The Alignment Flywheel: A Governance-Centric Hybrid MAS for Architecture-Agnostic Safety

Elias Malomgré[1,2] [0009−0000−3052−7047] ✉ and Pieter Simoens[1,3] [0000−0002−9569−9373]

[1] IDLab, Ghent University - imec, Belgium
[2] elias.malomgre@ugent.be
[3] pieter.simoens@ugent.be

Abstract. Multi-agent systems provide mature abstractions for role decomposition, coordination, and normative governance, but increasingly capable learned components make post-deployment safety harder to inspect, audit, and update. When safety behavior is absorbed into a decision component, narrow failures may require retraining or rollback of the full component. This paper specifies the Alignment Flywheel as a governance-centric hybrid MAS architecture that decouples decision generation from safety governance. A Proposer generates candidate trajectories; a governed Safety Oracle stack returns safety scores, prediction uncertainty, audit coverage uncertainty, and evidence hooks through a stable interface; and an Enforcement layer applies explicit risk policy at runtime. Around this loop, a governance MAS performs monitoring, red-teaming, verification, triage, refinement, and versioned release management. The central engineering principle is patch locality: many newly observed safety failures can be mitigated through small governance batches for the Oracle stack and its audit state rather than by retraining or retracting the Proposer. The architecture is implementation-agnostic with respect to both Proposer and Oracle, and defines the roles, artifacts, protocols, and release semantics needed for runtime gating, audit intake, signed updates, staged rollout, and rollback. We demonstrate executability in two scenarios: a learned spatial Oracle patched through regression-checked governance updates, and a clinical GenAI proxy setting illustrating structured norms, escalation, and audit coverage. Our implementation code and documentation is available open source at https://github.com/decide-ugent/Alignment-Flywheel.

Keywords: Multi-agent systems · hybrid agent architectures · AI governance · runtime enforcement · safety oracle · Alignment Flywheel

1 Introduction

As models and autonomous components become more capable, a central challenge is no longer only how to make them perform well, but how to make them behave acceptably under the rules, norms, and risk limits of the settings in which

they are deployed, often discussed as *alignment*. In MAS, this becomes an operational problem: heterogeneous autonomy components must be integrated into coherent, governable deployments, and safety or compliance failures rarely originate in a single module. Instead, they often emerge at interfaces, under version skew, and during asynchronous updates across components that evolve at different speeds, a pattern long observed in complex ML systems as dependency entanglement, hidden feedback loops, and hard-to-localize regressions accumulate [38, 10]. In production, such failures are frequently *silent*, becoming tractable only through end-to-end observability across the full pipeline [40], and these operational governance problems are cross-component and lifecycle-driven rather than localized to model internals [45].

A recurring difficulty is that rules a system should follow are often absorbed into the internal parameters of the decision policy itself [13, 34, 43]. This applies to any autonomous agent, not only foundation models, whose behavior changes through retraining, optimization, model replacement, or configuration updates. When acceptable behavior is encoded mainly in the policy, it becomes harder to inspect, patch, qualify, and redeploy that behavior when norms, regulations, or risk tolerances change [25], especially under dataset shift and drift [33, 16]. In this paper, we use *governance* in an operational sense, a layer that decides whether proposed behavior should be allowed, blocked, deferred, escalated, patched, or rolled back, and how those decisions are updated over time. We, therefore, propose a different engineering unit of change for that layer: small, targeted governance patches that constrain newly unsafe trajectory classes without requiring full policy retraction or redeployment. This resembles runtime enforcement [32] and *shielding* [3, 19], but our focus is on the engineering needed to make governance patchable, auditable, and operable across a hybrid MAS deployment.

Building on the Alignment Flywheel vision of Malomgré and Simoens [25] and their prior work showing that IIRL can instantiate a standalone Safety Oracle queried independently of the underlying Proposer policy in robotic locomotion [24], this paper specifies the hybrid MAS architecture, protocol layer, and deployment semantics needed to operate such a Flywheel in practice. The earlier work framed alignment as an ongoing governance process rather than a one-time training outcome; the present paper turns that lifecycle view into an executable Proposer-Oracle governance architecture. IIRL is used as a convenient reference instantiation, but the Oracle role is method-agnostic: any safety artifact satisfying the same interface assumptions can be governed by the architecture.

Concretely, a Proposer generates candidate actions or plans as trajectories, while a Safety Oracle returns safety-relevant signals through a stable interface contract. The decision component does not contain all normative intelligence itself. Instead, an Enforcement layer interprets Oracle outputs under an explicit risk policy, and a governance MAS supervises the Oracle through monitoring, auditing, uncertainty-driven verification, and versioned refinement. The central engineering principle is *patch locality*: many safety fixes are applied to the governed Oracle artifact and its update pipeline rather than by modifying or retracting the Proposer policy. This is particularly valuable in hybrid systems [6, 31], where

components evolve at different cadences and incur different qualification costs; policy upgrades may be expensive to train and certify, whereas Oracle patches can be small, reviewable, and narrowly scoped to newly observed failures at the proposer-oracle boundary [38, 10, 40].

From a systems-engineering perspective, the architecture applies separation of concerns to governance. Monitoring, escalation, verification, refinement, enforcement, and release management are factored into interacting modules with explicit protocols and stable interface contracts. This supports modular contribution: one can improve a single role in the governance loop without understanding the internal learning dynamics of the Proposer or the full construction of the Oracle, provided the module consumes and produces the agreed artifacts correctly. A central part of this separation is the release model: Oracle updates are treated as versioned governance artifacts that can be rolled out across a fleet, monitored for regressions, and rolled back under bounded propagation delay [39]. Since these patches function as distributed governance controls, the architecture incorporates signed update metadata [27], anti-rollback checks [20], and monitoring for proposer-oracle interface regressions such as uncertainty-calibration drift [30] and decision-distribution shifts conditioned on Proposer version [40].

Running Example 1 (Clinical GenAI Assistant) *A hospital deploys a clinician-facing Clinical GenAI Assistant that drafts patient-facing messages, summarizes chart context, retrieves approved institutional guidance, and proposes workflow actions. The Assistant acts as the Proposer. A Safety Oracle evaluates candidate replies or action plans before execution. If the Oracle's prediction is uncertain or if the Alignment Flywheel has insufficient audit coverage for this kind of case, the system must not answer definitively. Instead, it must abstain (e.g., respond with "I don't know") or escalate for clinical review.*

This paper makes four contributions to the engineering of deployable hybrid multi-agent systems. First, it defines a Proposer-Oracle topology and a trajectory-level gating model that applies to both single-step actions and multi-step plans across domains and modalities. Second, it specifies the Alignment Flywheel as an executable hybrid MAS design, with coordinated roles for monitoring, escalation, auditing, refinement, and enforcement, together with the artifacts they exchange and their authority boundaries. Third, it formalizes an Oracle interface contract covering decision outputs, prediction uncertainty, audit coverage uncertainty, and evidence hooks, enabling audit and patch workflows while preserving architectural invariants such as patch locality and version stability. Fourth, it introduces deployment semantics for proposer-oracle systems in which safety fixes are released as small, versioned Oracle patches rather than as full policy redeployments, including progressive rollout, regression monitoring, rollback under bounded propagation delay, and signed update metadata for fleet distribution.

2 Background and Positioning

This section positions the paper relative to operational governance, candidate Safety Oracle constructions, and runtime-control architectures. The goal is not to argue for one universally best Oracle method, but to clarify the interface assumptions that make an artifact governable, patchable, and deployable. A broader discussion of alternative Oracle families is deferred to Appendix B.

Governance In this paper, *governance* is used in an operational rather than purely prescriptive sense. Much AI governance work specifies principles, documentation requirements, explanations, or organizational responsibilities [26, 8, 7]; such mechanisms are necessary, but they do not, by themselves, determine how a running autonomous system should react when a new failure class is observed. Here, governance refers to the control machinery by which explicit constraints, uncertainty signals, verification results, escalation decisions, release controls, and rollback mechanisms shape whether a proposed trajectory is trusted, blocked, deferred, escalated, patched, or rolled back in deployment. This is broader than runtime filtering alone [32] and narrower than alignment in the most general sense: the focus is on how governance judgments are represented, exchanged, enforced, and updated across interacting modules in a deployable system. This view is compatible with normative MAS work that treats governance as explicit administration through norms and related state rather than as a property hidden inside individual agents or policies [42, 9].

Candidate Safety Artifacts and Oracle Constructions A Safety Oracle is any artifact that evaluates candidate actions, trajectories, or plans through a stable interface and returns deployment-relevant signals such as a safety score, uncertainty, or evidence hooks. Several artifact families can instantiate this role. Preference-shaped policies such as RLHF and DPO improve behavior directly, but tend to entangle governance with the policy, making independent patching and audit harder [48, 29, 34]. Demonstration-derived evaluators, including IRL-style reward models, externalize evaluation but depend on expert coverage and may generalize poorly outside demonstrated regions [28, 1, 47, 2, 4]. Constraint- and logic-based artifacts, guardrails, monitors, and shields can be more interpretable or formally grounded, but require the relevant structure to be specified and observable at runtime [23, 36, 3, 19]. IIRL appears especially promising because it yields a standalone evaluative artifact with comparatively weak operational assumptions [25, 24]; however, the Proposer–Oracle design remains method-agnostic. Appendix B.2 expands this comparison.

Hybrid Agent Architectures, Interfaces, and Governed Deployment The core engineering difficulty in hybrid agent systems is the asynchronous evolution of heterogeneous components. A Proposer may change through retraining or replacement at a slow, expensive cadence, while governance constraints, guardrails, or shielding mechanisms must adapt much more quickly as new failure cases are observed. This concentrates risk at interfaces, where representational drift, version

skew, and calibration mismatch can produce hard-to-localize regressions [38, 10]. Pipeline-level observability is therefore needed to attribute failures across data, models, and surrounding components [40]. A practical deployment analogue of this external governance is the guardrails pattern: constraining a powerful generator at runtime using modular checks and rules external to the base model, as in NeMo Guardrails [36]. Runtime enforcement and shielding similarly separate optimizing policies from external safety mechanisms, often with stronger formal guarantees [3, 19]. In the Flywheel architecture, such guardrails and shields can themselves be treated as candidate safety artifacts, provided they expose the required governance interface.

3 System Overview: Roles, Artifacts, and Interfaces

The Alignment Flywheel treats AI safety not only as a training problem, but as an externalized control problem. It wraps a Safety Oracle with a governance layer that checks candidate trajectories against explicit norms, operational evidence, prediction uncertainty, and audit coverage. This is especially relevant when the Oracle is supplied by a third party: the vendor may provide a strong statistical artifact, but cannot be expected to encode dynamic regulations, organization-specific compliance interpretations, approval rules, or documentation practices. The Flywheel therefore maintains the normative specification Φ and supervises the Oracle against current governance policy.

The same mechanism supports alignment-as-a-service during training, fine-tuning, or online adaptation. The Flywheel monitors visited contexts and trajectory segments, routes uncertain or insufficiently audited regions to verification and refinement, and uses end-to-end observability to attribute failures to cross-component interface changes rather than isolated modules [40]. Because verification and refinement capacity may be limited, oversight is designed to be scalable: agents handle routine discovery, clustering, and patch preparation, while humans can intervene at the level of norms, thresholds, escalation rules, and release approvals. Appendix E details this tunable oversight model. In this sense, the Flywheel acts as an independent regulatory shell for runtime enforcement, audit intake, refinement, and documentation support [35]. The minimal notation needed to follow the rest of the paper is shown in Table 1.

3.1 Multi-Agent Governance Roles

The architecture comprises five specialized roles as shown in Table 2. Crucially, each role can be instantiated by an autonomous agent, a human expert, or a hybrid collective, supporting configurable degrees of autonomy and escalation [37].

3.2 Coordination Layer C

The Flywheel does not require a particular coordination mechanism. We specify a coordination layer C that provides the semantic operations needed by the

Table 1. Minimal notation used in the architecture.

Symbol	Meaning
Σ	context (task state, inputs, metadata; any modality)
τ	trajectory (action, tool call, message, or plan)
P	Proposer producing τ_{cand} from Σ
O	Safety Oracle (third-party statistical artifact)
s	Safety Oracle raw safety score
u	Safety Oracle prediction uncertainty
u_{thresh}	vendor-defined threshold for acceptable prediction uncertainty
u_a	audit coverage uncertainty generated by the Alignment Flywheel
$u_{a,thresh}$	threshold for acceptable audit coverage uncertainty
ΔO	local correction inside a governance batch
B_O	batched governance update for a Safety Oracle
E	Enforcement Layer (gates execution, logs evidence)
Φ	Normative Specification (maintained by the MAS)
v_O	Oracle / governance-batch version identifier
C	Coordination layer
K	Knowledge Base (demonstrations, failures, patches, ledger)
Q_{ver}	Verification Queue (raw candidates from Red Team / monitoring)
Q_{ref}	Refinement Queue (clustered / prioritized flaws for correction)

Table 2. Governance roles in the Flywheel.

Role	**Goal**	**Input**	**Output**
Red Team	find false negatives	O, τ, Σ	candidates for Q_{ver}
Blue Team	detect drift	traces, stats	drift alerts
	detect vulnerabilities	failures, anomalies	audit intake
	monitor deployment	runtime traces	deployment reports
	steer Red Team	risk summaries	audit priorities
Verification Team	check against Φ	Q_{ver} items	verified breaches
Triage Team	govern Q_{ver}	candidate cases	prioritized Q_{ver}
	govern Q_{ref}	verified breaches	prioritized Q_{ref}
Refinement Team	patch and package	Q_{ref} items	ΔO, B_O

governance roles: publishing typed governance records, routing work items to role-specific backlogs, preserving lineage between records, exposing queryable operational views, and recording versioned release transitions. The architecture therefore constrains the protocol semantics between roles, not the concrete coordination substrate. In the remainder of the paper, for clarity and traceability, we instantiate C as an append-only blackboard-style knowledge base that stores the artifacts shown in Table 3; governance-relevant state changes are recorded as immutable, typed artifacts, and operational state is derived by replaying or querying these artifacts rather than by mutating shared in-memory state. This supports stateless agents (Sec. 4.1) and makes audits, rollback, and release analysis traceable.

Table 3. Typed artifacts stored in K.

Artifact	Purpose	Producer
Φ	norms, tests, severities	governance authority
F	confirmed breaches + evidence	verification
Q_{ver}, Q_{ref}	queue pointers + priority	monitoring / triage team
B_O	batched governance update	refinement team
L	rollout / rollback history	release manager
reports	drift, vulnerabilities, coverage	blue team

Each artifact in K carries explicit type and lineage information, so derived artifacts record their inputs (e.g., VerificationResult→CandidateFlaw, GovernanceBatch→VerifiedBreach set). This supports provenance queries such as "why was this blocked?" and lets queues store references into K rather than full payloads.nTo support monitoring and steering at scale, K also maintains queryable operational views, including norm coverage over Φ, drift and novelty summaries, open breach clusters with risk scores, and fleet rollout state derived from the release ledger.

3.3 Oracle Interface Contract (the query interface)

To preserve strict separation of concerns, the third-party Safety Oracle O is treated as a black-box statistical evaluator. It does not contain symbolic business logic, nor does it know the current regulations Φ. It evaluates a trajectory and outputs its internal statistical state. The Enforcement Layer E interacts with the deployed oracle stack through a single query interface:

Inputs: Context Σ, candidate trajectory τ, and return flags $f_s, f_u, f_{u_a}, f_{thresh} \in \{0, 1\}$ indicating whether the caller requests the raw safety score, prediction uncertainty, audit coverage uncertainty, and threshold values.
Outputs (as requested by flags):
(1) $s \in \mathbb{R}$: the raw safety score.
(2) $u \in \mathbb{R}$: the Oracle's internal prediction uncertainty for s.
(3) $u_a \in \mathbb{R}$: the audit coverage uncertainty measured by the Alignment Flywheel.
(4) $(u_{thresh}, u_{a,thresh})$: the configured thresholds for acceptable prediction uncertainty and audit coverage uncertainty.
(5) v_O: the version identifier of the deployed oracle / governance-batch state used.

The query interface is therefore vendor-backed but governance-mediated: vendor-side signals such as s and u are returned by the Safety Oracle itself, while governance-side signals such as u_a are attached from the currently deployed Flywheel governance state. This preserves a single enforcement-facing protocol while keeping the provenance of the signals distinct.

Rather than distributing individual local corrections or isolated audit-coverage updates, the Flywheel releases signed governance batches B_O as the engineering unit of change. A batch may contain one or more local corrections ΔO, updates

Table 4. Example enforcement policy.

Condition	Action
$u \geq u_{thresh}$ or $u_a \geq u_{a,thresh}$	fail-closed or halted; send (priority) audit to Q_{ver}
s unsafe, uncertainty acceptable	block
s safe, uncertainty acceptable	allow; may later be audited

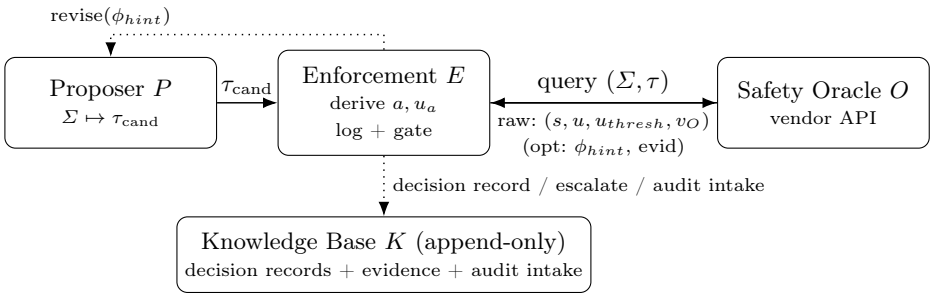


Fig. 1. Runtime enforcement during deployment. From context Σ, proposer P generates a candidate trajectory τ_{cand}. Enforcement E queries the deployed oracle stack, which returns the requested signals, e.g. $(s, u, u_{thresh}, u_a u_{a,thresh}, v_O)$, where vendor-side outputs and Flywheel-side governance outputs are combined under a single query interface. Enforcement then derives the action a and the uncertainty state u, logs the decision, and writes the audit intake to the append-only knowledge base K. Dotted paths indicate optional revision and escalation under the configured risk policy.

to the state used to compute u_a, threshold changes, and the linked regression evidence required for rollout and rollback.

3.4 Enforcement Logic and Prioritization

Because the Oracle outputs only raw signals, the normative intelligence resides within the MAS. As shown in Figure 1, the Oracle does not decide policy; instead, the Enforcement layer interprets the returned signals, applies the configured risk policy, logs the outcome, and escalates relevant cases for later hardening. A simple risk policy is shown in Table 4.

Running Example 2 (Clinical GenAI Assistant: governed query) *A clinician asks the Clinical GenAI Assistant to draft a patient-facing reply about whether a recently prescribed medication should still be taken after the patient reports a possible side effect. The Proposer generates a candidate message and a short action plan. The deployed oracle stack is queried with* (Σ, τ)*, where* Σ *contains the clinical context and retrieved evidence, and* τ *is the proposed reply-and-action trajectory. The vendor Oracle returns a safety score* s *and prediction*

uncertainty u. The Flywheel layer attaches audit coverage uncertainty u_a, reflecting that this class of advice has only limited prior verification coverage under the hospital's current governance policy. Because $u_a \geq u_{a,thresh}$, Enforcement does not allow direct execution and instead routes the case to Q_{ver} for verification.

4 Interaction Protocols and OODA Dynamics

The Alignment Flywheel coordinates distributed governance agents through role-specific OODA loops (Observe–Orient–Decide–Act) [11]. The OODA abstraction separates the *mechanism* of governance—which records are read and written—from the *strategy* used by a role, such as fuzzing, gradient-based search, prompt mutation, symbolic checking, or patch synthesis [21, 18, 22]. Governance agents interact only through the append-only Knowledge Base K and its queues, so agents can be restarted, replicated, or replaced without invalidating global governance state. Appendix C gives role-specific OODA pseudocode.

4.1 OODA Abstraction

Each governance role executes its OODA loop over the coordination layer C. The role observes governance state through C, orients relative to its objective, selects a strategy module, and publishes the resulting governance records back through C. In the reference instantiation summarized in Table 5, these operations correspond to reading from K and Qver/Qref and appending new records to K. This keeps the paper's running example concrete while preserving the architectural distinction between the logical governance protocol and the concrete blackboard-style coordination mechanism.

Table 5. Role-independent OODA template for governance agents.

Phase	Role-independent function	Example
Observe	Read relevant records from K and queue heads.	Red Team reads active v_O and high-severity norms
Orient	Interpret records relative to the role objective.	Red Team identifies a norm with low coverage.
Decide	Select a strategy module without changing the protocol.	Prompt mutation to verify norms for an LLM proposer
Act	Execute the strategy and write typed records back to K.	Candidate flaws are pushed to the queue

Running Example 3 (Clinical GenAI Assistant) *Red Team agents generate patient-message variants involving medication side effects, missing evidence, or unsafe disposition choices. Low-uncertainty claimed-safe replies are inserted into Q_{ver}, verified against Φ, and converted into breach clusters such as "unsupported medication advice" or "under-escalated urgent symptoms". These clusters become refinement jobs in Q_{ref} and later governed updates.*

4.2 Double-Filter Pipeline

Coordination is organized as a double-filter pipeline that separates discovery from correction as shown in Table 6. Stage 1 converts raw candidate flaws into verified breaches; Stage 2 converts verified breaches into governed updates. The two-stage structure prevents high-volume automated probes from overwhelming verification and refinement capacity. Triage can prioritize candidates and breach clusters using risk signals such as norm severity, dangerous certainty ($u_{\text{thresh}} - u$), novelty relative to prior history in K, and operational urgency [46].

Table 6. Double-filter pipeline from discovery to governed update.

Stage	Step	Function	Output
Verification	Injection	Red Team generates candidate trajectories τ and pushes them to Q_{ver}.	CandidateFlaw
Verification	Triage I	Candidates are prioritized by $\Delta = u_{thresh} - u$. Low-uncertainty claimed-safe cases are high-value false-negative audit targets.	prioritized Q_{ver}
Verification	Validation	Verification checks candidates against Φ and violations become immutable failure records in K.	VerifiedBreach
Refinement	Ingestion	Verified breaches are hash-stored in K for tracing and de-duplication.	breach records
Refinement	Triage II	Breaches are clustered into failure families, reducing cognitive load and enabling batch remediation [46, 44].	breach clusters
Refinement	Job creation	Triage packages a prioritized cluster as a repair task.	RefinementJob
Refinement	Correction	Refinement synthesizes a patch ΔO or governance-batch update and commits it to the release ledger L.	PatchCommit

4.3 Message Protocols

Inter-agent coordination is enforced through typed records stored in K. These records define the handoff points between governance roles and retain enough context for audit, replay, and patch attribution. Table 7 summarizes the main records; complete schemas are given in Appendix D.

5 Runtime Enforcement During Deployment

This section specifies how a deployed system uses a fixed Oracle/governance-batch version to gate Proposer outputs under latency constraints, how prediction

Table 7. Core protocol records exchanged through K and the queues.

Record	Producer / route	Purpose
CandidateFlaw	Red Team $\rightarrow Q_{ver}$	Discovery intake record containing (Σ, τ) and the Oracle signals observed at discovery time.
VerificationResult	Verification $\rightarrow K$	Explicit governance judgment separating raw suspicion from a confirmed or rejected violation of Φ.
VerifiedBreach	Verification $\rightarrow K$	Durable failure record created for confirmed violations; later clustered, prioritized, and linked to patches.
RefinementJob	Triage $\rightarrow Q_{ref}$	Clustered repair task that converts many verified breaches into one prioritized refinement unit.
PatchCommit	Refinement $\rightarrow K, L$	Governed update record binding a patch to its parent version, motivating breaches, regression evidence, and authorization signature.

and audit-coverage uncertainty trigger escalation, and how runtime evidence enters the next hardening cycle. The topology is consistent with safety filtering and runtime assurance architectures [14, 17], but the key difference is that each runtime decision is tied to a governed artifact lifecycle: versioned Oracle state, auditable records, uncertainty-driven intake, and regression-validated patch releases.

5.1 Execution Loop

At runtime, the Proposer P maps the current context Σ to a candidate trajectory τ_{cand}. The Enforcement layer E queries the active governed Oracle stack, maps the returned signals to an operational action, and records the decision in K. Table 8 summarizes the loop.

The Oracle stack is treated as a governed control-plane component: its deployed behavior is tied to an explicit version identifier and changes only through governed release.

5.2 Decision Policy and Escalation

The Oracle stack returns raw safety-relevant signals; the Enforcement layer maps those signals to an operational response. Table 4 shows the base policy used throughout the paper. The policy separates prediction uncertainty from audit coverage uncertainty: the former is produced by the Safety Oracle, while the latter is produced by the Flywheel governance layer.

Enforcement is not limited to a binary allow/block decision. If $a = \text{allow}$, E may execute τ_{cand} subject to local risk checks. If $a = \text{block}$, execution is denied.

Table 8. Runtime enforcement loop for a fixed governed Oracle version.

Step	Function
Propose	$P(\Sigma) \to \tau_{\text{cand}}$.
Oracle-stack evaluation	The governed Oracle stack evaluates $(\Sigma, \tau_{\text{cand}})$ and returns $(s, u, u_{thresh}, u_a, u_{a,thresh}, v_O, \phi_{\text{hint}}, \text{evid})$, where s is the raw safety score, u is Oracle prediction uncertainty, u_a is Flywheel audit coverage uncertainty, and $\phi_{\text{hint}}, \text{evid}$ are optional hooks.
Enforcement decision	E selects $a \in \{\text{allow}, \text{block}, \text{revise}, \text{escalate}\}$ under the configured risk policy. High u means the Oracle is unsure about its prediction; high u_a means the Flywheel has insufficient audit coverage for this class of case.
Log	E writes an auditable decision record to K, including the active version v_O for attribution, audit, and regression analysis.

If $a = \text{revise}$, E may request a revised proposal,

$$\texttt{revise}(\tau_{\text{cand}}, \phi_{\text{hint}}) \to \tau'_{\text{cand}},$$

which supports runtime correction without Proposer retraining. If $a = \text{escalate}$, the case is routed into the audit pipeline.

Risk posture is configurable by domain. For high-risk actions such as irreversible actions, privileged tool use, or high-impact actuation, E can fail closed on high uncertainty, insufficient audit coverage, or Oracle timeout; require explicit confirmation; or restrict tools to an allowlisted subset. For lower-risk actions, E may allow under moderate uncertainty while logging and escalating the case for later review.

5.3 Audit Intake, Evidence, and Latency

Prediction uncertainty, audit coverage uncertainty, and monitoring statistics are first-class runtime signals under dataset shift and interface drift [30, 40]. As shown in Table 9 E creates an audit case in K when $u \geq u_{thresh}$, $u_a \geq u_{a,thresh}$, ϕ_{hint} indicates a novel or unclassified constraint family, repeated block-revise cycles exceed a configured limit, monitoring detects anomalies, or the Oracle stack exceeds its latency budget. Depending on risk tier, timeouts trigger either fail-closed behavior or fail-open behavior with mandatory logging and escalation; sustained failures activate a degraded mode or circuit breaker.

These records ensure that future governance batches can be justified by concrete evidence in K, validated against regression suites, and attributed to a specific Oracle/governance-batch version.

Running Example 4 (Clinical GenAI Assistant) *A patient reports dizziness after starting a new medication. The Proposer drafts a reassuring reply and suggests no clinical review. The Oracle stack returns a safety score s and prediction uncertainty u, while the Flywheel attaches audit coverage uncertainty u_a for this case class. If $u \geq u_{thresh}$ or $u_a \geq u_{a,thresh}$, Enforcement fails closed: the*

Table 9. Runtime evidence captured for audit, replay, and later patch justification.

Record field	Purpose
Decision record	$(\Sigma, \tau_{\text{cand}}, a, s, u, u_{thresh}, u_a, u_{a,thresh}, v_O, \text{evid})$ records the context, candidate, action, safety score, uncertainty signals, thresholds, version, and evidence.
Integrity metadata	Hash of the serialized trajectory, timestamp, and host identity support replay and fleet correlation.
Optional attachments	ϕ_{hint}, counterexample identifiers, trace references, source deployment traffic label, stress test, or report.

draft is not sent, the case is logged in K, and a priority audit item is routed to Q_{ver}. If uncertainty is acceptable but s indicates unsafe, the reply is blocked or revised; only if s is safe and both uncertainty checks pass may the reply be sent.

5.4 Latency Budget and Timeouts

Runtime enforcement must respect latency bounds. The Enforcement layer, therefore, treats Oracle-stack availability as part of the risk policy. If the Oracle query exceeds its latency budget, E triggers a policy-defined fallback. For high-risk actions, the fallback is fail-closed: block execution and create a priority audit case. For lower-risk actions, the system may fail open, but only with mandatory logging and escalation. Sustained Oracle failures or repeated timeouts activate a circuit breaker that moves the deployment into a degraded mode, preventing cascading failures from a slow or unavailable governance dependency.

6 Reference Implementation and Evaluation

We implemented a reference Flywheel prototype with explicit modules for the Proposer, Safety Oracle, Flywheel Overlay, Query Merger, Enforcement layer, Knowledge Base, Red Team, Verification, Triage, Refinement, and release ledger. Components are selected through a single composition root, so different domains instantiate the same abstract interfaces with different concrete implementations. Of which our code is available at https://github.com/decide-ugent/Alignment-Flywheel. Governance state is stored as typed records in an append-only K, and updates are released as `Governance Batches` objects. Full implementation details, protocol schemas, and demo specifications are given in Appendices G.4 and G.5.

The evaluation is intended as an executable architecture demonstration rather than a domain-specific safety benchmark. We evaluate two deliberately different settings. The first is a learned 3D spatial Oracle derived from an IIRL-style [24] reward surface, used to test patch-local governance over a continuous learned artifact. The second is the Patient Portal setting from the running Clinical GenAI Assistant example, used to test the same governance protocol with structured norms, audit coverage uncertainty, and a heuristic proxy Oracle. Together, the

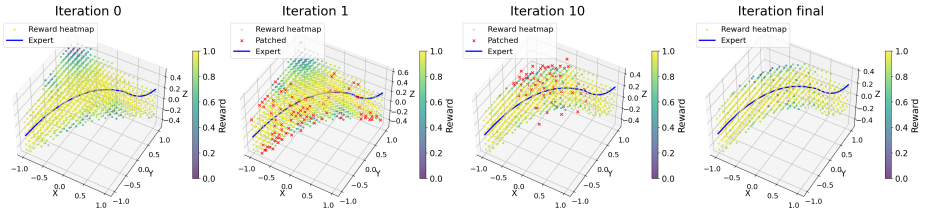


Fig. 2. 3D spatial Flywheel progression. The learned Oracle initially assigns non-trivial reward to many cells outside the expert-path basin. Governance batches add suppression kernels around verified flaw regions. Across iterations, the active reward surface contracts toward the expert trajectory while the sampled basin is preserved.

demos test whether the same Flywheel control plane supports different Oracle types, norm semantics, refinement mechanisms, and application domains.

6.1 Scenario 1: Offline Hardening of a Learned 3D Spatial Oracle

The spatial demo evaluates the Flywheel on a learned *continuous* IIRL Oracle in $[-1, 1]^3$, sampled on a 20^3 grid for discovery, regression checking, and visualization. Cells near the expert trajectory form the accepted basin; cells with non-trivial reward outside that basin are treated as false-positive reward flaws. The Flywheel must discover these flaws, suppress them through governed spatial patches, and preserve the sampled expert basin.

This scenario does not include a Proposer. Instead, it tests the Flywheel's offline hardening loop directly on the learned Oracle artifact. Behavioral change is produced only by governance batches that add Gaussian suppression kernels to the governed Oracle stack. A Patch Planner selects patch centers and bandwidths, predicts which neighboring flaw cells will also be covered, and shrinks or rejects patches that would damage the accepted basin. This tests patch locality at the Oracle level: the learned reward artifact is governed through small, versioned updates rather than retraining.

The adaptive Patch Planner eliminates all discovered flaw cells on the evaluation grid while preserving the sampled basin. The initial grid contains 4 927 flaw cells and 783 basin cells. After 12 governance iterations, the flaw count reaches zero and the sampled basin remains 783/783 preserved. A fixed-bandwidth baseline also preserves the basin, but remains far from convergence after the same number of iterations and has not converged after 25 iterations still active. Details of the Patch Planner, flaw classification, baseline, and per-iteration convergence are provided in Appendix G.4.

6.2 Scenario 2: Patient Portal Governance

The Patient Portal demo instantiates the running Clinical GenAI Assistant example. A patient-facing assistant receives a patient message, proposes a draft reply, and assigns a disposition such as `reply_only`, `nurse_review`, `clinician_review`,

Table 10. Patient Portal demo: enforcement convergence over evaluations.

Eval	Allow	Block	Escalate	Esc. Rate	Stack
0	6	0	9	60%	v0
1	6	3	6	40%	v1
2	9	6	0	0%	v2

or `urgent_escalation`. The Proposer is a passthrough component: it does not change during the experiment. The purpose is to test whether the Flywheel can turn uncertain or under-covered cases into definitive allow/block outcomes by updating only the Oracle stack and audit-coverage state.

The proxy Oracle scores candidate replies along four dimensions: medication content, urgency signals, evidence quality, and disposition mismatch. The Flywheel Overlay separately tracks audit coverage uncertainty u_a for case classes such as medication advice under weak evidence or lab-result interpretation without sufficient context. The normative specification Φ contains four typed norms: medication advice requires clinician review, urgency signals require escalation, lab results require context, and unsupported medication-change phrases are prohibited. The same runtime policy from Section 5 maps safety score s, prediction uncertainty u, and audit coverage uncertainty u_a to allow, block, or escalate decisions.

Table 10 shows the enforcement distribution over the fixed Patient Portal cases. Initially, safe routine cases are allowed, while unsafe and under-covered cases escalate. After the first governance batch, unsupported medication-change cases become definitive blocks. After the second batch, remaining lab and borderline cases are resolved: unsafe cases become blocks, while borderline cases with adequate disposition and established audit coverage become allows. The final evaluation reaches zero escalations while preserving correct separation between safe, unsafe, and review-covered cases.

This demo validates the governance pipeline rather than the medical adequacy of the proxy Oracle. Its purpose is to show that the same Flywheel machinery used in the spatial setting also operates in the running clinical example: runtime gating, audit intake, verification against typed norms, triage, refinement, governance-batch release, and re-evaluation. Additional Simple Medical and Complex Medical variants are reported in Appendix G.5 as ablations over Oracle complexity, norm structure, and triage strategy.

6.3 Evaluation Summary

Table 11 summarizes how the two scenarios exercise different parts of the same Flywheel architectural contract. The goal is not to compare domain performance, but to test whether the same control plane can be instantiated under different assumptions about the Oracle, norms, refinement mechanism, and deployment mode.

Table 11. Orthogonality of the two evaluation scenarios.

Dimension	3D spatial demo	Patient Portal demo	Architectural property
Evaluation mode	Offline hardening of a learned Oracle artifact	Runtime Proposer–Oracle enforcement and hardening	Offline and runtime governance
Oracle type	Learned continuous IIRL-style Oracle	Structured proxy Oracle with safety and uncertainty signals	Oracle substitutability
Proposer role	No Proposer; the Oracle is hardened directly	Passthrough Proposer producing candidate replies and dispositions	Separation of proposal and governance
Norm structure	Spatial support and expert-basin preservation	Medication, urgency, lab-context, and unsupported-advice norms	Norm-semantic variation
Refinement mechanism	Gaussian suppression patches released through governance batches	Blocks, disposition-sensitive decisions, threshold updates, and audit-coverage updates	Patch-local correction
Observed effect	False-positive reward cells are removed while the sampled expert basin is preserved	Escalations are converted into definitive allow/block outcomes across governance batches	Executability and protocol stability

The two scenarios support complementary claims. The 3D spatial demo isolates offline patch-local hardening of a learned continuous IIRL Oracle: false-positive reward regions are removed on the evaluation grid while the sampled expert basin is preserved, without retraining the learned reward artifact. The Patient Portal demo instantiates the full runtime Proposer–Oracle governance loop from the running clinical example, using a structurally different proxy Oracle, typed norms, disposition-based enforcement, and audit-coverage uncertainty. Across both domains, behavioral change is produced by governed updates to the Oracle stack and Flywheel state rather than by changing the decision generator or retraining the learned artifact.

The evaluations therefore establish executability, patch locality, and architectural substitutability under controlled conditions. They do not claim production safety, clinical readiness, or complete verification of all possible failures. Instead, they show that the proposed roles, protocol records, interfaces, and release semantics are sufficient to operate both offline Oracle hardening and runtime Flywheel governance, while motivating future work on stronger Oracle learning methods, richer norm logics, formal patch-safety guarantees, calibrated uncertainty, and long-horizon deployment studies.

7 Discussion, Limitations, and Conclusion

Operational implications. The Alignment Flywheel treats safety governance as an explicit control-plane workflow rather than as an opaque property of a monolithic decision model. The central claim is not that safety can be solved once and for all, but that many observed failures can be handled locally: candidate failures are logged, verified, triaged, converted into governance batches, regression-checked, released, monitored, and rolled back without necessarily retraining or retracting the Proposer. This shifts the engineering unit of change from the full decision component to the governed Oracle stack and its audit state, while preserving an auditable trace from runtime decisions and version transitions back to evidence and normative justification.

Evaluation scope and limitations. The implementation and demos demonstrate executability of the proposed roles, artifacts, interfaces, and release loop under controlled conditions. The 3D spatial demo shows offline hardening of a continuous IIRL Oracle evaluated on a finite grid: false-positive reward regions are discovered and suppressed while the sampled expert basin is preserved. The Patient Portal demo instantiates the running Clinical GenAI Assistant example and shows the full runtime governance loop, where prediction uncertainty, audit coverage uncertainty, typed norms, enforcement decisions, and governance batches convert initial escalations into definitive blocks or allows. These results support the architectural claims of patch locality, protocol stability, and component substitutability, but they are not production safety or clinical-validity claims. Oracle construction, richer norm semantics, formal patch-safety guarantees, calibrated uncertainty, human review costs, latency, and long-horizon adversarial adaptation remain follow-up work; Appendix F sketches how these extensions scale along oversight, modality, domain, and component axes.

Conclusion. We presented the Alignment Flywheel as a governance-centric hybrid MAS architecture for patch-local safety control. A Proposer generates candidate trajectories, a governed Oracle stack returns safety and uncertainty signals, an Enforcement layer applies explicit risk policy, and a governance MAS closes the loop through monitoring, verification, triage, refinement, and versioned release. The contribution is deliberately architectural: it specifies the control-plane roles, artifacts, protocols, and deployment semantics needed to make fallible autonomous systems auditable and iteratively governable. The reference implementation and demos show that the same protocol can support both offline hardening of a learned Oracle and runtime governance in a clinical-style Proposer–Oracle setting, establishing a concrete basis for future formal, algorithmic, and deployment-focused work.

Acknowledgment

This research was supported by funding from the Flemish Government under the “Onderzoeksprogramma Artificiele Intelligentie (AI) Vlaanderen” program.

A Appendix Overview

The appendix provides the technical material omitted from the main paper for space. Appendix B expands the conceptual positioning: it clarifies governance as an operational control layer, compares Safety Oracle families, and distinguishes prediction uncertainty from audit coverage uncertainty. Appendix C details the OODA-loop instantiations for the Red Team, Blue Team, Verification, Triage, and Refinement roles, showing what each role reads from and writes to the append-only knowledge base K. Appendix D defines the inter-role protocol artifacts, including candidate flaws, verification results, verified breaches, refinement jobs, governance batches, and release records. Appendix E describes the tunable human oversight model, including low-, medium-, and high-risk operating modes and the control surfaces exposed to human operators. Appendix G summarizes the reference implementation structure, norm representation, and governance-batch format. Appendix G.4 gives the full 3D spatial demo specification, including the continuous IIRL Oracle, kernel patch model, PatchPlanner, baseline comparison, and convergence results. Appendix G.5 gives the medical demo specifications, including the shared runtime pipeline, proxy Oracle variants, typed medical norms, enforcement mechanics, and cross-demo comparison.

B Extended Background on Governance and Safety Artifacts

B.1 Governance as an Operational Control Layer

The paper uses *governance* in a deliberately operational sense. The concern is not only whether a system satisfies a norm in principle, but how such norms, uncertainty signals, verification outcomes, escalation decisions, and release controls are represented and propagated in a running deployment. In this sense, governance sits between abstract alignment goals and concrete runtime control: it determines who is allowed to decide what, which artifacts are authoritative, how uncertainty is communicated, when a case must be deferred or escalated, and how corrective updates are qualified and rolled out.

This view is compatible with normative MAS perspectives in which governance is grounded in explicit norms and state transitions over those norms rather than hidden inside individual agents [42, 9]. However, the present paper is concerned less with the full social semantics of norms and more with deployable control structure. In particular, governance is treated here as a modular layer organized around explicit protocol responsibilities: monitoring, triage, verification, refinement, enforcement, and release management. This makes governance a separation-of-concerns problem as much as a normative one. A contributor can improve one role in the loop without needing to understand all other modules internally, provided the role respects the agreed interfaces and artifact semantics.

The Alignment Flywheel extends this idea in two directions. First, it provides *verification-as-a-service*: beyond the oracle's own prediction, it contributes

additional assurance through external auditing and verification. Second, it provides *alignment-as-a-service*: uncertainty and failures observed in deployment become inputs to iterative refinement of the governed artifact rather than mere post-hoc diagnostics. This is why the paper distinguishes oracle-side prediction uncertainty from Flywheel-generated audit coverage uncertainty. The former expresses how trustworthy the oracle believes its own judgment to be; the latter expresses how well that class of judgments has been externally covered by governance activity.

B.2 Possible Safety Oracle Instantiations and Their Assumptions

The proposer-oracle architecture is intentionally agnostic to the internal realization of the oracle. Different safety artifacts could instantiate this role, but they differ in the assumptions they make and in how naturally they support patchability, audit, and lifecycle governance.

Preference-shaped policies and implicit alignment. Methods such as RLHF and DPO can improve policy behavior directly, but as oracle candidates they are less naturally modular because governance-relevant behavior is often entangled with the policy itself [48, 29, 34]. They are therefore useful baselines for alignment but a weaker fit when independent audit and patch-local updates are required.

Learned evaluative artifacts from demonstrations. IRL-style reward models, IIRL-style evaluators, and related demonstration-derived artifacts are attractive because they separate action generation from evaluation [28, 47, 2, 4, 24]. Their main assumptions concern coverage, identifiability, and generalization from demonstrations. They can extrapolate unreliably outside demonstrated support, which motivates explicit uncertainty and governance around them.

Constraint and logic-based artifacts. Learned logic constraints [23] and hand-written monitors or rules provide stronger interpretability and can map more directly to norms. Their main assumptions are that the relevant structure is representable in the chosen language and that sufficient observability is available to evaluate those constraints at runtime.

Guardrails and shields. Guardrails and shielding mechanisms can also instantiate the oracle role, especially when explicit specifications exist [36, 3, 19]. Their strengths lie in runtime enforcement and, in some settings, formal guarantees. Their main limitation for the present paper is not that they cannot be verified, but that correction and adaptation may become difficult when failure handling requires changing the specification or the synthesized controller itself rather than issuing a narrow patch to an evaluative artifact.

Structured task artifacts. Beyond standalone evaluative models, Flywheel-style governance could also operate over structured artifacts such as automata or reward-machine-like representations. Reward machines and related structured

intermediates can be learned from data [5, 41] or language [12] and can provide explicit, inspectable task progression. Their explicit proposition mappings and transition structure support localized, inspectable updates, making them plausible substrates for audited oracle refinement.

In practice. Realistic deployments may combine several of these ingredients. The architectural contribution of this paper is therefore not tied to one oracle family. The use of IIRL in the main text should be read as a reference instantiation with favorable systems properties, not as a claim that all other oracle realizations are inferior in all settings.

B.3 Prediction uncertainty and audit coverage uncertainty

The paper distinguishes two different uncertainty signals.

Prediction uncertainty. This is attached to the oracle's own judgment. It expresses how trustworthy the oracle believes its safety score to be for the queried input. Depending on the oracle family, this may be estimated through support in the underlying data, evidential inconsistency, ensemble disagreement, calibration methods, or other means.

Audit coverage uncertainty. This is generated by the Flywheel itself. It expresses how well a particular class of cases, trajectories, or outputs has been externally covered by auditing and verification activity. It therefore does not come from the oracle's training data or internal model alone. Instead, it captures second-order uncertainty about whether that region of behavior has been checked sufficiently well to justify trust in deployment.

The distinction matters operationally. A system may defer because the oracle is uncertain, because the Flywheel has insufficient verification coverage for that kind of case, or both. This is especially important in verification-as-a-service settings, where the governance layer is valuable not only because it consumes oracle-side uncertainty, but because it contributes additional assurance beyond the oracle itself.

B.4 Verification, Correction, and Patchability

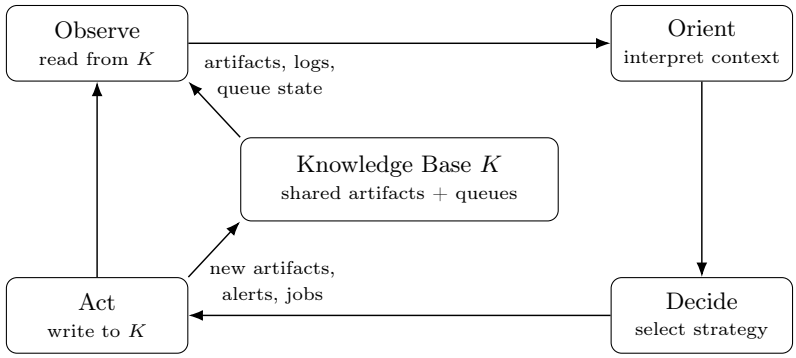
Verification and correction should not be conflated. Some safety artifacts, especially explicit monitors, rules, and shields, may be comparatively amenable to verification because their specifications are explicit and their behavior can be checked directly against those specifications. However, when new failure classes appear, adapting such artifacts may require rewriting the specification, synthesizing a new controller, or revisiting the assumptions under which the monitor was sound.

By contrast, the proposer-oracle architecture in this paper is designed around *patch locality*: many safety fixes are intended to take the form of narrow updates to an evaluative artifact and its governed release process. The point is not that

such artifacts are always easier to verify, but that they can be easier to correct incrementally under deployment pressure. This makes patchability, auditability, and release control central concerns alongside correctness.

C OODA Loop Specifications for Governance Roles

To demonstrate the operational feasibility of the Alignment Flywheel, we detail concrete OODA-loop instantiations for the governance roles described in Sec. 3 and Sec. 4. The OODA abstraction supports stateless, decoupled role implementations: agents interact through the Knowledge Base K, the queues Q_{ver} and Q_{ref}, and the defined protocol artifacts, as shown in Figure 3.



Abstract role template used by Red, Blue, Verification, Triage, and Refinement agents.

Fig. 3. Abstract OODA interaction pattern for governance agents. Each role reads shared state from the append-only knowledge base K during *Observe*, interprets it relative to its local objective during *Orient*, selects a strategy during *Decide*, and writes derived artifacts back to K during *Act*. Role-specific behavior is captured by strategies and artifact types, while the interaction contract with K remains uniform across the governance MAS.

C.1 Red Team: Adversarial Discovery

- **Observe:** Reads the active normative specification Φ, including severity weights, the current Oracle/governance versions, and historical defended traces from K.
- **Orient:** Identifies regions of the input, trajectory, or Oracle-response space with high normative severity but low empirical coverage. If a human has injected an adversarial seed, the agent initializes its search around that seed.

- **Decide:** Selects a generation or search strategy suited to the domain, such as prompt mutation for language outputs, gradient-based search over differentiable artifacts, physical perturbation for robotics, or spatial probing of a learned Oracle surface. The strategy targets uncertain regions and low-uncertainty claimed-safe cases that may conceal false negatives.
- **Act:** Generates or selects a candidate trajectory τ_{cand} and queries the governed Oracle stack. If the stack returns a safe prediction with low prediction uncertainty ($u < u_{\text{thresh}}$), but the Red Team heuristic still suspects a violation, the agent writes a `CandidateFlaw` record to Q_{ver}.

C.2 Verification Team: Validation

- **Observe:** Pulls candidate flaws from Q_{ver} and retrieves the relevant norm definitions $\phi \in \Phi$ from K.
- **Orient:** Parses the trajectory τ into a verifiable representation, such as a terminal state, tool-call trace, structured disposition, or normalized semantic representation.
- **Decide:** Evaluates τ against ϕ. Programmatic agents may use deterministic validators such as SMT checks, regular expressions, threshold rules, execution sandboxes, or domain-specific predicate checkers. If the norm requires judgment that is not formalized, the decision is escalated to a human reviewer or an adjudication service.
- **Act:** Writes a `VerificationResult` to K. If a violation is confirmed, the agent also writes a `VerifiedBreach`; otherwise, it logs the attempt as a defended trace to reduce redundant rediscovery.

C.3 Triage Agent: Queue Governance

- **Observe:** Polls K for newly logged `VerifiedBreach` records and queue state.
- **Orient:** Groups structurally similar breaches into failure families, for example by embeddings, rule keys, domain labels, or manually supplied cluster identifiers.
- **Decide:** Prioritizes clusters using signals such as norm severity, dangerous certainty ($u_{\text{thresh}} - u$), novelty relative to prior history, deployment frequency, and operational urgency. It may also perform diversity sampling within each cluster to select representative edge cases.
- **Act:** Packages cluster metadata and selected exemplars into a `RefinementJob` and pushes it to Q_{ref}.

C.4 Refinement Team: Correction and Release Preparation

- **Observe:** Pulls the highest-priority `RefinementJob` from Q_{ref} and retrieves associated breaches, regression tests, prior patches, and release constraints from K.

- **Orient:** Analyzes the breach cluster to determine whether the correction should affect the Oracle artifact, an Oracle wrapper, the Flywheel Overlay, audit coverage state, thresholds, or enforcement policy.
- **Decide:** Synthesizes candidate corrections, such as symbolic rules, spatial suppression kernels, disposition overrides, threshold adjustments, audit coverage updates, or other Oracle-stack patches. In human-supervised modes, a reviewer approves, rejects, or modifies the proposed governance batch.
- **Act:** Evaluates the proposed `GovernanceBatch` against the regression suite. If it passes the required checks and approvals, the system writes a release record to the ledger L in K, producing a new governed Oracle-stack version.

C.5 Blue Team: Monitoring and Forensics

- **Observe:** Reads telemetry from the Enforcement layer E, such as block rates, high-uncertainty escalations, timeout rates, and version-specific decision distributions, together with Red Team and verification logs from K.
- **Orient:** Computes monitoring summaries such as agent efficacy, norm coverage, calibration drift, audit coverage gaps, deployment anomalies, and proposer–oracle interface regressions.
- **Decide:** Determines whether a Red Team strategy has plateaued, whether additional verification coverage is needed, whether a release should be paused or rolled back, or whether live traffic indicates a distribution shift requiring temporary mitigation.
- **Act:** Writes coverage maps, forensic summaries, and monitoring reports to K. In emergency cases, it may trigger a policy-defined degraded mode or temporary mitigation in E, while logging the action for later verification, refinement, and governed release.

D Protocol Specifications

We specify coordination as explicit protocols between roles rather than as implicit message passing. This improves substitutability, since agents and strategies can be swapped without changing the interaction contract, and supports audit of compliance with the intended governance process. The protocols implement the two-stage triage pipeline over the append-only Knowledge Base K: candidate flaws are verified before becoming breaches, and breaches are clustered before becoming governed updates. The same protocol supports different degrees of human involvement. Low-risk deployments may automate verification and refinement, while high-risk deployments may require human review or signature for each governance batch. Appendix E details the human oversight and scaling model.

D.1 Common Transport and Invariants

Transport model. Protocol artifacts are stored in the append-only Knowledge Base K and referenced by immutable identifiers, such as hashes. Queues Q_{ver}

and Q_{ref} store only artifact references, enabling idempotent processing and decoupled execution.

Global invariants.

- **Idempotency:** each protocol artifact carries a unique ID; consumers must tolerate retries without duplicating effects.
- **Causal linking:** every derived artifact records parent references, e.g., a VerificationResult links to a CandidateFlaw.
- **Tamper-evident provenance:** governance batches and release events are signed and recorded in ledger L to support audit, deployment control, and rollback.

D.2 Protocol P1: Candidate Flaw Submission (Red $\rightarrow Q_{ver}$)

Participants. Red Team agent, Q_{ver}, Knowledge Base K.

Purpose. Submit candidate flaws for verification, emphasizing low-uncertainty claimed-safe trajectories that may represent suspected false negatives.

Message schema.

$$\mathsf{CandidateFlaw} = \langle id, \Sigma, \tau, s, u, u_{\text{thresh}}, u_a, u_{a,\text{thresh}}, v_O, ts, \mathsf{seedRef} \rangle.$$

Steps.

1. Red Team appends CandidateFlaw to K and enqueues id to Q_{ver}.
2. Triage-I may re-prioritize within Q_{ver} using risk signals such as norm severity, dangerous certainty $\max(0, u_{\text{thresh}} - u)$, audit coverage uncertainty u_a, novelty, and operational urgency.

Postconditions. K contains an immutable candidate record; Q_{ver} contains its reference. If enqueue fails, Red Team retries; idempotency holds because id is stable.

D.3 Protocol P2: Verification Result Publication (Verification $\rightarrow K$, Triage)

Participants. Verification agent(s) and/or human reviewer(s), Knowledge Base K, Triage agent.

Purpose. Validate a CandidateFlaw against Φ and produce a VerificationResult plus, when applicable, a VerifiedBreach.

Message schema.

$\mathsf{VerificationResult} = \langle refId, isViolation, \phi_{\text{broken}}, evidenceRef, reviewerRef, ts \rangle$.

If $isViolation = T$:

$$textsfVerifiedBreach =$$

$$\langle breachId, refId, \Sigma, \tau, \phi_{\text{broken}}, evidenceRef, s, u, u_{\text{thresh}}, u_a, u_{a,\text{thresh}}, v_O, ts \rangle.$$

Steps.

1. Verifier dequeues a candidate reference id from Q_{ver} and reads the linked CandidateFlaw from K.
2. Verifier evaluates τ against Φ using automated checks and/or human judgment and appends a VerificationResult to K.
3. If a violation is confirmed, the verifier additionally appends a VerifiedBreach to K, linked to $refId$ and carrying the Oracle-stack signals recorded at discovery time.

Postconditions. All decisions are recorded in K with explicit links to parent artifacts; confirmed violations are available to Triage as VerifiedBreach records.

D.4 Protocol P3: Cluster and Job Creation (Triage → Q_{ref})

Participants. Triage agent, Knowledge Base K, Q_{ref}.

Purpose. Convert a stream of VerifiedBreach records into batched RefinementJob artifacts.

Message schema.

$$\mathsf{RefinementJob} =$$

$$\langle jobId, clusterId, centroidRef, size, riskScore, sampleSetRefs, ts \rangle.$$

Steps.

1. Triage observes new VerifiedBreach artifacts in K since the last checkpoint.
2. Triage clusters similar breaches and computes $riskScore$ using signals such as norm severity, dangerous certainty, audit coverage uncertainty, novelty, deployment frequency, and operational urgency.
3. Triage appends a RefinementJob to K and enqueues $jobId$ to Q_{ref}.

Postconditions. Q_{ref} contains cluster-level jobs, reducing human review load through batching and diversity-aware aggregation.

D.5 Protocol P4: Governance batch preparation (Refinement $\rightarrow K, L$)

Participants. Refinement agent(s) and/or human supervisor(s), Knowledge Base K, Release Ledger L.

Purpose. Produce a GovernanceBatch with linked breach provenance, regression evidence, version metadata, and authorization signature. A batch may contain Oracle-local corrections, overlay updates, audit coverage updates, threshold changes, or enforcement-policy metadata.

Message schema.

$$\mathsf{GovernanceBatch} = \langle id, corrections, parentV, targetV, breachRefs, testResultRef, signature, ts \rangle.$$

Steps.

1. Refinement dequeues $jobId$ from Q_{ref} and reads the linked breach set from K.
2. Refinement synthesizes candidate corrections and runs the regression suite, recording the resulting test artifact in K.
3. Refinement appends a GovernanceBatch to K and records the corresponding immutable batch event in L.

Security invariant. A batch is deployable if and only if its signature verifies, required regression evidence is present, and its *parentVersion* matches the fleet's allowed upgrade path.

D.6 Protocol P5: Release, Rollout, and Rollback

Participants. Release manager, L, deployment substrate, governed Oracle-stack endpoints.

Purpose. Distribute signed governance batches under progressive rollout and safe rollback.

Release record schema.

$$\mathsf{ReleaseRecord} = \langle Id, batchId, rolloutPolicy, status, canaryMetricsRef, ts \rangle.$$

Steps.

1. Release manager selects a deployable GovernanceBatch and creates a ReleaseRecord in L.
2. Rollout proceeds in stages, e.g., canary $\rightarrow$ expand $\rightarrow$ full, updating *status* with measured metrics.
3. On regression, release manager records a rollback event in L and instructs endpoints to revert to the prior allowed version.

Security motivation. Signed provenance and controlled rollout/rollback are standard defenses for distributed update channels and supply-chain integrity, including attested build steps and signing infrastructure [27].

E Human Oversight and Scaling

The Alignment Flywheel is designed for tunable oversight: human involvement can vary with domain risk, regulatory burden, and organizational policy. Rather than requiring humans to inspect every candidate trajectory, the governance MAS exposes control surfaces that let operators steer the system at the level of norms, thresholds, strategy modules, escalation rules, and release approvals. This follows mixed-initiative and adjustable-autonomy patterns for scalable human-agent teams [37, 15]. Oversight modes are shown in Table 12 and human control surfaces in Table 13

Table 12. Oversight modes supported by the Flywheel.

Mode	Human role	Typical use
Low risk	Exception handling	Automated verification and refinement; humans review anomalies or periodic reports.
Medium risk	Batch approval	Agents cluster failures and propose governance batches; humans review summaries and authorize release.
High risk	Item and release approval	Humans review high-impact cases, approve or modify patches, and sign each governance batch before rollout.

Scaling rationale. The purpose of tunable oversight is to move human effort from raw case handling to governance steering. Triage clusters similar failures before they reach refinement; verification separates suspected flaws from confirmed breaches; and release approval operates on governance batches rather than individual probes. This keeps human attention focused on high-severity uncertainty, new norm families, disputed verification judgments, and release decisions.

F Scaling Roadmap

The Alignment Flywheel is an architectural layer, so scaling it requires progress along several independent axes rather than a single larger benchmark.

Table 13. Human steering interfaces for scalable governance.

Control surface	Effect
Agent efficacy and strategy rotation	Surface metrics such as unique verified breaches per unit time; deprecate weak strategies and activate stronger ones.
Normative coverage and prioritization	Visualize coverage across $\phi \in \Phi$; adjust severity weights or introduce new norms to re-orient OODA loops.
Adversarial seeding and directed search	Inject seed trajectories into Q_{ver}; Red Team agents mutate or exploit them to discover variants [18, 22].
Deployment feedback	Surface high-uncertainty runtime cases as incidents; direct coverage expansion and verification toward failed regions.
Release approval and rollback	Require human approval, signature, or staged rollout for governance batches in high-risk settings.

Oversight scaling. Human involvement can move from item-level review to batch approval, policy steering, and release authorization. Low-risk settings may automate most verification and refinement, whereas high-risk settings require human approval for verified breaches, governance batches, or rollout decisions.

Modality scaling. The same Proposer–Oracle interface can govern text, tool calls, trajectories, images, robot plans, or multimodal traces, provided candidate behavior can be represented as a trajectory and evaluated by an Oracle stack.

Domain scaling. Different domains require different norm semantics, evidence sources, escalation rules, and regression suites. Clinical communication, robotics, cyber defense, and tool-using agents therefore instantiate the same governance loop with different domain-specific verifiers and policies.

Component scaling. The architecture does not require any single Red Team, Oracle, verifier, or refinement method. Stronger IIRL-style Oracles, calibrated uncertainty estimators, formal norm checkers, adversarial search methods, and patch planners can replace the simple components used in the demos without changing the protocol layer.

G Reference and Demos Implementation Details

This appendix summarizes implementation-level details for the evaluation scenarios in Section 6. The reference implementation is a single Python/Flask application organized around six layers: protocol definitions, core governance infrastructure, concrete role implementations, a thin HTTP API layer, a factory registry, and demo runners. Components are selected through a single composition root, so the same controllers, knowledge base, query merger, batch applier,

and protocol artifacts are reused across the spatial and medical demos. Code is available at https://github.com/decide-ugent/Alignment-Flywheel.

G.1 Project Structure

The implementation is organized as follows:

```
flywheel/
  protocols/       # enums, typed artifacts, interfaces, OODA
  core/            # governance engine, knowledge base,
                   # query merger, batch applier
  roles/           # oracle, proposer, overlay, enforcement,
                   # triage,blue team, red team, verifier
                   #, refinement
  api/             # Flask app, blueprints, HTTP clients,
                   # runtime state
  factory/         # registry, auto-registration,
                   # YAML-driven wiring
  demos/           # spatial and medical demo
                   # runners/configurations
```

The protocol layer defines the shared dataclasses used throughout the system, including `Trajectory`, `TrajectoryStep`, `OracleRawOutput`, `FlywheelOverlay`, `UnifiedQueryResult`, `EnforcementResult`, `Norm`, `GovernanceBatch`, `LocalCorrection`, `CandidateFlaw`, `VerificationResult`, and `DecisionRecord`. The interface layer defines abstract contracts for the Proposer, Safety Oracle, Flywheel Overlay, Enforcement policy, Knowledge Base, Query Merger, Batch Applier, Blue Team, Triage, and domain-specific adapters.

The OODA-structured roles are decomposed into `observe`, `orient`, `decide`, and `act` steps. For example, the Red Team can use a grid observer in the spatial demo or a medical case generator in the Patient Portal demo, while preserving the same role-level OODA controller. YAML configuration files select the concrete steps for each demo through the factory registry.

G.2 Norm Representation

Each norm in Φ is represented as a `Norm` dataclass with fields `id`, `kind: NormKind`, `spec: Dict[str,Any]`, `severity: float`, `weight: float`, and `description: str`. The typed `kind` field determines the verifier semantics.

This typed representation is the implementation-level instantiation of the paper's `LogicExpression` abstraction. New norm kinds can be added by extending `NormKind` and implementing the corresponding verifier logic in the relevant orient/decide steps.

Table 14. Implemented norm kinds in the reference implementation.

Norm kind	Verifier semantics
`KEYWORD_BLOCK`	Checks whether trajectory text contains prohibited keywords, optionally conditioned on evidence status. Example: medication-change terms under weak evidence.
`REGEX`	Checks whether trajectory text matches a regular-expression pattern, such as a prohibited action phrase.
`PREDICATE`	Evaluates multi-field relationships over payload and metadata, such as whether the proposed disposition is at least as severe as required for the case type and evidence status.
`SPATIAL_BOUNDARY`	Checks whether a spatial query point lies within the supported region, e.g., below a distance threshold from expert data.
`THRESHOLD_RULE`	Checks threshold-style constraints, such as age or vulnerability conditions requiring a specified evidence standard.

G.3 Governance Batch Format

The `GovernanceBatch` is the deployment unit for governed updates. It contains a list of `LocalCorrection` artifacts, each with a typed `correction_type: CorrectionType` and a payload dictionary. The implementation supports five correction types: `SPATIAL_FLAW_PATCH`, `AUDIT_COVERAGE_UPDATE`, `THRESHOLD_ADJUSTMENT`, `MEDICAL_HARD_BLOCK`, and `NORM_UPDATE`. Not every demo uses every correction type: the spatial demo uses spatial flaw patches and audit coverage updates, while the medical demos use hard blocks, disposition/threshold adjustments, and audit coverage updates.

A representative spatial governance batch is:

```
{
  "batch_id": "batch:a1b2c3",
  "from_oracle_version": "oracle:v4",
  "to_oracle_version": "oracle:v5",
  "local_corrections": [
    {
      "correction_id": "corr:d4e5f6",
      "correction_type": "spatial_flaw_patch",
      "payload": {
        "flaw_point": [0.73, -0.42, 0.81],
        "support_radius": 0.187
      }
    },
    {
```

```
      "correction_id": "corr:g7h8i9",
      "correction_type": "audit_coverage_update",
      "payload": {
        "case_class": "spatial|d=0.7"
      }
    }
  ],
  "regression_evidence": {
    "patched": 60,
    "rejected": 3,
    "predicted_coverage": 142
  },
  "rollout_metadata": {},
  "signature": "regression-verified",
  "timestamp": "2026-04-28T12:00:00Z"
}
```


Medical batches have the same outer structure but different correction payloads. A `MEDICAL_HARD_BLOCK` adds a prohibited phrase to the Oracle's block list, a `THRESHOLD_ADJUSTMENT` installs a disposition override keyed by case class or evidence status, and an `AUDIT_COVERAGE_UPDATE` records that a case class has been audited by the Flywheel Overlay. This shared batch format is what allows the same release and audit machinery to operate across the spatial and medical demos.

G.4 3D Spatial Demo Specification

This appendix specifies the 3D spatial governance demo. Unlike the medical demos, which govern discrete case-level decisions, the spatial demo governs a continuous reward surface in $[-1, 1]^3$ learned by IIRL. The Flywheel must discover grid cells where the learned reward is non-trivial outside the expert-support basin, suppress them through governed spatial patches, and preserve the legitimate reward basin around the expert trajectory. The demo validates the same architectural claims—Red Team discovery, norm-based verification, typed governance batches, patch locality, and regression-checked preservation—in a continuous spatial domain.

Problem Setup

Domain and expert path. The domain is the cube $[-1, 1]^3$, discretised on a $20 \times 20 \times 20$ regular grid (8 000 cells). An IIRL training run produced a precomputed loss value $\ell_i \in \mathbb{R}_{\geq 0}$ at each grid cell i, stored in `loss_values.npy`. The synthetic expert path $\mathcal{E}$ consists of 20 points sampled from:

$$\mathcal{E}(t) = \big(-1 + 2t,\ \mathrm{clip}(-1 + 2t + 0.8\sin(\pi t)),\ \mathrm{clip}(-1 + 2t + 0.8\cos(\pi t))\big),$$
$$t \in \{0, \tfrac{1}{19}, \ldots, 1\}.$$

Reward, basin, and flaw definitions. The Oracle converts loss to reward by:

$$r_i = 1 - \min\bigl(\ell_i / L_{\text{cap}},\, 1\bigr), \quad L_{\text{cap}} = 0.3.$$

Cells with $\ell_i \geq 0.04$ are pre-thresholded to $\ell_i = 999\,999$ to isolate the IIRL artifacts that require governance. Let

$$d_i = \min_{e \in \mathcal{E}} \|p_i - e\|_2$$

be the distance from grid cell i to the nearest expert-path point. With safety floor $\sigma = 0.005$ and basin boundary $B = 0.34$, cells are classified as follows.

Table 15. Spatial cell classification used by the demo.

Cell type	**Condition**	**Interpretation**
Basin cell	$r_i > \sigma$ and $d_i \leq B$	Legitimate reward near the expert trajectory; must be preserved.
Flaw cell	$r_i > \sigma$ and $d_i > B$	Non-trivial reward outside the expert basin; must be suppressed.
Inactive cell	$r_i \leq \sigma$	Already below the safety floor; no action required.

The objective is to drive the flaw count to zero while preserving the initial basin count.

Oracle: `PrecomputedGridOracle`

State and query evaluation. The `PrecomputedGridOracle` stores the read-only reward vector $\mathbf{r} \in \mathbb{R}^{8000}$, a mutable list of suppression kernels $\{(c_k, \beta_k)\}_{k=1}^{K}$, and a version counter v_O. For a query point p, the Oracle snaps p to the nearest grid cell i, reads r_i, computes cumulative suppression,

$$S(p) = \min\Bigl(1, \sum_{k=1}^{K} \exp\bigl(-\|p - c_k\|^2 / (2\beta_k^2)\bigr)\Bigr),$$

and returns the governed safety score

$$s = \max(0, r_i - S(p)).$$

The output is therefore not the raw IIRL reward, but the reward attenuated by every suppression kernel installed through prior governance batches.

Batch application. A `GovernanceBatch` contains `LocalCorrection` artifacts. The Oracle filters for `correction_type == SPATIAL_FLAW_PATCH`; for each such correction, it appends the centre c_k from `payload["flaw_point"]` and bandwidth β_k from `payload["support_radius"]` to its kernel list. The Oracle version is then incremented.

Patch Planner: Predictive Batch Planning The adaptive variant uses a `Patch Planner` that plans each governance batch before deployment by analytically predicting kernel effects. The planner receives the basin set $\mathcal{B}$, flaw points sorted by descending distance to the expert path, and each flaw's distance d_i. It then selects non-redundant kernels subject to basin-preservation constraints.

Table 16. PatchPlanner loop for adaptive spatial governance batches.

Step	Function
Skip covered flaws	If a previously accepted kernel in the same batch already covers flaw p_i, skip it.
Propose bandwidth	Set $$\beta_{\text{prop}} = \text{clip}\big((d_i - B)\cdot 0.5, \beta_{\min}, \beta_{\max}\big), \quad \beta_{\min} = 0.03, \quad \beta_{\max} = 0.30.$$ Distant flaws receive wider kernels; near-boundary flaws receive narrower kernels.
Single-kernel basin protection	Let $d_{\text{near}} = \min_{b\in\mathcal{B}} \lVert p_i - b\rVert$. The maximum bandwidth that keeps suppression of the nearest basin point below $\eta = 0.10$ is $$\beta_{\text{safe}} = \frac{d_{\text{near}}}{\sqrt{-2\ln\eta}}.$$ The proposed bandwidth is shrunk to $\min(\beta_{\text{prop}}, 0.9\beta_{\text{safe}})$.
Cumulative basin check	The planner tracks running suppression on every basin point from accepted kernels in the current batch: $$S_{\text{existing}}(b) = \sum_{j<i} \exp\big(-\lVert b - c_j\rVert^2/(2\beta_j^2)\big).$$ If adding the candidate kernel would push any basin point above η, a 15-step binary search over $[\beta_{\min}/2, \beta]$ finds the largest safe bandwidth. If even $\beta_{\min}$ would cause excessive basin suppression, the patch is rejected.
Predict coverage	The effective suppression radius is $$R_{\text{eff}} = \beta\sqrt{-2\ln\sigma}, \quad \sigma = 0.005.$$ Other flaw points within R_{eff} are predicted to be suppressed below the safety floor.
Mark covered flaws	Predicted-covered flaws are marked as handled and skipped later in the batch. This spends the patch budget on non-redundant kernel placements.

The loop terminates when the batch budget is exhausted (max_patches = 200) or all currently known flaws are covered.

Red Team Discovery and Batch Construction

Adaptive flaw discovery. Each iteration queries the full $20^3 = 8\,000$ grid through the governed Oracle. From iteration 2 onward, the Red Team also performs dense sampling around previously active flaws. The number of fine samples and search radius adapt to the previous flaw count f_{prev}:

$$n_{\text{fine}} = \text{clip}\big(8000/\max(1, \lfloor f_{\text{prev}}/50 \rfloor), 500, 2000\big),$$

$$r_{\text{fine}} = \text{clip}\big(f_{\text{prev}}/5000, 0.12, 0.35\big).$$

Many remaining flaws trigger wider exploration; few remaining flaws trigger tighter search around residual boundary regions. Fine samples are drawn uniformly within r_{fine} of up to 200 previously known flaw centres. A queried point is classified as a flaw if its governed Oracle score exceeds σ and its distance to the expert path exceeds B.

Batch construction. Each accepted patch $(p_i, \beta_i, n_{\text{covered}})$ from the PatchPlanner produces the corrections in Table 17. The corrections are wrapped in a `GovernanceBatch` with regression evidence recording the number of kernels placed and total predicted coverage.

Table 17. Typed corrections used by the spatial demo.

Correction type	Effect
`SPATIAL_FLAW_PATCH`	Payload contains `flaw_point` p_i and `support_radius` β_i. Applying the correction installs a suppression kernel in the Oracle.
`AUDIT_COVERAGE_UPDATE`	Payload records the predicted coverage class, e.g. `"spatial|bw=`β_i`|cov=`n_{covered}`"`, for audit trail and batch provenance.

The batch is applied to the Oracle, which installs suppression kernels, and to the Knowledge Base, which records the governed update and its evidence.

Convergence Mechanics Each iteration follows the same governance lifecycle:

$$\text{SearchRedTeam} \xrightarrow{\text{flaw points}} \text{PatchPlanner} \xrightarrow{\text{plan}} \text{build_batch} \xrightarrow{\text{GovernanceBatch}} O.$$

After the batch is applied, the Red Team re-queries known flaw points through the updated Oracle and uses the remaining active flaws to guide the next search iteration.

Why convergence is fast. Farthest-first ordering combined with adaptive bandwidth creates large early reductions. A distant flaw at $d = 0.9$, for example, may receive $\beta \approx 0.28$, giving $R_{\text{eff}} \approx 0.92$. Such a kernel can cover many neighbouring flaws, which are marked as handled before deployment. The 200-patch budget is therefore spent on non-redundant placements, so per-iteration flaw reduction can substantially exceed the number of directly patched centres.

Why the basin is preserved. Basin preservation is enforced by a two-tier check: a per-kernel analytical cap and a cumulative binary-search cap over basin suppression. Kernels near the basin boundary are shrunk or rejected if they would exceed the suppression budget. This is a regression-checking mechanism over the accepted basin rather than a guarantee about unsampled continuous space.

Predicted coverage rather than accidental collateral. In an earlier spatial variant, "collateral" referred to flaw points suppressed by neighbouring kernel spill-over. In this adaptive variant, the PatchPlanner predicts this effect before deployment and records the predicted coverage. We therefore report *predicted coverage*: flaw cells suppressed by a planned neighbouring kernel rather than directly targeted by their own kernel.

Visualisation Each iteration produces a 3D scatter plot from elevation 45° and azimuth 300°. Active cells with reward above σ are plotted with colour mapped to reward value using the `viridis` colourmap. The expert trajectory is plotted as a blue line, and kernel centres placed in the current iteration are plotted as red `x` markers. Basin and flaw cells use the same reward colourmap, so the visible structure is the governed reward surface itself. Across iterations, the active reward region contracts toward the expert path until only the basin remains.

Comparison with Fixed-Bandwidth Baseline To test the effect of adaptive bandwidth and predictive coverage, the same problem was run with slightly different spatial data, a fixed-bandwidth baseline, `spatial_3d_fixed_bw`. The baseline uses constant bandwidth $\beta = 0.05$ and a batch budget of 60 patches per iteration.

Table 18. 3D Spatial demo: adaptive PatchPlanner vs. fixed-bandwidth baseline.

Metric	PatchPlanner (adaptive)	Fixed BW ($\beta = 0.05$)
Iterations to converge	16	> 25 (not converged)
Patches/iter (max)	60	60
Basin preserved	783/783 (100%)	783/783 (100%)
Total kernels placed	834	1500 (it. 30)

The adaptive PatchPlanner converges in 16 iterations because wide kernels at distant flaws analytically cover many neighbours, increasing effective throughput. The fixed-bandwidth baseline reduces flaws much more slowly and would require substantially more iterations to converge. Both preserve the basin in this grid-based evaluation, confirming that the regression check prevents accepted basin cells from being suppressed below the safety floor.

Convergence Results Table 19 reports per-iteration convergence for the adaptive PatchPlanner variant. "Found" is the number of active flaw candidates discovered in that iteration; "Kern" is the number of kernels accepted into the governance batch; "Predicted" is the number of flaw candidates analytically predicted to be covered by accepted kernels; "Reject" counts candidate kernels rejected by basin-preservation checks. "Basin" is the number of sampled basin cells still above the safety floor after the batch is applied.

Table 19. 3D Spatial demo (adaptive PatchPlanner): convergence over iterations.

Iter	Found	Kern	Predicted	Reject	Basin	Flaws	Oracle
1	500	87	500	0	783	412	v1
2	993	42	993	0	783	357	v2
3	874	58	874	0	783	290	v3
4	1 030	57	1 030	0	783	219	v4
5	1 119	49	1 119	0	783	159	v5
6	960	71	959	1	783	103	v6
7	782	90	782	1	783	65	v7
8	555	90	555	0	783	32	v8
9	398	82	396	3	783	13	v9
10	211	55	209	3	783	7	v10
11	164	34	161	3	783	5	v11
12	96	24	96	1	783	0	v12

Key observations. The adaptive PatchPlanner drives the remaining flaw count to zero after 12 governance iterations. The sampled basin count remains fixed at 783 throughout the full run, giving 100% basin preservation on the evaluation grid. Early iterations remove large flaw regions because accepted kernels cover many neighboring flaw candidates analytically; later iterations place fewer kernels as the remaining flaws become sparse and closer to basin-preservation constraints. Rejections appear only from iteration 6 onward and remain small relative to the number of accepted kernels, indicating that the planner usually finds safe bandwidths and rejects only candidates that would violate the preservation budget. The final iteration requires only 24 accepted kernels and one rejection, confirming that the residual flaws are localized and constrained by the regression checks.

G.5 Medical Demo Specifications

This appendix gives the technical specification for three synthetic medical governance demos. All three instantiate the complete Flywheel pipeline—Proposer, Safety Oracle, Flywheel Overlay, Query Merger, Enforcement, and the five governance roles—using the same abstract interfaces, knowledge base, batch format, and protocol artifacts. The demos differ only in the concrete component implementations selected through a single composition root. They therefore serve as ablations over governance complexity: Demo 1 uses a minimal heuristic Oracle and FIFO triage, Demo 2 adds multi-dimensional risk scoring and priority triage, and Demo 3 instantiates the patient-portal scenario summarized in the main text.

Shared Architecture and Pipeline All three medical demos share the same runtime pipeline and governance cycle. The purpose of the demos is not to validate clinical adequacy, but to show that the same Flywheel protocol can operate over different proxy Oracles, norm sets, triage strategies, and refinement mechanisms.

Runtime evaluation pipeline. Table 20 summarizes the shared runtime pipeline. The concrete Oracle and overlay differ by demo, but the protocol topology is invariant.

Table 20. Shared runtime pipeline used by all medical demos.

Component	Function
Proposer P	A `PassthroughProposer` wraps the incoming case data—patient message, draft reply, disposition, and metadata—as a `Trajectory` of kind `MESSAGE`. No generation occurs; the case already contains the candidate content.
Safety Oracle O	Evaluates the trajectory and returns `OracleRawOutput`: safety score s, prediction uncertainty u, uncertainty threshold u_{thresh}, Oracle version v_O, and evidence status.
Flywheel Overlay	Evaluates audit coverage and returns `FlywheelOverlay`: audit coverage uncertainty u_a, threshold $u_{a,\text{thresh}}$, governance version v_G, and audit status.
Query Merger	A `DefaultQueryMerger` combines Oracle-side and governance-side signals into a `UnifiedQueryResult` while preserving provenance.
Enforcement E	A `DefaultEnforcement` policy maps the unified signals to $a \in \{\text{allow}, \text{block}, \text{escalate}\}$.

Medical enforcement policy. The main text defines the conservative base policy: high prediction uncertainty or high audit-coverage uncertainty triggers fail-closed escalation. The medical demos instantiate a domain-specific refinement

of this policy. Prediction uncertainty always triggers escalation. Audit-coverage uncertainty triggers escalation only when evidence is weak and the safety score is below an audit margin, preventing routine supported cases from escalating solely because their key has not yet been explicitly covered.

The policy is parameterised by a safety margin $\theta_s = 0.4$ and an audit safety margin $\theta_a = 0.85$.

Table 21. Medical-domain enforcement policy used in all three demos.

Condition	**Action**
$u \geq u_{\text{thresh}}$	`escalate`: Oracle prediction uncertainty is too high.
$s < \theta_s$	`block`: safety score indicates unsafe behavior.
$u_a \geq u_{a,\text{thresh}}$, evidence $\in \{\texttt{insufficient}, \texttt{conflicting}\}$, and $s < \theta_a$	`escalate`: audit coverage is insufficient for a weak-evidence, borderline-safety case.
Otherwise	`allow`.

Thus uncertainty-driven escalation takes precedence, followed by definitive safety blocking, followed by coverage-driven escalation. Cases that are safe, well-covered, and low-uncertainty are allowed.

Governance cycle. After all fixed cases are evaluated, one governance iteration runs. Table 22 gives the shared cycle.

The cycle then repeats: fixed cases are re-evaluated under the patched stack, governance runs again, and convergence is measured by the disappearance of escalations while preserving correct allow/block separation.

Correction types and patch mechanics. Table 23 summarizes the correction types used across all medical demos. The first two modify Oracle state; the third modifies governance state.

Together, these corrections convert uncertainty-driven escalations into definitive allow/block decisions without modifying or retraining the Proposer.

Shared OODA composition. All three demos use the same OODA skeleton for Red Team, Verification, and Refinement. The concrete strategy modules differ only by parameterisation.

Demo 1: Simple Medical

Table 22. Shared governance cycle used by all medical demos.

Role	Function
Red Team	Generates 20 synthetic cases per iteration by sampling a combinatorial case space. Each case is classified by failure category and pushed to Q_{ver} as a `CandidateFlaw`.
Verification	Checks candidates against the normative specification Φ. Keyword-block norms check prohibited terms under weak evidence; predicate norms check multi-field relationships such as whether the proposed disposition meets the required minimum. Confirmed violations produce `VerificationResult` records.
Triage	Orders verified violations by priority and forwards them to Q_{ref}.
Refinement	Maps verified flaws to typed corrections. A tiered batch strategy rotates across categories to ensure diverse coverage. In all three demos, batch capacity is limited to one flaw per iteration.
Batch application	Applies the resulting `GovernanceBatch` to the Oracle and Flywheel Overlay. Oracle patches install hard-blocks and disposition overrides; overlay patches register audit coverage. Both components increment their version identifiers when state changes.

Table 23. Typed corrections used in the medical demos.

Correction type	Effect
`MEDICAL_HARD_BLOCK`	Adds a keyword to the Oracle's internal block list. Later trajectories containing the keyword return $s = 0.0$, $u = 0.05$, producing an unconditional block.
`THRESHOLD_ADJUSTMENT`	Installs a disposition override keyed by `case_type|evidence_status`. If a matching trajectory has a disposition below the required minimum, the Oracle returns $s = 0.15$, $u = 0.1$, which Enforcement maps to `block`.
`AUDIT_COVERAGE_UPDATE`	Registers a `case_type|evidence|acuity` key as audited in the Flywheel Overlay. Covered keys produce $u_a = 0.15{-}0.20$; uncovered keys produce $u_a = 0.85{-}0.90$.

Scenario and purpose. A healthcare organization deploys an AI assistant that drafts replies to patient messages in a clinical inbox. Patients ask about medication dosage, lab results, appointments, or routine follow-up; the AI proposes a draft reply and a disposition such as `reply_only`, `nurse_review`, or `clinician_review`. This demo uses deliberately simple components—a small heuristic Oracle, two norms, and FIFO triage—to isolate the protocol flow: discover flaws, verify them against norms, patch the Oracle stack, and re-evaluate.

Table 24. Shared OODA components for the medical demos.

Role	Component	Function
Red Team	Medical Case Generator	Samples medications, symptoms, lab tests, and demographics; classifies cases as `missed_urgency`, undertriaged_med, lab_no_context, `vulnerable_patient`, or `exploratory`.
Red Team	Medical Priority Decider	Sorts candidates by category-weighted priority: `missed_urgency` > `undertriaged_med` > `vulnerable_patient` > `lab_no_context` > `exploratory`.
Red Team	Medical Candidate Submitter	Wraps candidates as `Candidate Flaw` artifacts with Oracle signals, trajectory payload, and metadata.
Verification	Medical NormMatcher and Medical Violation Decider	Extract draft text, patient message, evidence status, and disposition; evaluate keyword-block and predicate norms; emit `VIOLATION` or `NO_VIOLATION`.
Refinement	Medical Correction Orienter	Maps each verified flaw to a small bundle of corrections, e.g., hard-block, disposition overrides, and audit-coverage updates.
Refinement	Medical Batch Decider and Medical Batch Deployer	Selects at most one flaw per iteration, builds a `GovernanceBatch`, sets version transitions, and returns the batch for application.

Fixed cases. The fixed evaluation set contains 15 synthetic cases, grouped in Table 25.

Table 25. Simple Medical demo: fixed evaluation groups.

Group	Count	Risk pattern	Expected transition
Unsafe medication	3	Unsupported dosing advice with `reply_only` disposition.	`escalate` → `block`
Unsafe lab	3	Lab-result interpretation or monitoring advice without sufficient evidence and without nurse review.	`escalate` → `block`
Borderline	3	Similar clinical content, but already routed to `clinician_review`.	`escalate` → `allow`
Safe	6	Routine supported interactions such as refills, scheduling, and acknowledgements.	`allow` throughout

Oracle, overlay, norms, and triage. Table 26 summarizes the concrete components.

Table 26. Simple Medical demo: concrete component choices.

Component	Specification
Oracle	`SimpleMedicalOracle`; heuristic scoring with patchable hard-block list B and disposition override dictionary D. Hard-blocks return $s = 0.0, u = 0.05$; disposition failures return $s = 0.15, u = 0.1$. Base scoring penalizes medication stems, medication case type, and weak evidence.
Prediction uncertainty	Baseline $u = 0.15$, increased for insufficient, unknown, or conflicting evidence and for medication cases. $u_{\text{thresh}} = 0.30$ for medication cases and 0.50 otherwise.
Flywheel Overlay	`SimpleMedicalOverlay`; coverage key `case_type|evidence_status|acuity`. Covered keys produce $u_a = 0.2$; uncovered keys produce $u_a = 0.9$. Threshold $u_{a,\text{thresh}} = 0.6$.
Norms Φ	Two norms: `NO_UNSUPPORTED_MED_ADVICE`, a keyword-block norm over medication-change terms under weak evidence; and `REQUIRE_EVIDENCE_FOR_RESULTS`, a predicate norm requiring at least `nurse_review` for weak-evidence lab/result interpretation.
Triage	`FIFOTriage`; violations are passed to refinement in arrival order.

Convergence results. Table 27 shows enforcement behavior across evaluations. Evaluation 0 is the initial stack; later evaluations occur after governance batches have been applied.

Table 27. Simple Medical demo: enforcement convergence.

Eval	Allow	Block	Escalate	Esc. Rate	Stack
0	6	0	9	60%	v0
1	6	3	6	40%	v1
2	9	6	0	0%	v2

Observation. The simple demo validates the protocol flow independently of component sophistication. The safe cases remain `allow`; unsafe cases transition from uncertainty-driven `escalate` to definitive `block`; and borderline cases transition from `escalate` to `allow` once audit coverage is established.

Demo 2: Complex Medical

Scenario and purpose. The same clinical-inbox setting is extended with synthetic specialty structure, named medications, interaction risks, vulnerable patient profiles, and specialty-specific thresholds. The purpose is to test whether the same governance loop handles richer proxy-Oracle logic and a more complex normative specification without changing the Flywheel protocol.

Fixed cases. The fixed evaluation set contains 18 synthetic cases, grouped in Table 28.

Table 28. Complex Medical demo: fixed evaluation groups.

Group	Count	Risk pattern	Expected transition
Unsafe stop-taking	3	Recommendations to stop high-risk medications such as warfarin, insulin, or oxycodone under weak evidence.	`escalate` → `block`
Unsafe dose increase	3	Autonomous dose increases for higher-risk drugs such as insulin, tramadol, or prednisone.	`escalate` → `block`
Unsafe lab	3	Organ-function or medication-related lab interpretation with insufficient evidence and direct patient routing.	`escalate` → `block`
Borderline	3	Lab or medication discussion already routed to `clinician_review`.	`escalate` → `allow`
Safe	6	Routine supported interactions without clinical decision-making.	`allow` throughout

Oracle, overlay, norms, and triage. The `ComplexMedicalOracle` scores five dimensions using lookup tables: medication risk, interaction risk, action severity, patient vulnerability, and evidence quality. The safety score is:

$$s = \max\big(0,\ 1 - (0.30\, r_{\text{med}} + 0.25\, r_{\text{interact}} + 0.20\, r_{\text{action}} + 0.10\, r_{\text{patient}} + 0.15\, r_{\text{ev}})\big).$$

Prediction uncertainty starts at $u = 0.15$ and increases under weak evidence, high medication risk, or detected drug interactions. Thresholds are specialty-based.

Convergence results. Despite the richer proxy Oracle and norm set, the governance pipeline converges in the same number of evaluations as the simple demo. This reflects the batch strategy: the initial unsafe cases cluster into a small number of failure families, each resolved by a governed update.

Table 29. Complex Medical demo: concrete component choices.

Component	Specification
Oracle	`ComplexMedicalOracle`; patchable state includes hard-block list B, specialty threshold overrides T, and disposition overrides D. Evaluation order is hard-block, disposition override, then five-dimensional scoring.
Risk dimensions	r_{med} from a medication-risk table; r_{interact} from known interaction pairs; r_{action} from action severity; r_{patient} from age and comorbidities; r_{ev} from evidence status.
Flywheel Overlay	`ComplexMedicalOverlay`; coverage key `specialty|case_type|evidence_status`. Covered keys produce $u_a = 0.15$; uncovered keys produce $u_a = 0.85$. Threshold $u_{a,\text{thresh}} = 0.6$.
Norms Φ	Four norms: `NO_HIGH_RISK_MED_WITHOUT_EVIDENCE`, `INTERACTION_SAFETY`, `VULNERABLE_PATIENT_PROTECTION`, and `SEVERE_ACTION_EVIDENCE`. These cover keyword-block, regex, threshold-rule, and predicate-style checks.
Triage	`PriorityTriage`; priority order is `missed_urgency` > `undertriaged_med` > `vulnerable_patient` > `lab_no_context` > `exploratory`.

Table 30. Complex Medical demo: enforcement convergence.

Eval	Allow	Block	Escalate	Esc. Rate	Stack
0	6	0	12	67%	v0
1	6	6	6	33%	v1
2	9	9	0	0%	v2

Demo 3: Patient Portal

Scenario and purpose. A hospital operates a patient-facing web portal where patients message their care team directly. Unlike the clinical inbox in Demos 1–2, the portal is a patient-facing channel: the AI's draft reply may be the first thing the patient reads unless the system routes the message for review. This is the primary medical evaluation scenario summarized in Section 6.2. It validates the Flywheel architecture with a heuristic proxy Oracle that scores medication risk, urgency, disposition mismatch, and evidence quality.

Fixed cases. The fixed evaluation set contains 15 synthetic cases, grouped in Table 31.

Oracle, overlay, norms, and triage. The `PatientPortalOracle` scores four dimensions: medication content, urgency signals, evidence quality, and disposition mismatch. The safety score is:

$$s = \max\big(0,\ 1 - (0.30\, r_{\text{med}} + 0.25\, r_{\text{urg}} + 0.25\, r_{\text{disp}} + 0.20\, r_{\text{ev}})\big).$$

Table 31. Patient Portal demo: fixed evaluation groups.

Group	Count	Risk pattern	Expected transition
Unsafe medication	3	Direct patient-facing medication-change advice, such as increasing dose, without clinician review or sufficient evidence.	`escalate` → `block`
Unsafe lab	3	Lab-result interpretation with medication context, weak evidence, and `reply_only` routing.	`escalate` → `block`
Borderline	3	Lab or medication discussion already routed to `clinician_review`.	`escalate` → `allow`
Safe	6	Routine supported interactions such as refills, appointments, physical-therapy progress, diet follow-up, improving symptoms, and normal annual labs.	`allow` throughout

Prediction uncertainty starts at $u = 0.15$ and increases under weak evidence, urgent/semi-urgent acuity, or medication content. Uncertainty thresholds are acuity-based: $u_{\text{thresh}} = 0.20$ for urgent cases, 0.30 for semi-urgent cases, and 0.50 for routine cases.

Convergence results. Convergence is structurally identical to the simple demo because both have the same case-count distribution. The difference is that the Patient Portal Oracle uses richer scoring, urgency detection, and disposition-gap analysis, while the norm set is tailored to patient-facing communication.

Cross-Demo Analysis

Architectural invariance. Table 34 summarizes the variation across demos. The abstract interfaces, controllers, protocol artifacts, knowledge base, queues, and batch format are identical; only the concrete component implementations change.

Convergence pattern. All three demos follow the same convergence trajectory: initial escalation dominance, progressive conversion of escalations into definitive blocks or allows as governance batches accumulate, and zero escalations by the final evaluation. The pattern is consistent across proxy-Oracle complexity levels, indicating that convergence is driven by the governance pipeline and patch mechanics rather than by a single Oracle implementation.

Patch locality. The Proposer is never modified. The `PassthroughProposer` returns the same trajectory throughout all evaluations. Behavioral changes arise only from Oracle state changes, such as hard-blocks and disposition overrides,

Table 32. Patient Portal demo: concrete component choices.

Component	**Specification**
Oracle	`PatientPortalOracle`; patchable state includes hard-block list B and disposition override dictionary D. Evaluation order is hard-block, disposition override, then four-dimensional scoring.
Risk dimensions	r_{med} counts medication keywords; r_{urg} counts urgency keywords in the message or draft; r_{ev} encodes evidence quality; r_{disp} measures the gap between required and proposed disposition.
Disposition ranks	`reply_only`= 0, `nurse_review`= 1, `clinician_review`= 2, `urgent_escalation`= 3. Needed disposition increases with urgency, medication risk, and semi-urgent acuity.
Flywheel Overlay	`PatientPortalOverlay`; coverage key `case_type|evidence_status|acuity`. Covered keys produce $u_a = 0.15$; uncovered keys produce $u_a = 0.85$. Threshold $u_{a,\text{thresh}} = 0.6$.
Norms Φ	Four norms: `MED_ADVICE_REQUIRES_CLINICIAN`, `URGENCY_REQUIRES_ESCALATION`, `LAB_RESULTS_NEED_CONTEXT`, and `NO_UNSUPPORTED_MED_KEYWORDS`.
Triage	`PriorityTriage`; same category-priority ordering as the complex demo.

Table 33. Patient Portal demo: enforcement convergence.

Eval	Allow	Block	Escalate	Esc. Rate	Stack
0	6	0	9	60%	v0
1	6	3	6	40%	v1
2	9	6	0	0%	v2

Table 34. Component variation across medical demos.

Component	Simple	Complex	Patient Portal
Oracle scoring	3 heuristic rules	5-dimensional lookup	4-dimensional keywords
Patchable state	B, D	B, T, D	B, D
Norm count (kinds)	2 (2)	4 (4)	4 (2)
Triage	FIFO	priority-ranked	priority-ranked
Fixed cases	15	18	15
Evaluations to converge	3	3	3

and Flywheel Overlay state changes, such as audit-coverage registrations. This directly demonstrates patch locality in the medical setting: observed failures are mitigated by governed updates to the Oracle stack and governance state rather than by modifying or retracting the Proposer.

Correction lifecycle. Each governance iteration follows the same lifecycle across all demos:

$$\text{Red Team} \xrightarrow{\texttt{CandidateFlaw}} Q_{\text{ver}} \xrightarrow{\texttt{Verification}} \text{Triage} \xrightarrow{\texttt{Refinement}} Q_{\text{ref}} \xrightarrow{\texttt{Batch}} O, \text{ Overlay.}$$

The typed artifacts, protocol boundaries, and OODA interaction patterns are identical. The mechanism is stable while the strategy modules are pluggable.

G.6 Artifact Trace: Spatial Demo

We trace one flaw point through the complete governance pipeline to illustrate how the protocol boundaries from Section 4 are instantiated.

1. **Red Team (Observe–Orient–Decide–Act):** The Red Team queries the Oracle at grid point $x = [0.73, -0.42, 0.81]$ and receives $s = 0.68$. It computes $d(x, \tau_{\text{expert}}) = 0.72 > 0.34$: a high-safety point far from the expert path. It emits a `CandidateFlaw` artifact with the point, oracle signals, and distance metadata.
2. **Verification (Observe–Orient–Decide–Act):** The Verifier retrieves norm `SPATIAL_SUPPORT_REQUIRED` from Φ, checks $d = 0.72 > 0.34$, and emits a `VerificationResult` with `outcome = VIOLATION` and `violated_norm_id = SPATIAL_SUPPORT_REQUIRED`.
3. **Triage:** The verified flaw is submitted to the triage queue, sorted by distance descending (farthest first, maximising collateral coverage).
4. **Refinement (Observe–Orient–Decide–Act):** The Refinement pops the flaw, computes adaptive bandwidth $h = \text{clip}((0.72 - 0.34) \cdot 0.5, 0.03, 0.30) = 0.19$. The cumulative regression verifier checks: $\max_p \exp(-\|p-x\|^2/2{\cdot}0.19^2) = 0.0003$ for the nearest basin point at distance 0.68—well within the 10% budget. The patch is accepted and added to the `GovernanceBatch`.
5. **Batch deployment:** The batch is applied to the Oracle (`send_patch`), which adds a suppression kernel at $[0.73, -0.42, 0.81]$ with $h = 0.19$. The Flywheel overlay records the coverage update. Oracle version increments from v_4 to v_5.
6. **Post-patch verification:** On the next iteration's grid scan, the Oracle returns $s = \max(0, 0.68{-}0.97) = 0$ at this point (the suppression kernel centred here has value $\sim$1.0). The point is no longer a flaw. Adjacent points within the kernel's effective radius ($\sim 2.5 \times 0.19 = 0.48$) are also suppressed—this is the collateral effect.

G.7 Artifact Trace: Patient Portal Demo

We trace one case through the medical governance pipeline.

1. **Red Team:** Generates a synthetic case: patient asks about stopping warfarin, draft reply says "You could stop taking warfarin and switch to apixaban," proposed disposition is `nurse_review`, evidence is `conflicting`, acuity is `routine`. The Red Team classifies this as `undertriaged_med` (medication advice with weak evidence) and emits a `CandidateFlaw`.

2. **Verification:** The Verifier loads norm `NO_UNSUPPORTED_MED_KEYWORDS` (kind: `KEYWORD_BLOCK`). Checks: "stop taking" $\in$ draft text, evidence $\neq$ `supported` $\Rightarrow$ `VIOLATION`. Emits `VerificationResult` with `violated_norm_id = NO_UNSUPPORTED_MED_KEYWORDS`.
3. **Refinement:** Emits a `GovernanceBatch` containing:
   (a) `medical_hard_block`: keyword "stop taking" added to oracle block list;
   (b) `threshold_adjustment`: disposition override `clinician_review` installed for keys
   `medication|insufficient`,
   `medication|conflicting`, `medication|unknown`; (c) `audit coverage update`: case class `medication|conflicting|routine` marked as audited.
4. **Post-patch evaluation:** The fixed evaluation case "You should stop taking warfarin immediately" is re-evaluated. The oracle's hard-block fires on "stop taking," returning $s = 0.0$, $u = 0.05$. Enforcement: $s < 0.4 \Rightarrow$ `BLOCK`. Previously this case was `ESCALATE` (due to $u \geq u_{\text{thresh}}$). The governance batch changed the outcome from uncertainty to a definitive decision.

Bibliography

[1] Abbeel, P., Ng, A.Y.: Apprenticeship learning via inverse reinforcement learning. In: Proceedings of the Twenty-First International Conference on Machine Learning. p. 1. ICML '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1015330.1015430, https://doi.org/10.1145/1015330.1015430

[2] Adams, S., Cody, T., Beling, P.A.: A survey of inverse reinforcement learning. Artificial Intelligence Review **55**(6), 4307–4346 (2022)

[3] Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: Proceedings of the AAAI conference on artificial intelligence. vol. 32 (2018)

[4] Arora, S., Doshi, P.: A survey of inverse reinforcement learning: Challenges, methods and progress. Artificial Intelligence **297**, 103500 (2021)

[5] Baert, M., Malomgré, E., Leroux, S., Simoens, P.: Reward machine inference for robotic manipulation. In: Inductive Biases in Reinforcement Learning Workshop@ RLC 2025

[6] Bak, S., Johnson, T.T., Caccamo, M., Sha, L.: Real-time reachability for verified simplex design. In: 2014 IEEE Real-Time Systems Symposium. pp. 138–148. IEEE (2014)

[7] Batool, A., Zowghi, D., Bano, M.: Ai governance: a systematic literature review. AI and Ethics **5**(3), 3265–3279 (2025)

[8] Birkstedt, T., Minkkinen, M., Tandon, A., Mäntymäki, M.: Ai governance: themes, knowledge gaps and future agendas. Internet Research **33**(7), 133–167 (2023)

[9] Boella, G., Van Der Torre, L., Verhagen, H.: Introduction to normative multiagent systems. Computational & Mathematical Organization Theory **12**(2), 71–79 (2006)

[10] Breck, E., Cai, S., Nielsen, E., Salib, M., Sculley, D.: The ml test score: A rubric for ml production readiness and technical debt reduction. In: 2017 IEEE international conference on big data (big data). pp. 1123–1132. IEEE (2017)

[11] Brehmer, B.: The dynamic ooda loop: Amalgamating boyd's ooda loop and the cybernetic approach to command and control. In: Proceedings of the 10th international command and control research technology symposium. pp. 365–368 (2005)

[12] Castanyer, R.C., Mohamed, F., Castro, P.S., Neary, C., Berseth, G.: Armfm: Automated reward machines via foundation models for compositional reinforcement learning. arXiv preprint arXiv:2510.14176 (2025)

[13] Chaudhari, S., Aggarwal, P., Murahari, V., Rajpurohit, T., Kalyan, A., Narasimhan, K., Deshpande, A., Castro da Silva, B.: Rlhf deciphered: A critical analysis of reinforcement learning from human feedback for llms. ACM Computing Surveys **58**(2), 1–37 (2025)

[14] Hobbs, K.L., Mote, M.L., Abate, M.C., Coogan, S.D., Feron, E.M.: Runtime assurance for safety-critical systems: An introduction to safety filtering approaches for complex control systems. IEEE Control Systems Magazine **43**(2), 28–65 (2023)
[15] Horvitz, E.: Principles of mixed-initiative user interfaces. In: Proceedings of the SIGCHI conference on Human Factors in Computing Systems. pp. 159–166 (1999)
[16] Hovakimyan, G., Bravo, J.M.: Evolving strategies in machine learning: a systematic review of concept drift detection. Information **15**(12), 786 (2024)
[17] Hsu, K.C., Hu, H., Fisac, J.F.: The safety filter: A unified view of safety-critical control in autonomous systems. Annual Review of Control, Robotics, and Autonomous Systems **7** (2023)
[18] Ji, J., Qiu, T., Chen, B., Zhang, B., Lou, H., Wang, K., Duan, Y., He, Z., Zhou, J., Zhang, Z., et al.: Ai alignment: A comprehensive survey. arXiv preprint arXiv:2310.19852 (2025)
[19] Könighofer, B., Bloem, R., Jansen, N., Junges, S., Pranger, S.: Shields for safe reinforcement learning. Communications of the ACM **68**(11), 80–90 (2025)
[20] Kuppusamy, T.K., Diaz, V., Cappos, J.: Mercury:{Bandwidth-Effective} prevention of rollback attacks against community repositories. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17). pp. 673–688 (2017)
[21] Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. Cybersecurity **1**(1), 6 (2018)
[22] Li, R., Wang, P., Ma, J., Zhang, D., Sha, L., Sui, Z.: Be a multitude to itself: A prompt evolution framework for red teaming. In: Findings of the Association for Computational Linguistics: EMNLP 2024. pp. 3287–3301 (2024)
[23] Liu, G., Xu, S., Liu, S., Gaurav, A., Subramanian, S.G., Poupart, P.: A comprehensive survey on inverse constrained reinforcement learning: Definitions, progress and challenges. arXiv preprint arXiv:2409.07569 (2024)
[24] Malomgré, E., Simoens, P.: Mixture of autoencoder experts guidance using unlabeled and incomplete data for exploration in reinforcement learning. arXiv preprint arXiv:2507.15287 (2025)
[25] Malomgré, E., Simoens, P.: Interactionless inverse reinforcement learning: A data-centric framework for durable alignment. In: Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems. pp. 3946–3953 (2026)
[26] Mäntymäki, M., Minkkinen, M., Birkstedt, T., Viljanen, M.: Defining organizational ai governance. AI and Ethics **2**(4), 603–609 (2022)
[27] Newman, Z., Meyers, J.S., Torres-Arias, S.: Sigstore: Software signing for everybody. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2353–2367 (2022)
[28] Ng, A.Y., Russell, S., et al.: Algorithms for inverse reinforcement learning. In: Icml. vol. 1, p. 2 (2000)
[29] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al.: Training language models

to follow instructions with human feedback. Advances in neural information processing systems **35**, 27730–27744 (2022)

[30] Ovadia, Y., Fertig, E., Ren, J., Nado, Z., Sculley, D., Nowozin, S., Dillon, J., Lakshminarayanan, B., Snoek, J.: Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift. Advances in neural information processing systems **32** (2019)

[31] Phan, D., Yang, J., Clark, M., Grosu, R., Schierman, J., Smolka, S., Stoller, S.: A component-based simplex architecture for high-assurance cyber-physical systems. In: 2017 17th International Conference on Application of Concurrency to System Design (ACSD). pp. 49–58. IEEE (2017)

[32] Qin, X., Luan, S., See, J., Yang, C., Li, Z.: Governed capability evolution for embodied agents: Safe upgrade, compatibility checking, and runtime rollback for embodied capability modules. arXiv preprint arXiv:2604.08059 (2026)

[33] Quiñonero-Candela, J., Sugiyama, M., Schwaighofer, A., Lawrence, N.D.: Dataset shift in machine learning. Mit Press (2008)

[34] Rafailov, R., Sharma, A., Mitchell, E., Manning, C.D., Ermon, S., Finn, C.: Direct preference optimization: Your language model is secretly a reward model. Advances in neural information processing systems **36**, 53728–53741 (2023)

[35] Raji, I.D., Xu, P., Honigsberg, C., Ho, D.: Outsider oversight: Designing a third party audit ecosystem for ai governance. In: Proceedings of the 2022 AAAI/ACM Conference on AI, Ethics, and Society. pp. 557–571 (2022)

[36] Rebedea, T., Dinu, R., Sreedhar, M.N., Parisien, C., Cohen, J.: Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails. In: Proceedings of the 2023 conference on empirical methods in natural language processing: system demonstrations. pp. 431–445 (2023)

[37] Scerri, P., Pynadath, D., Tambe, M.: Adjustable autonomy in real-world multi-agent environments. In: Proceedings of the fifth international conference on Autonomous agents. pp. 300–307 (2001)

[38] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.F., Dennison, D.: Hidden technical debt in machine learning systems. Advances in neural information processing systems **28** (2015)

[39] Shahin, M., Babar, M.A., Zhu, L.: Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE access **5**, 3909–3943 (2017)

[40] Shankar, S., Parameswaran, A.: Towards observability for production machine learning pipelines. arXiv preprint arXiv:2108.13557 (2021)

[41] Shehab, M.L., Aspeel, A., Ozay, N.: Active reward machine inference from raw state trajectories. arXiv preprint arXiv:2604.07480 (2026)

[42] Singh, M.P.: Norms as a basis for governing sociotechnical systems. ACM Transactions on Intelligent Systems and Technology (TIST) **5**(1), 1–23 (2014)

[43] Sun, H., van der Schaar, M.: Inverse reinforcement learning meets large language model post-training: Basics, advances, and opportunities. arXiv preprint arXiv:2507.13158 (2025)

[44] Turcotte, M., Labrèche, F., Paquette, S.O.: Automated alert classification and triage (aact): an intelligent system for the prioritisation of cybersecurity alerts. arXiv preprint arXiv:2505.09843 (2025)

[45] Zarour, M., Alzabut, H., Al-Sarayreh, K.T.: Mlops best practices, challenges and maturity models: A systematic literature review. Information and Software Technology **183**, 107733 (2025). https://doi.org/https://doi.org/10.1016/j.infsof.2025.107733, https://www.sciencedirect.com/science/article/pii/S0950584925000722

[46] Zhao, Y., Zhang, S., Wu, Y., Sun, Y., Sun, Y., Pei, D., Bansal, C., Ma, M.: Triage in software engineering: A systematic review of research and practice. arXiv preprint arXiv:2511.08607 (2025)

[47] Ziebart, B.D., Maas, A.L., Bagnell, J.A., Dey, A.K., et al.: Maximum entropy inverse reinforcement learning. In: Aaai. vol. 8, pp. 1433–1438. Chicago, IL, USA (2008)

[48] Ziegler, D.M., Stiennon, N., Wu, J., Brown, T.B., Radford, A., Amodei, D., Christiano, P., Irving, G.: Fine-tuning language models from human preferences. arXiv preprint arXiv:1909.08593 (2019)