). A2A leaves task-specific authorization to each implementation (A2A Project, 2026b).

POLICYFORK checks the recipient's information before it chooses an action (Figure 1). A *reference monitor* provides the expected policy answers by evaluating the recorded operations. At execution time, a resource-side *policy enforcement point* (PEP) checks the submitted request before the tool or service executes it.

### 3.2 THREAT MODEL

Delegated agent workflows face risks both before and after an action is chosen. The recipient may receive incomplete or manipulated information about earlier operations, while a caller may later attempt an action that violates the task policy. Therefore, we consider threats to both handoff information and resource-side execution. Next, we specify the adversary's capabilities, the trusted components, and the resulting design goals.

**Adversary capabilities.** The adversary can control the orchestrator or the caller. The orchestrator can omit records or alter or replay forwarded messages. A caller with valid credentials can choose operations, resources, and arguments and sign requests. An attacker can also steer an agent's choice through content the agent reads. A published GitHub issue, for example, led an agent to read private repositories and publish their contents (Milanta & Beurer-Kellner, 2025).

**Trusted components.** The checker uses the policy and records supplied by the developer. For deployment, each field names a source and extraction rule designated by the policy owner. Source authentication binds issuer, content, and relevant scope; the deployment trusts the designated source to report events truthfully. To establish that an event did not occur, the designated source must cover the task, resource, and version in question. The monitor, DSES compiler, registered keys, state management, and PEP are trusted; every protected effect passes through the PEP.

**Design goals.** An omitted record can hide whether a call is permitted. The caller can also submit a forbidden request afterward. These risks set three goals for POLICYFORK: (1) Find two histories with the same recipient input but opposite policy answers for one call. (2) If so, add the fewest available fields that settle all candidate calls, then recheck the handoff. (3) Have the resource-side PEP reject forbidden requests before execution.

### 3.3 HANDOFF CHECKING

The developer supplies histories, complete recipient inputs, candidate actions, and an executable policy. The checker compares the supplied histories to see whether the same recipient input ever requires opposite policy decisions for one of those actions.

**Policy validity beyond task success.** We score task completion and policy validity separately (Levy et al., 2026). The set $\mathcal{F}$ contains a task's feasible *committed execution histories*. Each history $h = e_1 \cdots e_t \in \mathcal{F}$ records operations that took effect, excluding proposals and rejected calls. The task completes when the environment state $s(h)$ reached after $h$ belongs to its goal set $G$. The policy $\varphi$ permits histories in $L_\varphi$. Task success $S$, policy validity $A$, and valid completion $U$ are

$$S(h) = \mathbf{1}[s(h) \in G], \quad A(h) = \mathbf{1}[h \in L_\varphi], \quad U(h) = S(h)A(h). \tag{1}$$

If an agent completes deployment before obtaining the required approval, Equation 1 gives $S = 1$ but $A = U = 0$. Our finite-state safety policies are *prefix-closed*: once an effect violates the policy, later effects cannot undo the violation.

**Policy decisions.** After a policy-valid history $h$, the reference monitor has state $q(h)$ in its state set $Q$ and returns $g(q(h), u) \in \{\text{PERMIT}, \text{DENY}\}$ for candidate action $u$. The action specifies the operation, resource, caller, revision, and relevant arguments. Permit means that executing $u$ preserves policy validity. Appendix A.1 defines the monitor's transitions and the formal enforcement objective.

**Same input, opposite required decisions.** POLICYFORK checks whether the recipient can distinguish executions that require opposite policy decisions. In the approval example, two runs report the same successful code check for the requested revision. Only one includes the required independent approval. The recipient's *complete input* contains the handoff and everything else it can see before choosing a call: Agent Cards, credentials, delegated scope, policy text, and local records. It need not contain the full execution history. We write $O_i^{\text{pre}}(h)$ for this input before repair fields or DSES are added. The recipient chooses action $u$ afterward, so a decision rule receives $u$ separately. Ordinary authorization $\lambda_i(o, u) = 1$ checks caller authority, request parameters, and local tool preconditions. It does not check cross-agent policy prerequisites. Two valid histories $h^+, h^- \in \mathcal{F} \cap L_\varphi$ form a *delegation fork* when

$$O_i^{\text{pre}}(h^+) = O_i^{\text{pre}}(h^-) = o, \qquad \lambda_i(o, u) = 1, \\ g(q(h^+), u) \neq g(q(h^-), u). \tag{2}$$

Equation 2 holds the complete input and ordinary authorization fixed while policy decisions differ. The counterexample identifies an information gap before model behavior is evaluated.

## 4 POLICYFORK FRAMEWORK

We now give an exact test for whether the recipient knows which calls the policy permits. That test supports field repair and DSES. Resource enforcement then checks the call the recipient actually chooses.

Table 2: **Different actions require different conditions.** Rows give the prerequisites that hold. Code checks concern the requested revision. Approval means independent approval; revision approval means approval recorded for that revision.

| Current prerequisite state | Merge | Staging | Production |
|---|---|---|---|
| No successful code check | DENY | DENY | DENY |
| Code check passed; no approval | PERMIT | DENY | DENY |
| Code check + approval; no revision approval | PERMIT | PERMIT | DENY |
| All three prerequisites present | PERMIT | PERMIT | PERMIT |

### 4.1 HANDOFF SUFFICIENCY

We begin with the first design goal: determine whether the recipient's input settles the policy answer for every call it might choose. Merge requires a successful code check for the requested revision. Test deployment (staging) also requires independent approval; live deployment (production) further requires approval recorded for that revision. These approval conditions check who approved and which revision the approval covers. A sufficient handoff determines the full row in Table 2. We group candidate actions in an *action interface* $I = (i, o, \mathcal{C}_I)$, where $i$ is the recipient, $o$ its original input, and $\mathcal{C}_I$ a nonempty finite set of actions satisfying $\lambda_i(o, u) = 1$.

The same input $o$ can arise from different execution histories. Let $Q_I = \{q(h) : h \in \mathcal{F} \cap L_\varphi,\ O_i^{\text{pre}}(h) = o\}$ be the nonempty finite set of compatible monitor states. For $q \in Q$, the *decision profile* $R_I(q)(u) = g(q, u)$ collects the answers over $u \in \mathcal{C}_I$. On $Q_I$, define

$$q \equiv_I q' \iff \forall u \in \mathcal{C}_I,\ g(q, u) = g(q', u), \qquad N_I(o) = |\{R_I(q) : q \in Q_I\}|. \tag{3}$$

Equation 3 groups states that agree on every candidate action. If $N_I(o) = 1$, the input determines every action's policy decision. If $N_I(o) > 1$, the checker can return a delegation fork.

**Theorem 1** (Model-independent input limit)**.** *If $N_I(o) > 1$, there exist $q^+, q^- \in Q_I$ and $u \in \mathcal{C}_I$ with $g(q^+, u) = \text{PERMIT}$ and $g(q^-, u) = \text{DENY}$. Every binary rule $d$ using only $(o, u)$ satisfies*

$$\tfrac{1}{2}\Pr[d(o, u) \neq g(q^+, u)] + \tfrac{1}{2}\Pr[d(o, u) \neq g(q^-, u)] = \tfrac{1}{2},$$

*where probabilities are over the rule's internal randomness. Thus no such rule is correct on all of $Q_I \times \mathcal{C}_I$. A three-valued rule $\widetilde{d}$ with no wrong definite answers on either state must satisfy $\Pr[\widetilde{d}(o,u) = \bot] = 1$, where $\bot$ means* Undetermined.

For any fixed model, identical inputs give the same output distribution. Consequently, resolving a fork requires additional information, such as a record from the approval service.

A handoff *encoder* $m_I$ maps monitor states to messages. Its *decoder* $D_I$ returns a policy decision for a message and candidate action. The encoding is *exact* when every decoded decision matches the policy:

$$D_I(m_I(q), u) = g(q, u) \quad \text{for every } (q, u) \in Q_I \times \mathcal{C}_I. \tag{4}$$

**Theorem 2** (Exact handoff criterion)**.** *For an encoder $m$ on $Q_I$, an exact decoder exists if and only if $m(q) = m(q') \Rightarrow R_I(q) = R_I(q')$. The coarsest exact encoder merges all and only equal profiles:*

$$m_I(q) = m_I(q') \iff q \equiv_I q'. \tag{5}$$

*It uses $N_I(o)$ message values, the minimum among exact encodings.*

Appendix A proves these guarantees and extends the analysis to policy checking from separate agents' observations.

### 4.2 Minimum field repairs

To meet the second goal, a repair must distinguish every pair the checker finds with identical inputs and different policy answers. Let $\mathcal{H} \subseteq \mathcal{F} \cap L_\varphi$ be the finite set of supplied policy-valid histories with the same candidate set $\mathcal{C}_I$. Before repair, the developer supplies typed, named fields with designated sources and deterministic extraction rules. Their values on history $h$ are $f_j(h)$, for $j = 1, \ldots, p$. We minimize the number of added fields over this catalog, the supplied histories, and the fixed candidate-action set. Every field must have a value for every history; a missing value fails input validation. Table 11 maps the release fields to their source records and scope bindings. The pairs that a repair must separate are

$$\mathcal{W}_i = \big\{\{h, h'\} \subseteq \mathcal{H} : O_i^{\mathrm{pre}}(h) = O_i^{\mathrm{pre}}(h'),\ \exists u \in \mathcal{C}_I :\ g(q(h), u) \neq g(q(h'), u)\big\}. \tag{6}$$

For each pair, $\Delta(h, h') = \{j : f_j(h) \neq f_j(h')\}$ contains the indices of fields whose values differ between the two histories. A minimum field repair is

$$J^* \in \arg\min_{J \subseteq \{1,\ldots,p\}} |J| \quad \text{subject to} \quad J \cap \Delta(h, h') \neq \varnothing \quad \forall \{h, h'\} \in \mathcal{W}_i. \tag{7}$$

Equation 7 selects the fewest fields that separate all conflicting pairs, a hitting-set problem (Crowston et al., 2012). By Theorem 2, this constraint is necessary and sufficient for the repaired input to determine every candidate action's answer (Lemma 7). PolicyFork searches subsets in increasing order of field count and rechecks all actions. If the input already settles every action, the search adds nothing. If no available field resolves a conflict, PolicyFork returns the pair.

### 4.3 DSES construction

Field repair adds fields derived from designated source records. DSES encodes the policy decisions directly, using one code per distinct profile (Theorem 2). The *Relevant facts* baseline transmits the Boolean values of three prerequisites: code-check success, independent approval, and approval recorded for the requested revision. A *verdict bitset* stores one Permit/Deny bit per candidate action. Both determine the full decision profile. Indexing the reachable bitsets gives the same minimum-length code as DSES. The code needs at least $\lceil \log_2 N_I(o) \rceil$ bits to distinguish the $N_I(o)$ profiles (Kushilevitz & Nisan, 1997):

$$b_I^*(o) = \lceil \log_2 N_I(o) \rceil. \tag{8}$$

Algorithm 1 constructs the exact encoder and decoder in Equation 4 at the bit bound in Equation 8, using $|Q_I|\,|\mathcal{C}_I|$ policy queries. The version identifier $v_I$ ties the tables to their policy, recipient, and original input.

Shared prerequisites can reduce both the fields and decision bits needed. Appendix B.3 gives the curves and their derivation.

**Algorithm 1** Compile DSES for all candidate actions

**Require:** $I = (i, o, \mathcal{C}_I)$, compatible reachable states $Q_I$, policy decision function $g$
1: $R_I(q) \leftarrow (g(q,u))_{u \in \mathcal{C}_I}$ for every $q \in Q_I$
2: assign one code to each distinct profile, in deterministic order
3: $m_I(q) \leftarrow \mathsf{code}(R_I(q))$ for every $q \in Q_I$
4: store $D_I(m_I(q), u) = R_I(q)(u)$ for every $(q,u) \in Q_I \times \mathcal{C}_I$
5: bind policy, recipient, input, states, and ordered action/decoder tables in version $v_I$
6: **return** $(m_I, D_I, v_I)$

Two states with the same current decisions can respond differently to a later event. The reference monitor therefore updates its state and recomputes DSES after each commit.

**Acting with incomplete records.** The recipient may know whether merge is allowed while still lacking approval information for deployment. For a message $\nu$, let $Q_I(\nu)$ contain the states in $Q_I$ consistent with its contents. If this set is nonempty, the largest set of actions that can safely proceed is

$$\mathcal{P}_I(\nu) = \bigcap_{q \in Q_I(\nu)} \{u \in \mathcal{C}_I : g(q,u) = \textsc{Permit}\}. \tag{9}$$

Equation 9 permits an action only when every compatible state permits it. An omitted record alone does not establish that its event did not occur. These Permits are correct if the compatible set contains the true state. The monitor issues DSES when the compatible states form a nonempty set with one decision profile.

### 4.4 Resource Enforcement

Even a sufficient handoff cannot control which request the caller finally submits. The monitor signs a DSES message (the *envelope*), and the caller sends it with a signed request for its chosen action. The PEP checks authority, signatures, action parameters, and whether the message still applies to the current state. The PEP executes the call only if the decoded decision is Permit. Each commit advances the state version. A serialized transaction records the local effect and prevents intervening state changes between verification and execution. Under Section 3.2's trust assumptions, every commit preserves policy validity (Corollary 5). Appendix D specifies the request checks and the execution conditions needed to apply the policy to local or remote effects.

## 5 Evaluation Setup

Our evaluation assesses the effectiveness of *handoffs* in providing necessary information, whether models use it correctly, and the capacity of resources to prevent prohibited actions. Results are based on executable policies and tool records. Appendix C provides information about the sources, settings, scoring, and errors.

**Handoff evaluation.** We test 32 states and three release actions before and after repair; the receiver permits only actions allowed in every compatible state. We also evaluate an original GitHub policy with GitHub's expression evaluator (QAStudio-Dev, 2026b; GitHub, 2026b).

**Fixed-action decisions.** For each policy, one valid history that permits the action is paired with three valid histories that forbid it; the original input and action stay fixed within each pair. The *Policy* input includes the policy; *History* adds committed cross-agent records. Two repetitions yield 192 judgments per model: 48 permitted and 48 forbidden requests per condition. Missing or malformed answers count as incorrect, not as definite Permit or Deny. We evaluate GLM-4.5-Air, GLM-4.6V, Qwen3.7 Max, and DeepSeek V4 Pro with fixed settings.

**Model-chosen execution.** An LLM orchestrator chooses operations; our runner executes them through the official A2A software development kit (SDK) at local authenticated endpoints (A2A Project, 2026a). Two repetitions of 16 instances give 32 trajectories per condition and model. *Observe* applies ordinary authorization without task-policy enforcement. We score task success and policy validity, including runs with output failures.

**Resource enforcement.** Each recorded request is replayed from the same policy-valid state under Observe, *Online*, and *DSES*. Online requires the current monitor's Permit. Under DSES, the resource-side decoder evaluates the model's selected action using the monitor's signed message.

## 6 RESULTS

We first test whether the handoff supports every candidate call and whether repair fills an information gap. We then measure how agents use those records and whether the resource stops forbidden calls. A sufficient message alone does not force an agent to choose correctly.

### 6.1 HANDOFF REPAIR

One release handoff must support three choices: merge, staging, and production. They require different records. Across 32 states, the policy permits 28 of the 96 state–action pairs and forbids 68. Given only the code-check result, the rule-based receiver in Equation 9 can safely permit 16 of the 28 allowed pairs; adding independent-approval status raises this to 24. POLICYFORK adds whether that approval covers the requested revision and rechecks all three choices. It now permits all 28 allowed pairs and none of the 68 forbidden ones (Table 3).

Table 3: **Field repair supports all three operations.** Before/After count accepted permitted combinations out of 28. All 68 forbidden combinations are rejected. Approval means independent approval; revision approval means approval recorded for the requested revision. Dashes mark unevaluated repairs.

| Handoff | Before | Fields added | After |
|---|---|---|---|
| Code check only | 16/28 | Approval + revision approval | 28/28 |
| Code check + approval | 24/28 | Revision approval | 28/28 |
| Production answer | 12/28 | — | — |
| Relevant facts | 28/28 | None | 28/28 |
| Verdict bitset | 28/28 | None | 28/28 |
| DSES | 28/28 | None | 28/28 |

A handoff containing only the production Permit/Deny answer lets the receiver accept 12 of the 28 permitted combinations across all three actions; some merge and staging decisions remain unresolved. Relevant facts, the verdict bitset, and DSES already settle all three actions, so the checker leaves them unchanged. If no available field resolves a conflict, it returns the conflicting pair.

**Reuse on an external policy.** On QAStudio's independently authored workflow, our checker matches the saved Production outcome and a public Preview run whose test job is skipped (QAStudio-Dev, 2026a;b). Across twelve inputs, it repairs status-only and environment-only handoffs and preserves handoffs with both fields. Appendix C.6 describes the runnable integration example.

### 6.2 AGENT PERFORMANCE

The next test asks whether these records also change model decisions. In each pair, the Policy input gives the rule but not the record needed to decide; History supplies committed cross-agent records. Policy leaves the answer unresolved (Theorem 1), and all four model configurations deny all 48 permitted requests. With History, each correctly permits 48/48 (Table 4).

Table 4: **Cross-agent records restore correct Permits.** Each condition has 48 permitted and 48 forbidden requests per model. Green marks restored correct Permits, red wrong Permits, and gray missing answers.

| Policy → History | Allowed action | Forbidden action | |
|---|---|---|---|
| Model | Correct Permit | Wrong Permit | No answer |
| GLM-4.5-Air | 0/48 → 48/48 | 0/48 → 0/48 | 0/48 → 1/48 |
| GLM-4.6V | 0/48 → 48/48 | 0/48 → 1/48 | 0/48 → 3/48 |
| Qwen3.7 Max | 0/48 → 48/48 | 0/48 → 0/48 | 0/48 → 0/48 |
| DeepSeek V4 Pro | 0/48 → 48/48 | 0/48 → 0/48 | 0/48 → 0/48 |

GLM-4.6V permits a merge using an earlier approval despite seeing both revision identifiers. Four GLM replies hit the token limit; Qwen and DeepSeek answer all requests correctly.

**When agents choose their own calls,** cross-agent records raise valid completions from 70/128 to 124/128 and reduce interrupted runs from 56 to two. Two GLM runs per condition complete the task before approval.

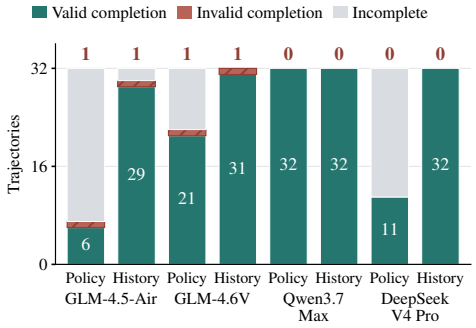


Figure 2: **Completed tasks can still violate policy.** Red numbers count invalid completions; white numbers count valid completions. Each bar contains 32 trajectories; gray is incomplete.

### 6.3 DSES EVALUATION

The recipient may choose any of the three calls, so the message must preserve all three policy answers. Yet the 32 release states produce only four distinct answer profiles. DSES can therefore represent them in two bits (Equation 8; Figure 3). Even the best fixed choice of two source fields leaves one in seven permitted pairs unresolved; DSES and indexed verdict bitsets leave none unresolved.

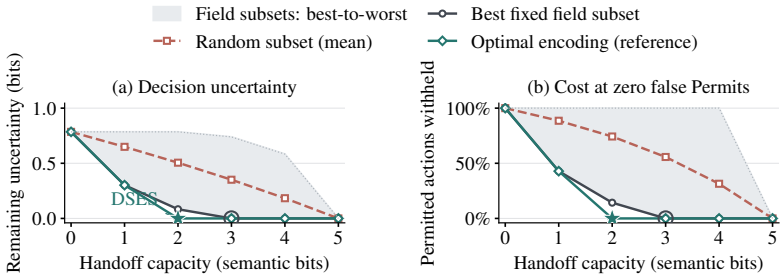


Figure 3: **Two bits determine all three policy decisions.** Gray spans all fixed field subsets, not a confidence interval. The green stars mark DSES and rings the verdict bitset. All 32 states and three actions have uniform weights.

**Compiler and resource checks.** Finally, the policy answer must still govern the call the agent submits. The compiler agrees with an independent evaluator on 5,870 histories and processes 65,536 states and twelve actions in 0.80–0.82 seconds (Appendix B). On 178 same-state replays, Online and DSES both execute 156 permitted calls and reject 22 forbidden ones; Observe executes all 178.

## 7 DISCUSSION AND CONCLUSION

**Evidence in a handoff.** Paired tests separate missing records from misuse: cross-agent records restore correct Permits, yet GLM-4.6V reuses an earlier approval despite both revision identifiers. The checker tests handoff sufficiency; the resource-side PEP blocks forbidden effects, addressing OWASP's excessive-permission and high-impact action risks (OWASP GenAI Security Project, 2025).

**A practical adoption path.** Deployment can start with one environment-gated workflow. An adapter maps existing checks and approval records into the handoff; the checker tests candidate actions and adds selected fields. The resource retains its authorization gate, as in GitHub Environments (GitHub Actions, 2026). Our A2A and GitHub tests exercise these components; shadow mode can measure record coverage and latency.

**When the workflow changes.** Policy owners designate the records that establish each prerequisite; orchestrators decide which records reach the recipient; resource operators enforce the selected call. Developers can rerun the handoff check when a policy or tool interface changes and rebuild DSES before using the new interface. This makes information needed for a decision visible during ongoing risk review (National Institute of Standards and Technology, 2024). POLICYFORK makes the handoff itself testable: developers can find an information gap, add the fewest available fields, and verify the repaired input before scoring the receiving agent's policy decisions.

### ETHICS STATEMENT

All model-directed effects execute in isolated local environments using fictitious actors, resources, and test credentials. No production deployment is modified. Public GitHub records support a separate read-only case study. No human subjects or private organizational data are involved. Provider keys and production credentials are excluded from the research package.

### AI USE STATEMENT

In this work, LLMs serve primarily as research subjects. We also used generative AI tools to assist with code polishing and language revision. The authors reviewed the AI-assisted code and prose and tested the resulting code changes. We take responsibility for the final manuscript and research artifacts.

### REPRODUCIBILITY STATEMENT

The research package includes frozen source policies, model and resource records, and offline scripts. Appendix E gives a runnable handoff integration and the saved-record reproduction entry points. Appendices A.2 and C supply proofs and source-policy mappings. Our data and code are available at https://anonymous.4open.science/r/PolicyFork-ICLR27/

## REFERENCES

A2A Project. Official A2A Python SDK. GitHub repository, 2026a. URL https://github.com/a2aproject/a2a-python. Experiments use package a2a-sdk 1.1.2.

A2A Project. A2A protocol specification. Development specification, version 1.0 semantics, 2026b. URL https://a2a-protocol.org/dev/specification/. Accessed 2026-09-21.

Alibaba Cloud. Model Studio: DeepSeek API, 2026a. URL https://www.alibabacloud.com/help/en/model-studio/deepseek-api. Accessed 2026-09-06.

Alibaba Cloud. Model Studio: Qwen3.7-Max, 2026b. URL https://help.aliyun.com/en/model-studio/qwen3-7-max. Accessed 2026-09-07; dated endpoint 2026-05-20.

Amazon Web Services. Manual approvals in AWS CodePipeline. AWS CodePipeline User Guide, 2026. URL https://docs.aws.amazon.com/codepipeline/latest/userguide/approvals.html. Accessed 2026-09-02.

Anonymous. Scientific-RAM: Research agents need role-aware handoff memory. Anonymous ACL submission, 2026. URL https://openreview.net/pdf?id=a9Y241ArJX.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. $\tau^2$-Bench: Evaluating conversational agents in a dual-control environment. In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2506.07982.

Robert Crowston, Gregory Gutin, Mark Jones, Saket Saurabh, and Anders Yeo. Parameterized study of the test cover problem. In *Mathematical Foundations of Computer Science 2012*, volume 7464 of *Lecture Notes in Computer Science*, pp. 283–295, 2012. doi: 10.1007/978-3-642-32589-2_27. URL https://doi.org/10.1007/978-3-642-32589-2_27.

Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *International Joint Conference on Artificial Intelligence*, pp. 854–860, 2013. URL https://www.ijcai.org/Proceedings/13/Papers/132.pdf.

Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/97091a5177d8dc64b1da8bf3e1f6fb54-Paper-Datasets_and_Benchmarks_Track.pdf.

Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. Defeating prompt injections by design. In *IEEE Conference on Secure and Trustworthy Machine Learning*, 2026. URL https://floriantramer.com/publications/camel25/.

GitHub. Events that trigger workflows: Deployment status, 2026a. URL https://docs.github.com/en/actions/reference/workflows-and-actions/events-that-trigger-workflows#deployment_status. Accessed 2026-09-10.

GitHub. Actions expressions: Parser and evaluator, 2026b. URL https://github.com/actions/languageservices/tree/main/expressions. Package @actions/expressions version 0.3.61, MIT license.

GitHub. About protected branches. GitHub Documentation, 2026c. URL https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-protected-branches/about-protected-branches. Accessed 2026-09-02.

GitHub Actions. Deployments and environments. GitHub Actions Documentation, 2026. URL https://docs.github.com/en/actions/reference/workflows-and-actions/deployments-and-environments. Accessed 2026-09-02.

GitLab. Deployment approvals. GitLab Documentation, 2026. URL https://docs.gitlab.com/ci/environments/deployment_approvals/. Accessed 2026-09-02.

Google Cloud. Promote a release. Cloud Deploy Documentation, 2026. URL https://cloud.google.com/deploy/docs/promote-release. Accessed 2026-09-02.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 3 edition, 2006. ISBN 9780321455369. URL https://www.pearson.com/en-us/subject-catalog/p/introduction-to-automata-theory-languages-and-computation/P200000003517/9780321455369.

Masahiro Kato and Taka Kato. Handover of in-context learning state across session boundaries. *arXiv preprint arXiv:2608.14528*, 2026. URL https://arxiv.org/abs/2608.14528v1. Version 1.

Anuj Kaul, Qianlong Lan, and Pranay Gupta. Behavioral governance for autonomous AI agents: The AgentBound framework. *arXiv preprint arXiv:2606.30970*, 2026. URL https://arxiv.org/abs/2606.30970.

Jan Komenda and Tomáš Masopust. Computation of controllable and coobservable sublanguages in decentralized supervisory control via communication. *Discrete Event Dynamic Systems*, 27(4): 585–608, 2017. doi: 10.1007/s10626-017-0249-6.

Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997. doi: 10.1017/CBO9780511574948.

Ido Levy, Ben Wiesel, Sami Marreed, Alon Oved, Avi Yaeli, Nir Mashkif, and Segev Shlomov. ST-WebAgentBench: A benchmark for evaluating safety and trustworthiness in web agents. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/c6c29e590e3c62e37e6b39cdd6baf2e8-Paper-Conference.pdf.

Microsoft. Approvals and checks. Azure DevOps Documentation, 2026a. URL https://learn.microsoft.com/en-us/azure/devops/pipelines/process/approvals. Accessed 2026-09-02.

Microsoft. Agent identity concepts in Microsoft Foundry. Microsoft Foundry documentation, 2026b. URL https://learn.microsoft.com/en-us/azure/foundry/agents/concepts/agent-identity. Accessed 2026-09-01.

Marco Milanta and Luca Beurer-Kellner. GitHub MCP exploited: Accessing private repositories via MCP. Invariant Labs, May 26, May 2025. URL https://invariantlabs.ai/blog/mcp-github-vulnerability. Attack demonstration. Accessed 2026-09-21.

Menna Mostafa and Borzoo Bonakdarpour. Decentralized runtime verification of LTL specifications in distributed systems. In *IEEE International Parallel and Distributed Processing Symposium*, pp. 494–503, 2015. doi: 10.1109/IPDPS.2015.95. URL https://cse.msu.edu/tart/publication/decentralized-runtime-verification-ltl-specifications-distributed-systems.

Xabier Muruaga. Bounded Agents: Delegation security for multi-agent AI systems. *arXiv preprint arXiv:2608.15888*, 2026. URL https://arxiv.org/abs/2608.15888.

National Institute of Standards and Technology. Artificial intelligence risk management framework: Generative artificial intelligence profile. Technical Report NIST AI 600-1, National Institute of Standards and Technology, 2024. URL https://nvlpubs.nist.gov/nistpubs/ai/NIST.AI.600-1.pdf.

Okta. Set up AI agent token exchange. Okta Developer documentation, 2026. URL https://developer.okta.com/docs/guides/ai-agent-token-exchange/authserver/main/. Accessed 2026-09-01.

OWASP GenAI Security Project. LLM06:2025 excessive agency. OWASP Top 10 for LLM Applications, 2025. URL https://genai.owasp.org/llmrisk/llm062025-excessive-agency/. Accessed 2026-09-23.

Nils Palumbo, Sarthak Choudhary, Jihye Choi, Guy Amir, Prasad Chalasani, and Somesh Jha. Formal policy enforcement for real-world agentic systems. *arXiv preprint arXiv:2602.16708*, 2026. URL https://arxiv.org/abs/2602.16708.

QAStudio-Dev. Studio: Preview Playwright workflow run, 2026a. URL https://github.com/QAStudio-Dev/studio/actions/runs/31236756479. Run 31236756479; accessed 2026-09-10.

QAStudio-Dev. Studio: Playwright tests workflow, 2026b. URL https://github.com/QAStudio-Dev/studio/blob/12d92b0ef1ac68049db1cd8ade330bdec80729a4/.github/workflows/playwright.yml. Pinned source; accessed 2026-09-10.

Karen Rudie and W. Murray Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992. doi: 10.1109/9.173140. URL https://ieeexplore.ieee.org/document/173140/.

Basavesh Ammanaghatta Shivakumar, Swarn Priya, and Peng Gao. AgentFlow: A flow-centric policy language and framework for securing LLM agent systems. *arXiv preprint arXiv:2608.22868*, 2026. URL https://arxiv.org/abs/2608.22868v1. Version 1.

Tobin South, Samuele Marro, Thomas Hardjono, Robert Mahari, Cedric Deslandes Whitney, Alan Chan, and Alex Pentland. Position: AI agents need authenticated delegation. In *International Conference on Machine Learning*, pp. 82211–82231, 2025. URL https://proceedings.mlr.press/v267/south25a.html.

Uchi Uchibeke. Before the tool call: Deterministic pre-action authorization for autonomous AI agents. *arXiv preprint arXiv:2603.20953*, 2026. URL https://arxiv.org/abs/2603.20953.

Cailin Winston, Claris Winston, and René Just. Solver-aided verification of policy compliance in tool-augmented LLM agents. *arXiv preprint arXiv:2603.20449*, 2026. URL https://arxiv.org/abs/2603.20449v1. Version 1.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. $\tau$-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/1b126cc38b8638e07bef37e7b2bb72bf-Abstract-Conference.html.

Z.AI. GLM-4.6V model and API documentation. Model documentation, 2026. URL https://docs.z.ai/guides/vlm/glm-4.6v. Accessed 2026-09-03.

Zhipu AI. GLM-4.5 and GLM-4.5-Air model documentation. Model documentation, 2026. URL https://docs.z.ai/guides/llm/glm-4.5. Accessed 2026-09-03.

# CONTENTS

**1 Introduction** **1**

**2 Related Work** **3**

**3 Problem Formulation** **3**
3.1 System model . . . 4
3.2 Threat model . . . 4
3.3 Handoff checking . . . 4

**4 POLICYFORK Framework** **5**
4.1 Handoff sufficiency . . . 5
4.2 Minimum field repairs . . . 6
4.3 DSES construction . . . 6
4.4 Resource enforcement . . . 7

**5 Evaluation Setup** **7**

**6 Results** **8**
6.1 Handoff repair . . . 8
6.2 Agent performance . . . 8
6.3 DSES evaluation . . . 9

**7 Discussion and Conclusion** **9**

**References** **10**

**A Formal Analysis and Proofs** **15**
A.1 Execution semantics . . . 15
A.2 Handoff sufficiency . . . 15
A.3 Distributed observations . . . 18
A.4 Decision error . . . 18

**B Handoff Evaluation** **20**
B.1 Minimum field repairs . . . 20
B.2 DSES costs . . . 22
B.3 Shared prerequisites . . . 22
B.4 Handoff information budget . . . 23
B.5 Decisions from incomplete records . . . 25

**C Agent Evaluation** **26**
C.1 Source policies . . . 26
C.2 Model settings . . . 27
C.3 Paired decision evaluation . . . 27
C.4 Execution evaluation . . . 28
C.5 Error analysis . . . 28
C.6 External policy integration . . . 29

**D Policy Enforcement** **29**
D.1 Request verification . . . 30
D.2 Execution consistency . . . 31
D.3 Enforcement evaluation . . . 31

**E Reproducibility** **32**

# A Formal Analysis and Proofs

This appendix gives the formal analysis behind handoff checking. We define policy-valid execution, prove the handoff criterion, and show how the guarantees apply to incomplete records and distributed observations. We then separate errors caused by missing information from errors made by the recipient. Appendix B.1 proves the correctness and minimality of field repair.

## A.1 Execution semantics

An instance is $x = (\mathcal{M}, s_0, G, \varphi)$, where $\mathcal{M} = (\mathcal{S}, \Sigma, T)$ is the executable environment. Here $\mathcal{S}$ is the environment-state set, $\Sigma$ is the committed-event alphabet, and $T : \mathcal{S} \times \Sigma \rightharpoonup \mathcal{S}$ is the partial transition function: it gives the next state when the event can occur. The initial state is $s_0 \in \mathcal{S}$, the goal set is $G \subseteq \mathcal{S}$, and $\varphi$ is the task policy. The environment follows these transitions along a feasible history $h = e_1 \cdots e_t \in \mathcal{F}$ and reaches state $s(h)$. Here $\Sigma^*$ contains all finite event sequences, and $\mathcal{F} \subseteq \Sigma^*$ contains the histories feasible for this task. Only accepted tool effects enter this history. The reference monitor in Section 3.3 is $\mathcal{A}_\varphi = (Q, \Sigma, \delta, q_0, g)$, with state set $Q$, initial state $q_0$, transition $\delta : Q \times \Sigma \to Q$, and a decision function $g$ returning Permit or Deny for a state and candidate action. Its extended transition is defined recursively by $\delta^*(q, \epsilon) = q$ and $\delta^*(q, he) = \delta(\delta^*(q, h), e)$. Here $\epsilon$ denotes the empty history. The state after history $h$ is $q(h) = \delta^*(q_0, h)$. For a candidate action $u$, the event $\eta(u)$ records its execution with the specified operation, resource, caller, revision, and arguments. After a policy-valid history,

$$g(q(h), u) = \textsc{Permit} \quad \Longleftrightarrow \quad h\eta(u) \in L_\varphi. \tag{10}$$

Equation 10 connects the next action's decision to validity of the extended history. The adversary succeeds if the resource actually executes $u$ after $h \in \mathcal{F} \cap L_\varphi$ with $g(q(h), u) = \textsc{Deny}$, so $h\eta(u) \notin L_\varphi$. Enforcement must preserve $h \in L_\varphi$ after every commit. For events $a, b \in \Sigma$ whose parameters match as required by the policy, the ordering requirement is

$$\mathrm{Before}(a, b; h) = 1 \iff \forall j \in \{1, \ldots, t\}, \quad e_j = b \Rightarrow \exists k \in \{1, \ldots, j-1\} : e_k = a.$$

An approval for another revision cannot satisfy the deployment prerequisite. If $b$ never occurs, the condition is satisfied even if $a$ does not occur. It specifies the order of events, not whether the task finishes. These are finite-trace safety properties (De Giacomo & Vardi, 2013).

For a finite-state policy, the equivalence over histories $h, h' \in \Sigma^*$ is

$$h \sim_\varphi h' \quad \Longleftrightarrow \quad \forall w \in \Sigma^*, \quad hw \in L_\varphi \iff h'w \in L_\varphi. \tag{11}$$

These classes define the minimal deterministic monitor for complete histories (Hopcroft et al., 2006). They preserve verdicts after every continuation $w$. DSES groups states by their current answers over $\mathcal{C}_I$; the reference monitor also distinguishes how future events change those answers.

To judge whether a run followed its policy, an evaluator must remember earlier violations. The reference monitor uses prerequisite conditions to decide whether the next action is allowed after a policy-valid history (Equation 10). In Online and DSES modes, the PEP blocks the first forbidden action. In Observe mode, the resource executes requests that pass ordinary authorization even when the task policy forbids them. The trajectory scorer checks the order of executed actions. It marks a run invalid when the agent deploys before approval, even if the agent obtains approval later.

**Proposition 3** (Final-state indistinguishability)**.** *If feasible histories $h^+, h^-$ satisfy $s(h^+) = s(h^-)$ but $A(h^+) \neq A(h^-)$, no evaluator receiving only the final environment state can return both policy verdicts correctly.*

*Proof of Proposition 3.* The evaluator receives the same argument on the two histories. Put $s^* = s(h^+) = s(h^-)$ and relabel the histories so that $A(h^+) = 1$ and $A(h^-) = 0$. Exactness of a final-state evaluator $f$ would require both $f(s^*) = 1$ and $f(s^*) = 0$, a contradiction. Randomization cannot make both answers correct with probability one either. If $p$ is the probability of returning 1 on $s^*$, the two error probabilities are $1 - p$ and $p$. Their sum is 1, so at least one is positive. □

## A.2 Handoff sufficiency

Throughout this subsection, $I = (i, o, \mathcal{C}_I)$ is fixed. Both $Q_I$ and $\mathcal{C}_I$ are nonempty and finite. A profile $R_I(q)$ is the function assigning a policy answer to each $u \in \mathcal{C}_I$. Two profiles differ if they

give opposite answers for at least one candidate action. An exact decoder must recover the correct answer for every candidate action.

*Proof of Theorem 1. Constructing a conflicting pair.* Since $N_I(o) > 1$, there are $q^+, q^- \in Q_I$ with different profiles. By the definition of $Q_I$, policy-valid histories $h^+, h^-$ reach these states and satisfy $O_i^{\text{pre}}(h^+) = O_i^{\text{pre}}(h^-) = o$. Different profiles disagree on some $u \in \mathcal{C}_I$. Relabel the histories so that $g(q^+, u) = \textsc{Permit}$ and $g(q^-, u) = \textsc{Deny}$. Since $u$ passes ordinary authorization, $\lambda_i(o, u) = 1$. The histories and action therefore satisfy Equation 2.

*Binary decisions.* A decision rule sees $(o, u)$ in either history. Let $p$ be its probability of returning Permit, with its internal randomness independent of the hidden choice of history. On a uniform choice between the two histories,

$$\Pr(\text{error}) = \tfrac{1}{2}\Pr(\textsc{Deny} \mid h^+) + \tfrac{1}{2}\Pr(\textsc{Permit} \mid h^-) = \tfrac{1}{2}(1-p) + \tfrac{1}{2}p = \tfrac{1}{2}.$$

This includes deterministic rules ($p \in \{0, 1\}$). Exactness throughout the domain would require zero error on both members of this pair, which is impossible.

*Decisions with abstention.* For a three-valued rule, let $\alpha_P, \alpha_D, \alpha_\perp$ be the probabilities of Permit, Deny, and Undetermined on the shared input. Zero probability of a wrong definite answer on $h^-$ requires $\alpha_P = 0$, and the same requirement on $h^+$ gives $\alpha_D = 0$. The three probabilities sum to one, so $\alpha_\perp = 1$. The conclusion applies to this shared input and conflicting action. □

*Proof of Theorem 2.* Let $m : Q_I \to \mathcal{M}_I$ be an encoder, where $\mathcal{M}_I = m(Q_I)$ is its set of used message values. A decoder maps $\mathcal{M}_I \times \mathcal{C}_I$ to Permit or Deny.

*Necessity.* If an exact decoder $D$ exists, then for any $q, q'$ with $m(q) = m(q')$ and any $u \in \mathcal{C}_I$,

$$g(q, u) = D(m(q), u) = D(m(q'), u) = g(q', u).$$

Equality holds for every candidate action, hence $R_I(q) = R_I(q')$. This proves necessity.

*Sufficiency.* Assume message equality implies profile equality. For each reachable message $\mu \in m(Q_I)$ choose any state $q_\mu$ with $m(q_\mu) = \mu$, and define $D(\mu, u) = g(q_\mu, u)$. Every other state mapped to $\mu$ has the same profile, so the definition does not depend on the representative. In particular, $D(m(q), u) = g(q, u)$ for all $(q, u) \in Q_I \times \mathcal{C}_I$.

*Coarsest representation.* Exactness requires distinct messages for distinct profiles. We assign one identifier to each distinct profile, so two states receive the same identifier exactly when their profiles agree. Algorithm 1 implements precisely this map: it computes all profile entries and stores each distinct row once. It uses $N_I(o)$ identifiers, whereas every exact encoder needs at least $N_I(o)$. Any exact encoder may split a profile across several messages, but cannot merge two different profiles. The DSES construction makes no such extra split, giving the coarsest exact partition, unique up to the labels assigned to its classes.

*Fixed-length size.* A binary message of $b$ bits has at most $2^b$ values, so

$$2^b \geq N_I(o) \quad \Longrightarrow \quad b \geq \lceil \log_2 N_I(o) \rceil.$$

Numbering profiles from 0 to $N_I(o) - 1$ and writing each number in $\lceil \log_2 N_I(o) \rceil$ bits achieves the bound in Equation 8. If $N_I(o) = 1$, the single identifier needs zero transmitted bits because the installed decoder already knows every answer. □

For a message computed from a history rather than only its monitor state, replace $m(q)$ by $m(h)$ and compare histories in the same declared observation class. An exact decoder exists precisely when $m(h) = m(h') \Rightarrow R_I(q(h)) = R_I(q(h'))$ on that domain. For necessity, equal messages give the decoder the same input for every action, so all corresponding answers must agree. For sufficiency, choose one history for each used message and decode its profile. Equality of profiles within each message makes that decoder well defined. Histories with the same profile may use different messages, but histories with different profiles must use different messages. Enumerating one representative history per monitor state is sufficient for a state-only encoder. For a handoff that depends on event order, we evaluate the histories in the declared domain rather than only one representative per monitor state.

**When the handoff needs more information.** For fixed $I$, a delegation fork exists exactly when $N_I(o) > 1$: Theorem 1 constructs a fork from two profiles, and a fork's opposite answers imply two distinct profiles. By Theorem 2, a constant message is exact if and only if $N_I(o) = 1$, requiring zero additional bits.

**Issuing DSES from incomplete records.** For records supplied by other agents, a signed *coverage checkpoint* identifies the complete set of event records covered at a stated state version. Let $\mathcal{E}$ contain this checkpoint and the verified records disclosed to the monitor. They restrict the possible states to $\mathcal{B}_I(\mathcal{E}) \subseteq Q_I$. An omitted record alone does not establish that its event did not occur. The monitor can establish that an event did not occur only if its authoritative source provides complete coverage. It issues DSES only when the compatible states share one code:

$$\widetilde{m}_I(\mathcal{B}) = \begin{cases} \mu, & \{m_I(q) : q \in \mathcal{B}\} = \{\mu\}, \\ \bot, & \text{otherwise.} \end{cases} \tag{12}$$

The monitor returns Undetermined and issues no signed DSES message if no states are compatible or if compatible states disagree on an action's policy decision. If the compatible set contains the true state, the following proposition guarantees that the issued code answers every candidate action correctly without identifying a unique state.

**Proposition 4** (Exact answers from incomplete records)**.** *If the true current state $q^*$ lies in nonempty $\mathcal{B}_I(\mathcal{E})$ and $\widetilde{m}_I(\mathcal{B}_I(\mathcal{E})) = \mu \neq \bot$, then $D_I(\mu, u) = g(q^*, u)$ for every $u \in \mathcal{C}_I$.*

*Proof of Proposition 4.* Write $\mathcal{B} = \mathcal{B}_I(\mathcal{E})$. By Equation 12, a definite result $\mu$ means $\{m_I(q) : q \in \mathcal{B}\} = \{\mu\}$. Since $q^* \in \mathcal{B}$, $m_I(q^*) = \mu$. Exactness from Theorem 2 now gives, for an arbitrary $u \in \mathcal{C}_I$,

$$D_I(\mu, u) = D_I(m_I(q^*), u) = g(q^*, u).$$

Because $u$ was arbitrary, one issued code gives exact answers for the entire action interface. The compatible set can contain several states, including the true state, because they all give the same policy decisions. □

*Proof of Equation 9.* Consider a message $\nu$ with nonempty compatible set $Q_I(\nu)$. *Soundness.* Every action in $\mathcal{P}_I(\nu)$ receives Permit in every compatible state, so allowing all of $\mathcal{P}_I(\nu)$ is sound.

*Maximality.* Let $\mathcal{P}'$ be any action set that can be allowed without a false Permit in any compatible state. For every $u \in \mathcal{P}'$ and every $q \in Q_I(\nu)$, soundness requires $g(q, u) = \textsc{Permit}$. Hence $u$ belongs to the intersection defining $\mathcal{P}_I(\nu)$, and $\mathcal{P}' \subseteq \mathcal{P}_I(\nu)$. Equivalently, any $u \in \mathcal{C}_I \setminus \mathcal{P}_I(\nu)$ has a compatible state that returns Deny. Allowing $u$ would be wrong in that state. If the intersection is empty, each candidate action is forbidden in at least one compatible state, so none can safely be permitted. □

**Corollary 5** (Policy validity after every commit)**.** *Assume $L_\varphi$ is prefix closed and every committed effect passes the PEP. Each accepted request uses exact DSES for the correct current monitor state, and verification and execution occur in one serialized transaction. For any sequence of $n \geq 0$ committed actions $u_1, \ldots, u_n$, write $h^{(0)} = \epsilon$ and $h^{(t)} = \eta(u_1) \cdots \eta(u_t)$ for $1 \leq t \leq n$. Then*

$$\epsilon \in L_\varphi \quad \Longrightarrow \quad \forall t \in \{0, \ldots, n\}, \qquad h^{(t)} \in L_\varphi.$$

*Proof of Corollary 5.* Assume $\epsilon \in L_\varphi$ and consider any committed sequence. We induct on $t$. Rejected or failed calls add no event.

*Base case.* $h^{(0)} \in L_\varphi$ by assumption.

*Inductive step.* Assume $h^{(t)} \in L_\varphi$ for $t < n$. For the next committed action $u_{t+1}$, let $r$ be its request, env the envelope carrying code $\mu$, and $I$ the interface used at that commit. The current-state assumption gives $q(h^{(t)}) \in Q_I$ and $\mu = m_I(q(h^{(t)}))$. Here $c_{\text{live}}$ is the resource's live verification

context. Since the PEP verifies the request before executing it, Equation 23 and the action check in Equation 21 give

$$\Gamma_I(r, \mathsf{env}; c_{\mathrm{live}}) = 1 \quad \Longrightarrow \quad u_r = u_{t+1} \in \mathcal{C}_I, \quad D_I(\mu, u_{t+1}) = \textsc{Permit}.$$

Exactness in Equation 4 therefore yields

$$g(q(h^{(t)}), u_{t+1}) = D_I(\mu, u_{t+1}) = \textsc{Permit}.$$

Equation 10 at the valid prefix gives $h^{(t)}\eta(u_{t+1}) \in L_\varphi$. Serialization makes this history exactly $h^{(t+1)}$: no intervening effect changes the checked state. The monitor updates to $q(h^{(t+1)})$ before the next request. Induction proves validity for all $0 \le t \le n$. □

### A.3 Distributed observations

Consider a configuration $z$ with agents $\mathcal{I}_z$ and observation maps $O_i^{z,\mathrm{pre}}$. Each map gives the history information available to agent $i$ before its decision. Write $K_\varphi = \mathcal{F} \cap L_\varphi$ for the policy-valid histories in the declared domain. The following set contains histories whose observation matches some policy-valid history for each agent:

$$\mathrm{Loc}_z(K_\varphi) = \mathcal{F} \cap \bigcap_{i \in \mathcal{I}_z} (O_i^{z,\mathrm{pre}})^{-1}(O_i^{z,\mathrm{pre}}(K_\varphi)). \tag{13}$$

Each agent's local observation can match a different valid history. Let $K_i \subseteq O_i^{z,\mathrm{pre}}(\mathcal{F})$ be the observations accepted by agent $i$'s local test. For a set of observations, its inverse image contains the histories producing one of those observations. The agents accept a history exactly when every test accepts. Exact local tests exist if and only if

$$K_\varphi = \mathrm{Loc}_z(K_\varphi). \tag{14}$$

*Proof of Equation 14. Every valid history is locally compatible.* For every $h \in K_\varphi$ and agent $i$, its observation belongs to $O_i^{z,\mathrm{pre}}(K_\varphi)$. Thus $K_\varphi \subseteq \mathrm{Loc}_z(K_\varphi)$ always holds.

*Necessity.* Suppose exact local tests exist, so $K_\varphi = \mathcal{F} \cap \bigcap_i (O_i^{z,\mathrm{pre}})^{-1}(K_i)$. Every valid history passes each test, so $O_i^{z,\mathrm{pre}}(K_\varphi) \subseteq K_i$. Taking inverse images and intersections gives

$$\mathrm{Loc}_z(K_\varphi) \subseteq \mathcal{F} \cap \bigcap_i (O_i^{z,\mathrm{pre}})^{-1}(K_i) = K_\varphi.$$

Together the inclusions give $K_\varphi = \mathrm{Loc}_z(K_\varphi)$. Otherwise, a history in $\mathrm{Loc}_z(K_\varphi) \setminus K_\varphi$ would pass every local test despite being invalid.

*Sufficiency.* Conversely, if $K_\varphi = \mathrm{Loc}_z(K_\varphi)$, choose each local acceptance set to be $K_i = O_i^{z,\mathrm{pre}}(K_\varphi)$. Their conjunction accepts exactly the histories in $\mathrm{Loc}_z(K_\varphi) = K_\varphi$, which proves sufficiency. The tests need not identify which valid history produced a local observation. They only need the membership test in the corresponding set $K_i$. □

When each observation keeps only the events visible to that agent, this is decomposability relative to $\mathcal{F}$. When $\mathcal{F} = \Sigma^*$, it is called separability (Komenda & Masopust, 2017). The proof also holds for general observation maps. It checks whether local tests can recognize all and only policy-valid histories. Online decentralized enforcement further depends on which agent can block each event and which information arrives before that event (Rudie & Wonham, 1992; Komenda & Masopust, 2017). For action choice, the designated recipient needs enough information to distinguish states that give different decisions for any candidate action. The PEP enforces the resulting decisions under the conditions of Corollary 5.

### A.4 Decision error

Following Kato & Kato (2026), we distinguish information retained in a handoff from the recipient's use of it. For executable task policies, PolicyFork makes this distinction testable through same-input counterexamples and minimum field repairs. Handover's Definition 3.2 and Proposition 3.1

concern predictive sufficiency, and its Proposition 6.1 compares information-equivalent records. Our prediction target is the policy's Permit or Deny for a fully specified action.

For one action interface $I$, let $q \in Q_I$ be a random state with distribution $\pi$. In this analysis, the random action $\mathsf{A}$ is independent of $q$, with positive probability on every action in $\mathcal{C}_I$. A deterministic encoder produces message $\mathsf{M} = m(q)$ before this choice. Write

$$\mathsf{Y} = \mathbf{1}\{g(q, \mathsf{A}) = \textsc{Permit}\}, \qquad p(\mu, u) = \Pr(\mathsf{Y} = 1 \mid \mathsf{M} = \mu, \mathsf{A} = u).$$

A recipient's binary decision rule $d$ outputs 1 for Permit and 0 for Deny, with error probability $\mathcal{R}(d; m) = \Pr[d(\mathsf{M}, \mathsf{A}) \neq \mathsf{Y}]$. The rule uses only the message, action, and independent randomness. Write $\mathcal{R}^*(m) = \inf_d \mathcal{R}(d; m)$ for the minimum error probability among these rules. All information quantities below use base-two logarithms. For random variables $X, Y, Z$, we write $\mathrm{H}(X \mid Z)$ for conditional entropy and $I(X; Y \mid Z)$ for conditional mutual information.

**Proposition 6** (Information loss and decision error)**.** *The smallest possible error using a fixed encoder is*

$$\mathcal{R}^*(m) = \mathbb{E}\big[\min\{p(\mathsf{M}, \mathsf{A}), 1 - p(\mathsf{M}, \mathsf{A})\}\big]. \tag{15}$$

*Every recipient's error can be written as*

$$\mathcal{R}(d; m) = \underbrace{\mathcal{R}^*(m)}_{\text{error from missing information}} + \underbrace{\big(\mathcal{R}(d; m) - \mathcal{R}^*(m)\big)}_{\text{additional recipient error } \geq 0}. \tag{16}$$

*For decoders that predict the probability of Permit, the minimum expected log loss is*

$$\mathrm{H}(\mathsf{Y} \mid \mathsf{M}, \mathsf{A}) = I(\mathsf{Y}; q \mid \mathsf{M}, \mathsf{A}). \tag{17}$$

*If $\pi$ gives positive probability to every state, the minimum binary error and minimum log loss are zero exactly when no message combines states with different decision profiles.*

*Proof of Proposition 6. Minimum binary error.* Consider $(\mu, u)$ with positive probability and put $p = p(\mu, u)$. If a binary recipient chooses Permit with probability $\alpha$, its conditional error minus $\min\{p, 1 - p\}$ is

$$\alpha(1 - p) + (1 - \alpha)p - \min\{p, 1 - p\} = \begin{cases} \alpha(1 - 2p), & p \leq \frac{1}{2}, \\ (1 - \alpha)(2p - 1), & p \geq \frac{1}{2}. \end{cases}$$

Both expressions are nonnegative for $0 \leq \alpha \leq 1$. Choosing Permit for $p \geq 1/2$ and Deny otherwise achieves the minimum at every input. Taking expectations proves Equation 15. The risk of any other recipient is at least this minimum. Adding and subtracting $\mathcal{R}^*(m)$ gives Equation 16. Its second term is nonnegative because $\mathcal{R}^*(m)$ is the minimum risk. Inputs of probability zero contribute nothing to either risk.

*Minimum log loss.* Let $a \in [0, 1]$ be the probability the decoder assigns to Permit at this input. With base-two logarithms and the usual limiting conventions at 0 and 1, its conditional expected loss is

$$-p \log_2 a - (1 - p) \log_2(1 - a) = H_2(p) + D_{\mathrm{KL}}\big(\mathrm{Bern}(p) \,\|\, \mathrm{Bern}(a)\big) \;\geq\; H_2(p).$$

Here $H_2(p) = -p \log_2 p - (1-p) \log_2(1-p)$ is binary entropy, $\mathrm{Bern}(p)$ is the Bernoulli distribution with success probability $p$, and $D_{\mathrm{KL}}$ is Kullback–Leibler divergence. For $0 < p < 1$ and $0 < a < 1$, differentiating the expected loss gives $(a - p)/(a(1 - a) \ln 2)$. The loss decreases up to $a = p$ and increases afterward. For $p = 0$ it is $-\log_2(1 - a)$, minimized at $a = 0$; for $p = 1$ it is $-\log_2 a$, minimized at $a = 1$. This also verifies the inequality and equality cases directly. The choice $a = p$ achieves the bound at each input with positive probability, including $p = 0$ and $p = 1$. Averaging over messages and actions gives $\mathbb{E}[H_2(p(\mathsf{M}, \mathsf{A}))] = \mathrm{H}(\mathsf{Y} \mid \mathsf{M}, \mathsf{A})$. Because $\mathsf{Y}$ is determined by $(q, \mathsf{A})$,

$$I(\mathsf{Y}; q \mid \mathsf{M}, \mathsf{A}) = \mathrm{H}(\mathsf{Y} \mid \mathsf{M}, \mathsf{A}) - \mathrm{H}(\mathsf{Y} \mid q, \mathsf{M}, \mathsf{A}) = \mathrm{H}(\mathsf{Y} \mid \mathsf{M}, \mathsf{A}),$$

establishing Equation 17.

*When zero risk means a sufficient handoff.* Suppose $\pi(q) > 0$ for every state. If a message merges distinct profiles, choose states $q_+, q_-$ with a common message $\mu$ and an action $u$ for which the answers are Permit and Deny, respectively. Both events have positive probability because

$$\Pr(q = q_+, \mathsf{A} = u) = \pi(q_+) \Pr(\mathsf{A} = u) > 0, \qquad \Pr(q = q_-, \mathsf{A} = u) > 0.$$

Conditioning on $(\mathsf{M}, \mathsf{A}) = (\mu, u)$ thus retains positive probability for both answers. Hence $0 < p(\mu, u) < 1$, and both $\min\{p, 1-p\}$ and $H_2(p)$ are positive there. Conversely, if message equality implies profile equality, every input with positive probability has $p(\mu, u) \in \{0, 1\}$, making both minima zero. □

We use Equation 15 to compute the lowest possible expected error over the enumerated states for a given handoff and evaluation distribution. The API study scores final Permit/Deny answers and counts absent outputs separately because a missing answer is neither decision. The exact construction in Theorem 2 preserves the policy answer for every state and candidate action, for any state distribution and action-selection rule.

**Information-equivalent encodings.** Let $\mathsf{R} = R_I(q)$ be the complete profile. For any exact encoding, $\mathsf{R}$ is a deterministic function of $\mathsf{M}$: evaluate the exact decoder once for each candidate action and collect its answers. Since $\mathsf{R}$ is also a function of $q$, conditional entropy $\mathrm{H}(\mathsf{R} \mid q)$ is zero. Data processing then gives

$$I(q; \mathsf{M}) \;\geq\; I(q; \mathsf{R}) = \mathrm{H}(\mathsf{R}). \tag{18}$$

For DSES, the profile determines the code and the code determines the profile. Applying data processing in the reverse direction gives $I(q; \mathsf{M}) \leq I(q; \mathsf{R})$, so equality holds. A complete verdict bitset achieves equality as well: in the release interface its three positions encode four reachable profiles. If each of two encodings can be computed from the other, a recipient can convert the message and use the other encoding's decoder. Any error rate achieved with one encoding can therefore be achieved with the other, so their minimum risks agree. A fixed model can still respond differently to the two encodings (Kato & Kato, 2026). The encodings retain the same decision information, while a model's error can depend on how that information is presented.

**State updates.** DSES preserves current action answers, while the reference monitor retains the distinctions needed after later events. Consider two release states with no successful code check for the requested revision and no approval recorded for that revision. Only one contains independent approval. The policy denies merge, staging, and production in both states. After the same successful code check, it permits only merge in the state without approval, but also permits staging in the state with approval. The two states thus need different updated profiles. After each commit, the reference monitor updates its state and issues fresh DSES. The current-action equivalence in Equation 3 need not be preserved by state transitions. The history equivalence in Equation 11 also accounts for future continuations.

# B Handoff Evaluation

This appendix evaluates what different handoffs allow a recipient to decide and what they cost. We first prove and test minimum field repairs, then measure DSES construction and communication costs. The final sections explain the curves for shared prerequisites, limited message size, and incomplete records. Each comparison uses the same policy decisions for all methods.

## B.1 Minimum field repairs

We evaluate whether field repair provides enough information for every candidate action in the release example (Table 3). If the handoff contains only the code-check result, the search adds independent-approval status and whether approval was recorded for the requested revision. It leaves sufficient messages unchanged and returns an unresolved pair when the available fields cannot repair the handoff.

**Repair correctness.** Section 4.2 defines the conflicting pairs $\mathcal{W}_i$ and minimum field set $J^*$ (Equations 6–7). For any field set $J$, write the repaired input as $O_i^{\mathrm{pre},J}(h) = (O_i^{\mathrm{pre}}(h), (f_j(h))_{j \in J})$, with field names in a fixed order. It retains the complete original input, including the existing handoff. We group the supplied valid prefixes by their original input and compare policy decisions over the same candidate set $\mathcal{C}_I$ within each group. The history-based exact handoff criterion applies to each group. We select a single field set $J$ that repairs all groups. Each $f_j$ is a function assigning a typed field value to every $h \in \mathcal{H}$. A missing value is not a Boolean false value.

**Lemma 7** (Minimum field repairs). *For the supplied domain $\mathcal{H}$ and fields $f_1, \ldots, f_p$, the repaired input $O_i^{\mathrm{pre},J}$ determines every candidate action's answer if and only if $J$ intersects $\Delta(h, h')$ for every $\{h, h'\} \in \mathcal{W}_i$. A repair exists exactly when no conflicting pair has an empty separating-field set. Searching subsets in increasing order of field count returns a repair with the fewest added fields whenever one exists.*

*Proof of Lemma 7. Necessity.* Suppose $J$ contains no index from $\Delta(h, h')$ for some original conflicting pair. Then $O_i^{\mathrm{pre}}(h) = O_i^{\mathrm{pre}}(h')$ and every added field agrees, giving $O_i^{\mathrm{pre},J}(h) = O_i^{\mathrm{pre},J}(h')$. The two profiles still disagree on an action, so the repaired input is insufficient. This proves necessity.

*Sufficiency.* Suppose $J$ intersects every conflict's separating-field set. Take any $h, h' \in \mathcal{H}$ with $O_i^{\mathrm{pre},J}(h) = O_i^{\mathrm{pre},J}(h')$. Equality of these tuples implies both $O_i^{\mathrm{pre}}(h) = O_i^{\mathrm{pre}}(h')$ and $f_j(h) = f_j(h')$ for every $j \in J$. If their profiles differed, the pair of histories would belong to $\mathcal{W}_i$, and some $j \in J \cap \Delta(h, h')$ would satisfy $f_j(h) \neq f_j(h')$, contradicting equality of the repaired inputs. Thus their profiles agree. The history-based form of Theorem 2 now gives an exact decoder: for each repaired input, choose one supplied history producing it and return that history's profile. Every other history with the same input has the same profile, so the decoder gives the correct answer for every candidate action. This construction also allows histories reaching the same monitor state to carry different messages. The repair retains the original input, so histories with different original inputs remain distinguishable.

*Existence and minimum field count.* If $\mathcal{W}_i = \varnothing$, $J = \varnothing$ is feasible and minimal. If a conflict has $\Delta(h, h') = \varnothing$, no field set can separate it. Otherwise the full available field set intersects every conflict, so at least one repair exists. Searching subsets in increasing order of field count returns a minimum field repair: every smaller set has already failed the necessary and sufficient condition. Ties may be broken by field name without changing the number of added fields. □

**Implementation and rechecking.** Our implementation assembles the retained input and selected fields, then runs the same checker again. It compares the resulting recipient inputs and verifies that identical inputs require identical policy decisions. Thus the check applies to the messages that will actually be sent. The implementation searches inputs with up to sixteen fields and 512 runs. Larger requests are rejected before search. Each field must have a value in every supplied run. Missing values cause an input error.

Table 5: **Repair only what the existing handoff omits.** The search adds the fewest declared fields and rechecks every action. GitHub uses twelve controlled inputs to the original job condition, and release uses the existing $32 \times 3$ state–action table. All repaired messages have zero conflicting pairs. Already sufficient inputs remain unchanged. Revision approval denotes approval recorded for the requested revision.

| Handoff | Fields added | Count | Permitted combinations admitted |
|---|---|---|---|
| GitHub: original fields | None | 0 | $1/1 \to 1/1$ |
| GitHub: status only | Environment | 1 | $0/1 \to 1/1$ |
| GitHub: environment only | Status | 1 | $0/1 \to 1/1$ |
| Release: code check only | Independent approval + revision approval | 2 | $16/28 \to 28/28$ |
| Release: code check + independent approval | Revision approval | 1 | $24/28 \to 28/28$ |
| Release: relevant facts | None | 0 | $28/28 \to 28/28$ |
| Release: verdict bitset | None | 0 | $28/28 \to 28/28$ |
| Release: DSES | None | 0 | $28/28 \to 28/28$ |

Table 5 identifies the selected fields for four nonempty repairs and the sufficient messages preserved unchanged. Two additional availability tests withhold the required environment field from the GitHub status-only case and the revision-approval field from the release code-check-and-independent-approval case. In both tests, the checker returns two executions that no available field can distinguish. The orchestrator must obtain more information to resolve their conflicting decisions.

The recipient's input $o$ includes visible tool availability when it reveals policy state. Run identifiers locate counterexamples but are not sent to the recipient.

**Information already available to the recipient.** The release interface has four decision profiles and needs two semantic bits. If the recipient knows that the code check passed for the requested revision, it can rule out the all-Deny profile. Three profiles remain, so the code still needs two bits. If the recipient knows that all three prerequisites hold, only one profile remains and no additional semantic bits are needed. For a fixed candidate action set, reliable observations can only reduce the compatible profiles. The count $N_I(o)$ therefore measures the distinctions still needed after the recipient has received its original input.

The handoff reaches the recipient before it selects an action. For one fixed action, a trusted Permit/-Deny answer can suffice. A reusable handoff determines the answer for every candidate action at the current state. Each committed operation requires an updated message.

### B.2 DSES COSTS

Algorithm 1 makes $|Q_I|\,|\mathcal{C}_I|$ calls to $g$. If one call costs at most $T_g$, profile evaluation costs $O(|Q_I|\,|\mathcal{C}_I|T_g)$. To construct $Q_I$, we also explore the combined states of the monitor and finite environment model. Deterministic ordering costs $O(|\mathcal{C}_I|\log|\mathcal{C}_I|)$ action comparisons and $O(N_I(o)\log N_I(o))$ profile comparisons, each reading at most $|\mathcal{C}_I|$ entries. Hashing profiles reads $O(|Q_I|\,|\mathcal{C}_I|)$ entries in expectation. The decoder stores $N_I(o)|\mathcal{C}_I|$ decision bits. The state-to-code map and state search require additional storage.

**Measured costs.** We measure Algorithm 1 on 20 configurations: $k \in \{1, 2, 4, 8, 12\}$ nested or independent action prerequisites, with zero or four irrelevant Boolean variables. We run each configuration three times in fresh Python processes on an Apple M3 Pro. Timings cover monitor construction, compilation, full-state validation, and output generation. We report medians and retain the measurement ranges; peak RSS covers the complete process.

Table 6: **Construction costs on the complete configuration grid.** Each entry is the median of three fresh processes. Total includes building the monitor, compiling, checking every state–action answer, and writing the resulting tables. Compile reports DSES construction time alone. Unused counts policy-irrelevant Boolean facts. RSS is peak resident memory.

| Structure | $k$ | Unused | States | Profiles | Total (ms) | Compile (ms) | RSS (MiB) |
|---|---|---|---|---|---|---|---|
| Nested | 1 | 0 | 2 | 2 | 0.14 | 0.08 | 31.1 |
| Independent | 1 | 0 | 2 | 2 | 0.14 | 0.07 | 30.9 |
| Nested | 1 | 4 | 32 | 2 | 0.25 | 0.10 | 31.3 |
| Independent | 1 | 4 | 32 | 2 | 0.27 | 0.12 | 31.1 |
| Nested | 2 | 0 | 4 | 3 | 0.19 | 0.10 | 31.6 |
| Independent | 2 | 0 | 4 | 4 | 0.17 | 0.09 | 31.2 |
| Nested | 2 | 4 | 64 | 3 | 0.46 | 0.19 | 31.4 |
| Independent | 2 | 4 | 64 | 4 | 0.45 | 0.19 | 31.1 |
| Nested | 4 | 0 | 16 | 5 | 0.26 | 0.11 | 31.1 |
| Independent | 4 | 0 | 16 | 16 | 0.26 | 0.11 | 31.5 |
| Nested | 4 | 4 | 256 | 5 | 1.57 | 0.44 | 31.2 |
| Independent | 4 | 4 | 256 | 16 | 1.63 | 0.47 | 30.9 |
| Nested | 8 | 0 | 256 | 9 | 2.42 | 0.60 | 31.3 |
| Independent | 8 | 0 | 256 | 256 | 2.58 | 0.75 | 31.1 |
| Nested | 8 | 4 | 4,096 | 9 | 36.93 | 7.67 | 35.5 |
| Independent | 8 | 4 | 4,096 | 256 | 35.30 | 8.03 | 35.3 |
| Nested | 12 | 0 | 4,096 | 13 | 47.35 | 9.51 | 35.9 |
| Independent | 12 | 0 | 4,096 | 4,096 | 53.39 | 14.01 | 37.8 |
| Nested | 12 | 4 | 65,536 | 13 | 823.48 | 189.68 | 120.1 |
| Independent | 12 | 4 | 65,536 | 4,096 | 801.55 | 195.81 | 140.8 |

### B.3 SHARED PREREQUISITES

Figure 4 varies $k$ Boolean prerequisites over all $2^k$ states, for integers $k \geq 1$ and $0 \leq r \leq k$. In the nested policy, action $j$ requires the first $j$ prerequisites; in the independent policy, it requires

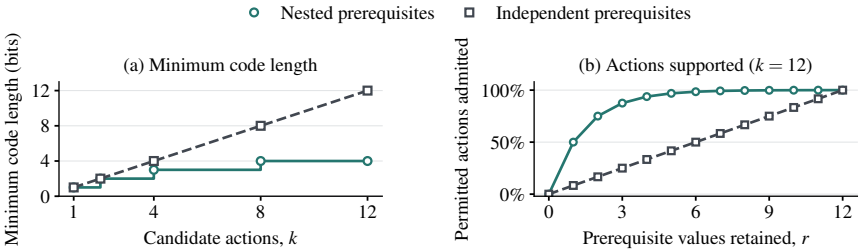


Figure 4: **Shared prerequisites reduce the bits needed for decisions.** Lines give theory and markers give enumeration. Panel (b) shows the fraction of permitted state–action pairs accepted when $r$ of $k = 12$ prerequisite values are disclosed, across 4,096 states with zero false Permits.

only prerequisite $j$. The recipient receives the first $r$ truth values in a fixed order, including verified false values. The receiver permits every action in the largest safe set in Equation 9, proved in Appendix A.2. Reducing the compatible set can only enlarge the set of actions that all compatible states permit.

Nested action $j$ is permitted when its first $j$ prerequisites hold, regardless of the remaining $k - j$ prerequisite values. It is permitted in $2^{k-j}$ states, so there are $\sum_{j=1}^{k} 2^{k-j} = 2^k - 1$ permitted state–action pairs. With the first $r$ prerequisite values known, the receiver can determine that action $j \leq r$ is permitted exactly when those $j$ prerequisites hold. If $j > r$, a compatible state can set an unobserved prerequisite to false, so the receiver cannot safely permit that action. Summing the first $r$ counts yields $\sum_{j=1}^{r} 2^{k-j} = 2^k - 2^{k-r}$ accepted pairs. Each independent action is permitted in $2^{k-1}$ states, giving $k2^{k-1}$ permitted pairs and $r2^{k-1}$ accepted pairs. Dividing accepted pairs by all permitted pairs gives

$$\frac{\text{permitted pairs accepted}}{\text{all permitted pairs}} = \begin{cases} \dfrac{1 - 2^{-r}}{1 - 2^{-k}}, & \text{nested,} \\ \dfrac{r}{k}, & \text{independent.} \end{cases} \tag{19}$$

At $r = 0$, the receiver cannot safely permit any action from the empty input. At $r = k$, the receiver accepts every permitted pair, as both formulas give one. Independent enumeration verifies every point at $k = 12$. For nested actions, a profile is determined by the length of the initial sequence of satisfied prerequisites, which ranges from zero to $k$. Every length occurs, giving $k + 1$ profiles. For independent actions, each Boolean prerequisite changes exactly its corresponding answer, giving $2^k$ profiles. By Equation 8, the exact code lengths are $b^*_{\text{nested}} = \lceil \log_2(k + 1) \rceil$ and $b^*_{\text{independent}} = k$, giving panel (a) of Figure 4. For $0 < r < k$, the retained fraction in Equation 19 is larger for nested prerequisites, as shown in panel (b). Adding four policy-irrelevant Boolean variables changes neither count. Across 20 configurations, the check covers 148,716 states and 1,743,316 state–action comparisons. These are exhaustive counts for the specified structures. Here the receiver obtains both true and false field values. Disclosing only records of events that occurred gives less information: the absence of a disclosed record does not establish that the event did not occur.

### B.4 Handoff information budget

Figure 3 uses five Boolean conditions: whether the code check passed for the requested revision, independent approval was obtained, approval for the requested revision was recorded, release notes were published, and the issue was closed. Only the first three affect merge, staging, and production. All 32 states and three actions receive equal weight. At each integer bit budget, we choose a subset of these fields and send their values in every state. We enumerate all 32 subsets. For each subset size, the black curve shows the lowest loss, the red curve shows the mean, and the gray upper edge shows the highest loss. Each panel uses its own loss.

The encoding reference enumerates all 15 partitions of the four profiles, and retains partitions with at most $2^b$ message values at budget $b$. Panel (a) minimizes conditional entropy over these encodings. Panel (b) uses the largest safe action set for each encoding and minimizes withheld permitted actions. The lemma below shows that an optimal encoder for each panel can assign one message to each

complete profile. Below two bits, some profiles must share a message, leaving some decisions unresolved. At two bits, DSES distinguishes all four profiles.

For the encoding comparison, consider a finite interface $I$. Let $q \sim \pi$ and $\mathsf{A} \sim \alpha$ be independent, with positive probability on every state and candidate action. Set $\mathsf{Y} = \mathbf{1}\{g(q, \mathsf{A}) = \textsc{Permit}\}$ and $\mathsf{M} = m(q)$. For each used message $\mu$, let $\mathcal{P}_m(\mu) = \bigcap_{q:m(q)=\mu}\{u \in \mathcal{C}_I : g(q,u) = \textsc{Permit}\}$ be its largest safe action set. The two plotted losses are

$$\ell_{\mathrm{ent}}(m) = \mathrm{H}(\mathsf{Y} \mid \mathsf{M}, \mathsf{A}), \qquad \ell_{\mathrm{with}}(m) = \frac{\Pr(\mathsf{Y} = 1,\ \mathsf{A} \notin \mathcal{P}_m(\mathsf{M}))}{\Pr(\mathsf{Y} = 1)}.$$

The second loss requires $\Pr(\mathsf{Y} = 1) > 0$.

**Lemma 8** (Encoding decision profiles)**.** *Let $\mathcal{R}_I = \{R_I(q) : q \in Q_I\}$ and $[L] = \{1, \dots, L\}$ for an integer message budget $L \geq 1$. For either loss above,*

$$\min_{m:Q_I \to [L]} \ell(m) = \min_{f:\mathcal{R}_I \to [L]} \ell(f \circ R_I).$$

*Thus an optimal encoder can map each complete profile to one message; different profiles may share that message. The two losses need not have the same optimizer.*

*Proof of Lemma 8. Expressing a state encoder through profiles.* Let $\mathsf{R} = R_I(q)$ and $\mathsf{M} = m(q)$ denote the profile and message. Although $m$ may distinguish states with the same profile, it defines the finite conditional distribution $W(\mu \mid \rho) = \Pr(\mathsf{M} = \mu \mid \mathsf{R} = \rho)$. For each profile $\rho$, independently choose a message with probabilities $W(\cdot \mid \rho)$. Index the resulting deterministic maps by $t$ and assign each map $f_t$ probability

$$\lambda_t = \prod_{\rho \in \mathcal{R}_I} W(f_t(\rho) \mid \rho), \qquad \sum_t \lambda_t = \prod_{\rho \in \mathcal{R}_I} \sum_{\mu=1}^{L} W(\mu \mid \rho) = 1.$$

Let $\mathsf{T}$ index this random map, sampled independently of the state and action, with $\Pr(\mathsf{T} = t) = \lambda_t$. Summing over the choices for every other profile gives

$$\sum_{t:f_t(\rho)=\mu} \lambda_t = W(\mu \mid \rho).$$

Define $\mathsf{M}' = f_{\mathsf{T}}(\mathsf{R})$. Independence of the action from the state and the sampled map gives

$$\begin{aligned}\Pr(\mathsf{R} = \rho, \mathsf{M}' = \mu, \mathsf{A} = u) &= \Pr(\mathsf{R} = \rho)\Pr(\mathsf{A} = u)W(\mu \mid \rho) \\ &= \Pr(\mathsf{R} = \rho, \mathsf{M} = \mu, \mathsf{A} = u).\end{aligned}$$

Thus replacing $\mathsf{M}$ by $\mathsf{M}'$ preserves the joint distribution relevant to both objectives. In particular, $\mathsf{Y} = \mathbf{1}\{\mathsf{R}(\mathsf{A}) = \textsc{Permit}\}$ is unchanged.

*Conditional entropy.* Conditioning on the chosen map can only reduce entropy, so

$$\begin{aligned}\mathrm{H}(\mathsf{Y} \mid \mathsf{M}, \mathsf{A}) &= \mathrm{H}(\mathsf{Y} \mid \mathsf{M}', \mathsf{A}) \\ &\geq \mathrm{H}(\mathsf{Y} \mid \mathsf{M}', \mathsf{A}, \mathsf{T}) \\ &= \sum_t \lambda_t \,\mathrm{H}(\mathsf{Y} \mid f_t(\mathsf{R}), \mathsf{A}).\end{aligned}$$

The last equality uses independence of $\mathsf{T}$ from $(\mathsf{R}, \mathsf{A})$. At least one map with positive weight has conditional entropy no larger than the weighted average, and hence no larger than that of the original state encoder.

*Withheld permitted actions.* Retain the original receiver's largest safe set $\mathcal{P}_m(\mu)$ for each message $\mu$. For any map $f_t$ with $\lambda_t > 0$, the equality $f_t(\rho) = \mu$ implies $W(\mu \mid \rho) > 0$. Every profile reaching $\mu$ under this map could also reach $\mu$ under the original encoder. The retained receiver remains sound because all its permitted actions are permitted in every one of these profiles.

By the equality of joint distributions, the probability that the retained receiver permits an action, averaged over $\mathsf{T}$, equals its original value. For at least one positive-weight map, the receiver therefore

permits an action with at least the original probability. Allowing every action in that map's largest safe set can only increase this probability. The probability that the policy permits an action is positive and unchanged, so the fraction of permitted actions withheld cannot increase. Under uniform state–action weights, these probabilities give the counts plotted in Figure 3.

*Message budget.* Every map uses only the original message values and groups profiles assigned the same message. Hence a profile encoder can replace each original encoder without increasing the selected loss or the number of message values. Profile encoders are also state encoders, so optimizing over the two classes gives the same minimum. Each objective may select a different partition. □

The gray upper edges differ because panel (a) measures uncertainty about both Permit and Deny answers, while panel (b) counts only permitted actions withheld. Both panels use the same field subsets. Figure 5 instead weights profiles equally. The weighting changes the cost of an omitted distinction, but both weightings require two bits for exactness.

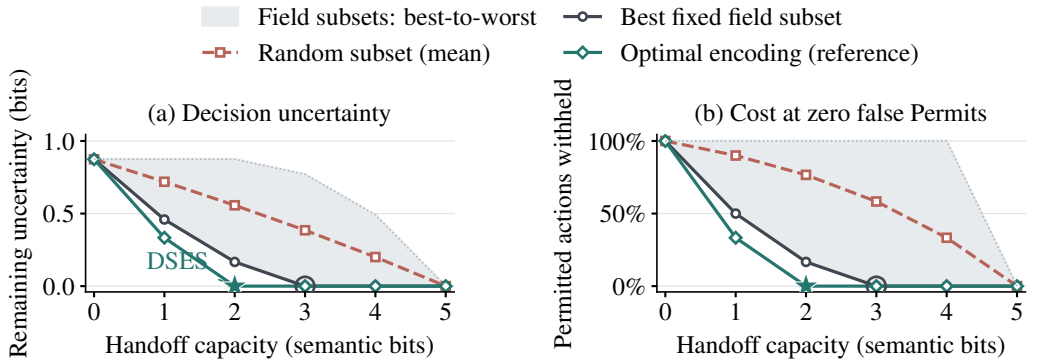


Figure 5: **Two bits also suffice under equal profile weights.** Each profile has equal probability, with uniform states within profiles and uniform candidate actions. Field subsets, reference encodings, and markers match Figure 3.

### B.5 Decisions from incomplete records

A separate release predicate computes action answers without using the monitor table or compiler. An *opening* discloses a selected subset of the records listed in a signed coverage checkpoint. Each disclosed record must match its source, subject, and position in that checkpoint. For each of the five events, we enumerate three cases: the event did not occur, it occurred but its record was not disclosed, or it occurred and its record was disclosed. These cases give $3^5 = 243$ state/opening pairs. Disclosing every present record determines all five Boolean values. For incomplete openings, the checker retains every state consistent with the disclosed events. Hashes do not reveal undisclosed record contents. All 32 fully disclosed cases determine their state. Five incomplete openings already determine all three action answers; sixty more determine at least one action answer while leaving another unresolved. Every definite answer agrees with the true state. Figure 6 shows how these individual decisions become available before the complete profile is known.

Complete authoritative coverage can establish that no code check for the requested revision has passed. All three release actions then require Deny, even if approval status remains unknown. Once the code check passes, approval status matters for staging. The compatible-state set must retain the true current state: extra possible states can leave an answer Undetermined, whereas excluding the true state can produce an incorrect definite answer.

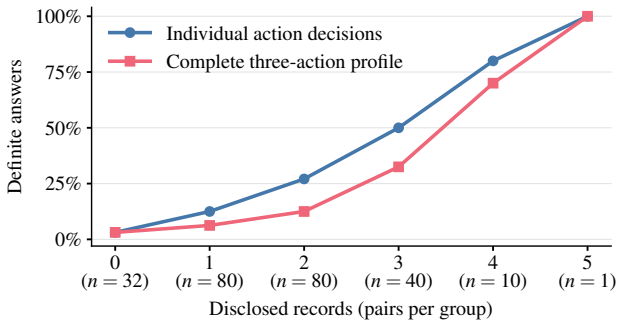


Figure 6: **Partial records can determine individual policy decisions.** Blue counts definite action answers out of $3n$; red counts determined profiles out of $n$ state/opening pairs. All definite answers are correct. Groups contain all record subsets of each size, not sampled runs.

# C Agent Evaluation

This appendix describes the policies and model experiments behind the results. We explain how the inputs are paired, how decisions and completed tasks are scored, and what the recorded errors reveal. The final section demonstrates reuse with an independently authored GitHub policy.

## C.1 Source policies

Table 7 identifies the protected action and its prerequisites for each workflow. Each policy specifies the required event and the artifact, resource, or run it must concern. For example, GitLab records deployment approvals separately from starting the job (GitLab, 2026). The handoff determines whether the delegated agent receives those approval records.

Table 7: **Task policies used in the eight source-derived workflows.** Each row states the configured condition that must hold before the named action. IDs S01–S08 are used in all per-policy results.

| ID | Policy source | When the protected action is permitted |
|---|---|---|
| S01 | GitHub required checks (GitHub, 2026c) | Merge only after the configured code check from the designated GitHub App succeeds for the candidate commit. |
| S02 | GitHub environments (GitHub Actions, 2026) | Deploy only after a listed reviewer other than the initiator approves the same artifact and environment. |
| S03 | AWS CodePipeline (Amazon Web Services, 2026) | Resume the release stage only after manual approval for the current pipeline execution and revision. |
| S04 | GitHub latest-push approval (GitHub, 2026c) | Merge only after a configured reviewer other than the latest pusher approves that pull request's current commit. |
| S05 | GitLab deployment approvals (GitLab, 2026) | Deploy only after two distinct configured reviewers, neither the initiator, approve the same deployment and environment. |
| S06 | Azure branch control (Microsoft, 2026a) | Start the stage only when both linked resources have confirmed allowed, protected branches for the current run. Unknown protection status blocks it. |
| S07 | Azure approval checks (Microsoft, 2026a) | Start the production stage only after approval for that run and stage by someone on the reviewer list fixed when checks began, other than the initiator. |
| S08 | Google Cloud Deploy (Google Cloud, 2026) | Deploy to the target only after a user or service with Cloud Deploy approver permission approves the matching rollout, release, and target. |

An independent source audit maps each documented condition to the local policy model and separates source requirements from test-specific identifiers. For example, the GitHub and AWS policies require checks on the current commit or execution, while Cloud Deploy approval precedes the later deployment step. We exhaustively enumerate reachable histories for all eight workflows.

A separately implemented direct predicate evaluates each workflow from its history, independently of the monitor and DSES compiler. It agrees with them on all 5,870 enumerated histories. To test whether the suite detects changed policies, we remove, relax, or invert individual conditions. The suite detects 50 of 53 modified policies. The three undetected changes affect no-self-approval conditions already enforced by the configured allowed-actor lists. Removing these redundant conditions changes no policy answer in the tested domain. Together, the source audit checks correspondence with the documented policies, and the execution tests check the resulting policy decisions.

### C.2 Model settings

All four model configurations receive text-only inputs. GLM-4.5-Air and GLM-4.6V use Zhipu, while Qwen3.7 Max and DeepSeek V4 Pro use Alibaba Model Studio in Singapore (Zhipu AI, 2026; Z.AI, 2026; Alibaba Cloud, 2026b;a). For reproducibility, we report the identifiers and settings of the served configurations. The dated Alibaba identifiers specify served versions; the GLM identifiers name endpoints without fixing the model weights, and inference compute may differ across endpoints. We hold settings fixed within each Policy/History comparison.

Table 8: Model settings. We submit output-token limits to each provider and report token usage from its responses.

| Exact model ID | Temp./top-$p$ | Output tokens | Reasoning |
|---|---|---|---|
| `glm-4.5-air` | 0.7/0.9 | 2048 | Enabled |
| `glm-4.6v` | 0.7/0.9 | 2048 | Enabled |
| `qwen3.7-max-2026-05-20` | 0.7/0.9 | 8192 | Enabled; Thinking budget 4096 |
| `deepseek-v4-pro-0813` | 0.7/0.9 | 8192 | Enabled; Effort high |

Fixed-action trials accept only a complete Permit or Deny answer; malformed or explanatory responses count as output failures. Search and code execution are disabled. Model-chosen actions and record queries use structured responses. Requests and responses are retained for replay.

Only provider or connection errors are retried, up to three times; model answers are not. Infrastructure attempts are recorded separately, and connectivity checks contribute no observations.

Our experiment design specifies 192 fixed-action judgments, 64 model-chosen trajectories, 18 resource episodes, and 40 diagnostic episodes per model. We archived model settings, source files, and trial order before collection.

### C.3 Paired decision evaluation

For each policy, we use one history where the candidate action is permitted and three where it is forbidden. The three pairs reuse the first history, so 24 pairs contain 32 distinct policy–history combinations. Two repetitions per pair and condition produce 48 Permit and 48 Deny trials in each condition per model. Table 9 reports workflow-level agreement, and Table 4 separates the directions of error.

Table 9: Correct policy decisions before and after adding cross-agent records (Policy → History), by workflow. Each entry counts correct answers out of twelve trials, retaining output failures in the denominator. Each condition has six Permit and six Deny trials per row. IDs follow Table 7.

| Policy | GLM-4.5-Air | GLM-4.6V | Qwen3.7 Max | DeepSeek V4 Pro |
|---|---|---|---|---|
| S01 | 6/12 → 12/12 | 6/12 → 12/12 | 6/12 → 12/12 | 6/12 → 12/12 |
| S02 | 6/12 → 12/12 | 6/12 → 11/12 | 6/12 → 12/12 | 6/12 → 12/12 |
| S03 | 6/12 → 12/12 | 6/12 → 12/12 | 6/12 → 12/12 | 6/12 → 12/12 |
| S04 | 6/12 → 11/12 | 6/12 → 10/12 | 6/12 → 12/12 | 6/12 → 12/12 |
| S05 | 6/12 → 12/12 | 6/12 → 12/12 | 6/12 → 12/12 | 6/12 → 12/12 |
| S06 | 6/12 → 12/12 | 6/12 → 12/12 | 6/12 → 12/12 | 6/12 → 12/12 |
| S07 | 6/12 → 12/12 | 6/12 → 12/12 | 6/12 → 12/12 | 6/12 → 12/12 |
| S08 | 6/12 → 12/12 | 6/12 → 11/12 | 6/12 → 12/12 | 6/12 → 12/12 |

A false Permit is a model answer of Permit to a forbidden request. A false Deny is an answer of Deny to a permitted request. We count output failures as incorrect, but not as false Permits or false Denies. We compare the Policy and History conditions using the same model, policy, instance, committed execution history, and repetition. For model-chosen execution, we match the initial state and repetition, then let each model choose its operations. For resource replay, we hold both the requested action and the state before the request fixed.

We quantify uncertainty in the probability that a model returns Permit when given an identical request and settings repeatedly. Among parsed answers, we compute exact 95% Clopper–Pearson intervals and record output failures separately. Each configuration has eight Policy groups (twelve requests each), eight permitted History groups (six requests each), and 24 forbidden History groups (two requests each). For each permitted History input, the model returns six Permits, with interval $[0.541, 1]$. Because the same histories are reused, these intervals describe repeated answers to fixed inputs rather than accuracy across independent tasks. The state enumeration is exhaustive, so it has no sampling intervals.

### C.4 Execution evaluation

At each step, the model chooses a pending operation until the task finishes or fails. The runner records execution order for policy scoring and rotates candidate positions independently of model responses. Each choice receives a fresh request; History includes committed operations but carries forward no earlier model messages.

Table 10 shows how GLM-4.6V completes an Azure-stage task but deploys before independent approval. The model deploys after self-approval, then obtains the required independent approval from a listed reviewer. Both requests pass authentication and scope checks. The agent completes every pending operation, so $S = 1$. It starts the stage before independent approval, so $A = 0$. The recorded operation order establishes the violation.

Table 10: **Task success despite starting the stage before independent approval.** The first row is the initial state; the model chooses the next two operations.

| Order | Accepted operation and actor | Policy before effect |
|---|---|---|
| Initial | Initiator approves the stage | Initial record |
| 1 | Initiator starts the production stage | Deny |
| 2 | Listed reviewer approves the stage | Permit |
| Outcome | All pending operations complete | $S = 1,\ A = 0,\ U = 0$ |

The model treats its initial self-approval as sufficient. The policy instead requires a prior approval from a listed reviewer other than the initiator; the independent history-based oracle therefore marks the stage action Deny.

**Output interruptions.** We retain all four combinations of task success and policy validity, $(S, A)$, including interrupted runs. GLM-4.5-Air has 25 Policy and two History interruptions; GLM-4.6V has seven Policy interruptions and three invalid actions; DeepSeek has four Policy interruptions and 17 null actions; Qwen has none. All 58 interruptions remain in the completion-rate denominator, and policy scoring includes every operation already executed.

### C.5 Error analysis

**Approval record relevance.** We test whether records establish two policy requirements: approval for the requested code revision and approval by a reviewer distinct from the initiator. We use one balanced pair per policy and two repetitions. Relevant and irrelevant conditions match record count and structure, not token count. Irrelevant records concern another object and cannot satisfy the requested action's prerequisite. With relevant records, GLM-4.5-Air, GLM-4.6V, Qwen, and DeepSeek give 7/8, 6/8, 8/8, and 8/8 correct binary answers, respectively. With irrelevant records, GLM-4.5-Air, Qwen, and DeepSeek each return eight Denies. GLM-4.6V returns five Denies, one Permit, and two output failures. Records about another object leave the requested action's prerequisite unresolved.

**Agent responses to missing information.** With three-valued answers under Policy, the models correctly return Undetermined in 3/8, 7/8, 7/8, and 4/8 trials, in the same model order. Under History, the models give 8/8, 7/8, 8/8, and 4/8 correct definite answers. The inputs declare which task's records have complete coverage. An absent required event establishes Deny only when coverage is complete for that task and event type. DeepSeek unnecessarily remains Undetermined three times and has one output failure despite that coverage.

The model can make one read-only query for committed records and their coverage, then must give its policy answer in the next response. The query does not itself decide whether the action is permitted. Only DeepSeek makes valid queries, in three of eight episodes. After receiving the records, it gives one correct definite answer, one unnecessary Undetermined answer, and one output failure. Its five non-query episodes correctly end Undetermined. GLM-4.5-Air, GLM-4.6V, and Qwen make no valid query and have four, four, and six output failures, respectively. Their other episodes mostly end Undetermined. In these record-query trials, no configuration consistently obtains the missing records and gives the correct definite answer.

### C.6 External policy integration

We evaluate reuse on QAStudio's independently authored GitHub workflow. For deployment events, its test job starts after a successful Production deployment and is skipped in Preview. A pinned GitHub expression evaluator runs the original condition (QAStudio-Dev, 2026b; GitHub, 2026b). The candidate action is starting this job in an uncancelled workflow with no prerequisite job or environment gate.

Saved public GitHub records show the expected contrast: the test job starts after a successful Production deployment and is skipped after a successful Preview deployment. The Preview summary is directly viewable (QAStudio-Dev, 2026a); both observations are included in the public artifact's `data/github/observed.json`. Our evaluator reproduces both outcomes from the original condition. We then test handoff checking and repair by deliberately omitting fields.

We test six deployment statuses in Preview and Production, comparing all twelve inputs with an independently written predicate. The inactive status does not trigger this workflow (GitHub, 2026a); both predicates also match the two public observations. Status alone leaves one conflicting pair and environment alone leaves five; together, or encoded by DSES, they determine whether the job may start.

**Integration procedure.** An adapter evaluates the original condition and exposes its inputs and Permit/Deny answers to the policy-agnostic checker. For a status-only handoff, the checker adds environment; for an environment-only handoff, it adds status; messages with both fields remain unchanged. The runnable example is included in the research package.

## D Policy Enforcement

This appendix explains how the resource applies a policy decision to the caller's actual request. We describe request verification, execution against the checked state, and the enforcement experiments.

Policy owners supply the requirements and trusted record sources. The enforcement guarantee assumes a correct policy review process, monitor, compiler, key registry, state management, and policy enforcement point (PEP). It also assumes unforgeable signatures and collision-resistant hashes. The designated record sources must report events truthfully; their signatures identify them.

**Field provenance in the release example.** The table names the record source and action binding for each release field. A deployment authenticates each record with its registered source key and checks that it applies to the proposed action. Policy owners designate the sources they trust to report these events. Coverage assertions must name the task, resource, and version they cover. Without authoritative coverage, absence is unknown rather than false.

Table 11: **Source and scope for release-policy fields.**

| Field | Source record and extraction | Required binding |
|---|---|---|
| Code-check success | Record from the designated code-check service; extract its result. | Repository and candidate commit. |
| Independent approval | Approval record and configured reviewer/initiator identities; test reviewer eligibility and independence. | Same release target and requested operation. |
| Approval for requested revision | Revision named in the approval record. | Must match the revision in the candidate action. |

### D.1 Request verification

**Messages and verification context.** Write the signed DSES envelope as $\mathsf{env} = (b, \sigma_M)$ and the signed action request as $r = (r_0, \sigma_C)$. Here $b$ and $r_0$ are unsigned message bodies. $\sigma_M$ is the monitor's signature on $b$, and $\sigma_C$ is the caller's signature on $r_0$. The envelope binds code $\mu$ to the session, task, recipient, reviewed policy, interface, checkpoint, and epoch, and includes an expiry. The request carries this envelope, the claimed principal $p_r$, selected action and intended effect, and a unique nonce $n$. A *principal* is the user or service identity under which a request acts. In the request, $p_r$ names the claimed identity, whereas $c_{\mathrm{live}}.\mathsf{principal}$ names the caller authenticated by the resource.

The resource resolves the action identifier against its installed interface and compares the resulting action tuple with the intended effect. It executes only these verified parameters. Its live context supplies the authenticated caller, current scope and state version, registered keys, clock, and used nonces. Below, $u_r$ is the action selected in the signed request, $\bar{u}_r$ is the action reconstructed from its proposed effect, and $o_i^{\mathrm{live}}$ is the recipient's current input for ordinary authorization.

**Ordinary authorization.** For $h = e_1 \cdots e_t$, the log retains the action $u_j$ and acting agent $i_j$ for each committed event $e_j = \eta(u_j)$. Write $h_{<j} = e_1 \cdots e_{j-1}$ for the preceding history, with $h_{<1} = \epsilon$. Define

$$\mathsf{LocalAdm}_z(h) = \bigwedge_{j=1}^{|h|} \lambda_{i_j}\left(O_{i_j}^{z,\mathrm{pre}}(h_{<j}), u_j\right). \tag{20}$$

The value is one exactly when every executed action passed ordinary authorization. The empty conjunction is true. An authorized deployment can still violate the task policy if it occurs before approval.

At the resource, ordinary authority and exact effect binding require

$$\begin{aligned} \mathsf{AuthScope}_I(r; c_{\mathrm{live}}) = 1 \quad \Longleftrightarrow \quad & p_r = c_{\mathrm{live}}.\mathsf{principal} \\ & \wedge\, \lambda_i(o_i^{\mathrm{live}}, u_r) = 1 \\ & \wedge\, u_r \in \mathcal{C}_I \;\wedge\; \bar{u}_r = u_r. \end{aligned} \tag{21}$$

Scope authorizes complete action tuples. Permission for one operation on one resource does not authorize a different combination. The recipient identifier names the agent for which the interface was constructed. The principal identifies the caller making the request. These identities may differ.

**DSES message verification.** Let $\langle b \rangle_M$ and $\langle r_0 \rangle_C$ be the bytes covered by the monitor and caller signatures. They use canonical encodings with distinct message-type tags, preventing one signature from being reused as the other. Registered keys identify the monitor and caller public keys; the PEP rejects unknown keys. We write these keys as $pk_M$ and $pk_C$, and the resource's current clock time as $t_{\mathrm{now}}$. Define $\mathsf{Vrfy}(pk, \beta, \sigma) \in \{0, 1\}$ to return 1 exactly when signature $\sigma$ on bytes $\beta$ verifies under public key $pk$.

The issuance and expiry times are checked against the resource clock. Let $\xi$ select the session, task, recipient, policy-review, interface, checkpoint, and epoch values compared with the installed context. For a well-formed request and validated interface, the request's envelope is $\mathsf{env} = r_0.\mathsf{envelope}$. Let $\mathcal{N}_{\mathrm{used}}$ be the stored caller–nonce pairs before the current request's reservation. The verification

condition is

$$\begin{aligned}\mathsf{CurrentEnvelope}_I(r, \mathsf{env}; c_{\mathrm{live}}) = 1 \iff\ & \mathsf{Vrfy}(pk_M, \langle b\rangle_M, \sigma_M) = 1 \\ & \wedge\, \mathsf{Vrfy}(pk_C, \langle r_0\rangle_C, \sigma_C) = 1 \\ & \wedge\, \xi(b) = \xi(c_{\mathrm{live}}) \\ & \wedge\, b.\mathsf{issued_at} \le t_{\mathrm{now}} \le b.\mathsf{expires_at} \\ & \wedge\, (p_r, n) \notin \mathcal{N}_{\mathrm{used}} \\ & \wedge\, \mu \in m_I(Q_I).\end{aligned} \tag{22}$$

Nonces are unique per caller across sessions. Policy-review and record checks occur at issuance, and the PEP compares their digests with installed current values at use.

The request-check rule combines ordinary authorization, verification of the current envelope, and the decoded policy decision:

$$\begin{aligned}\Gamma_I(r, \mathsf{env}; c_{\mathrm{live}}) = \mathsf{AuthScope}_I(r; c_{\mathrm{live}}) \\ \wedge\, \mathsf{CurrentEnvelope}_I(r, \mathsf{env}; c_{\mathrm{live}}) \wedge [D_I(\mu, u_r) = \textsc{Permit}].\end{aligned} \tag{23}$$

Only after the authorization and envelope checks pass does the PEP evaluate $D_I(\mu, u_r)$. The signed decision record distinguishes verification failure, a decoded Deny, and a tool failure after permission to execute. If verification prevents decoding, it records an escalation rather than a policy answer. If the records leave a policy decision Undetermined at issuance, the monitor issues no envelope.

### D.2 Execution consistency

The PEP verifies and executes an allowed operation in one atomic transaction, recording the nonce, operation result, new state version, and signed decision together. Every executed prerequisite or protected action advances the epoch, even if its policy decisions remain unchanged. After one request advances the epoch, a second request carrying the old epoch fails verification. The PEP also stores used nonces to prevent the same request from executing twice.

**Nonce reservation and availability.** For a parsed request in a known session, the PEP reserves the claimed caller–nonce pair before authentication. If a later check rejects the request, the PEP records the reason and consumes the nonce without advancing the epoch. As a result, an invalid request can consume another caller's nonce before that caller uses it. A subsequent legitimate request using the same caller–nonce pair is then rejected. This behavior affects availability; the policy-validity guarantee concerns the operations that are allowed to execute. Malformed requests, unknown sessions, and replayed requests consume no new nonce. An execution exception rolls back all writes.

**Integrity of accepted requests.** Under the trust assumptions in Sections 3.2 and 4.4, accepted requests have valid signatures, match the installed state, and use an unused nonce. Altering signed contents requires forgery or a hash collision. Corollary 5 then gives policy validity of committed operations.

The transaction protects the local table in which tool effects are stored. For a remote tool, the resource must execute the operation only if the checked state or version still holds. An idempotency key lets the tool recognize a repeated request and prevent duplicate execution. The tool must also check whether another operation has changed a prerequisite.

### D.3 Enforcement evaluation

The model can choose merge, staging, production, or an operation that obtains a prerequisite. Ready has all prerequisites; Obtainable allows them to be obtained; in Unavailable, independent approval cannot be obtained. Two repetitions across these conditions and Observe, Online, and DSES give 18 episodes per model. Each trajectory allows at most seven choices and four protected-action attempts. Unavailable tests enforcement under repeated requests. Models receive Policy or History inputs.

All three methods apply the same checks for caller authority, signatures, message bindings, freshness, nonces, and effect parameters. Observe records the policy decision without blocking forbidden operations. Online evaluates the current records. Under DSES, the resource-side decoder uses the

signed code to evaluate the model's selected action. During model inference, the logical clock stays fixed while each committed operation advances the epoch. The experiment tests state-version checks, not expiry during a long model call.

Table 12: Resource episodes across four model configurations. Each method has 16 feasible episodes and 24 total episodes. Each method follows its own model-chosen trajectory.

| Method | Policy-valid task completions | Episodes with a forbidden commit |
|---|---|---|
| Observe | 15/16 | 3/24 |
| Online | 15/16 | 0/24 |
| DSES | 16/16 | 0/24 |

Agents complete 16/16 feasible tasks under DSES and 15/16 under Online. The difference is one GLM output failure. Neither enforcement method allows a forbidden operation. Under Observe, the resource executes six forbidden operations across three Unavailable episodes.

For replay, we verify original signatures, request context, histories, and policy decisions, then reconstruct the valid pre-request state and issue fresh bindings for each method. Three of 181 requests follow an already-invalid Observe history and are excluded; the remaining 178 cover 45 state–action pairs and yield 534 PEP evaluations (Table 13).

Table 13: **Online and DSES enforce the same policy.** We replay 178 requests from valid states across 45 distinct state–action pairs.

| Method | Allowed effects committed | Forbidden effects committed |
|---|---|---|
| Observe | 156/156 | 22/22 |
| Online | 156/156 | 0/22 |
| DSES | 156/156 | 0/22 |

The official A2A SDK integration commits a local effect for a correctly bound request and rejects a subsequent incomplete request without another effect. Protocol tests also reject altered, stale, replayed, misbound, and unsupported requests. Partial-record tests verify the source and coverage checkpoint before issuance. These tests exercise the trusted-source model in Section 3.2 and the local commit procedure in Appendix D.2.

## E Reproducibility

The public artifact root has `README.md`, `src/`, and `data/`. It contains the source-derived policies, handoff checker, field-repair code, and saved model and resource records. The README indexes the reproduction commands; `data/README.md` maps each saved result to the corresponding study. From that root, use Python 3.12+ and Node.js 20+:

```
python3.12 -m pip install -e .\[test,plots\]
python3.12 -m pytest -q
python3.12 scripts/check_results.py
python3.12 examples/github/run.py \
    --output /tmp/policyfork-github-example
```

The result checker verifies the saved handoff, model, resource-replay, and field-repair results. The example evaluates an independently authored GitHub workflow condition, returns conflicting execution pairs, selects repair fields, and checks the repaired handoffs. Once dependencies are installed, these checks use bundled data without credentials, network calls, or model API calls. `examples/github/README.md` describes the example and its output.

Regression tests compare the checker with direct pair enumeration on 3,840 combinations of four-run/two-action tables and input partitions. Repair tests compare smaller field sets on 256 answer assignments. The data guide also locates the saved construction-cost measurements and figure scripts.