Regimented or Enforced

The Controllability Boundary for Agent Governance

Paper 2 of the Harbor program: a supervisory-control characterization of what an agent runtime can prevent

Prepared for Erich Owens

August 2026

Abstract

An agent hypervisor — a runtime that mediates an LLM agent's file writes, network egress, tool executions, pushes, spawns, and API calls — is asked to *enforce* governance policies. Some it can enforce by prevention: the forbidden thing never happens. Others it can only enforce by detection and compensation: the thing happens, and the runtime notices afterward. We show the boundary between the two is not an engineering contingency but a theorem: a prefix-closed safety policy is *regimentable* (preventable pre-effect) if and only if it is controllable in the sense of Ramadge–Wonham supervisory control theory, with respect to the uncontrollable event set of an LLM (model-internal steps and in-context reads). Schneider's classical result — an EM-enforceable policy is a safety property, though not conversely — is the ambient case of this characterization, and Basin et al.'s controllable/observable refinement of it is the criterion we identify with controllability; the identification is what imports supervisory control's synthesis machinery. The characterization is executable: a product-automaton checker classifies a realistic policy set (egress, push, write, exec, spawn: regimentable; token emission, context reads, and internal plans, where the confident falsehood lives: detect-only forever), and settles the decisive compound case — "no egress after reading a secret" *is* regimentable, because the policy permits the uncontrollable trigger and gates only the controllable effect. That single classification is the design theorem behind clean-room agent computation: gate the channel, never the token.

Express lane. *A governance policy can be enforced by prevention iff no legal prefix of agent behavior can be pushed out of the policy by an event the runtime cannot intercept — Ramadge–Wonham controllability with respect to* $\Sigma_u = \{$*model-internal steps, in-context reads*$\}$*; formal statement and proof in the box of §5, machine-checked policy table in §6.* Experts: read the box, the table, and §7; the rest is scaffolding.

1 The scene: two kinds of "the runtime enforces it"

A team runs coding agents behind a single-writer daemon. Every file write, network call, subprocess spawn, and `git push` passes through the daemon's effect boundary, where policy is checked before the effect lands. The governance document lists rules: *no force-push to main*; *no network egress from sandboxed jobs*; *no writes outside the granted scope*; and also *no confident falsehoods in status reports*; *never incorporate the customer's secret into your plan.*

The operations lead asks the obvious question: which of these does the runtime *guarantee*, and which does it merely *watch for*? The first three feel different from the last two, and every practitioner senses it. The force-push rule is airtight — the daemon owns the git remote credentials, and a push that never happens needs no forgiveness. The confident-lie rule is not airtight and never will be: the lie is composed inside the model's forward pass, and by the time any mediator sees tokens, the lie already exists. The practitioner's instinct is correct. This

paper proves it is not an instinct but an instance of a thirty-nine-year-old theorem, and extracts the exact boundary.

The idea in one breath.

Split the agent's event alphabet into what the runtime can refuse (mediated effects) and what it cannot (model-internal steps); a safety policy is preventable iff no uncontrollable event can carry a legal history out of the policy — and that is precisely Ramadge–Wonham controllability.

Intuition: the bouncer. A club has one door and a bouncer. The bouncer can refuse *entry* — entry is controllable, mediated by the door. The bouncer cannot refuse a patron's *thought* — thought is uncontrollable, unmediated by anything the club owns. A door policy ("no entry after 2am") is enforceable by prevention. A thought policy ("no ill intent inside") is enforceable only by observation and ejection — detection and compensation. Now the structural mapping (Figure 1): the daemon's effect boundary is the door; Σ_c (fs_write, net_egress, exec_tool, git_push, spawn_child, api_call) is entry; Σ_u (model_emit_token, in_context_read, internal_plan, hidden_activation) is thought. The analogy maps relations, not surface features, so it earns a prediction: a *compound* rule mentioning thought but gating only the door — "no entry for anyone who arrived after the fight started" — should still be enforceable by prevention, because the bouncer needs only to *observe* the trigger and *refuse* the door. Section 7 confirms exactly this for agents.

The predictable misread, preempted: the analogy does *not* say internal events are invisible — tokens are eminently loggable after emission. Uncontrollable means *unrefusable before the fact*, not unobservable after it. (When events are also unobservable, a second, stricter theory applies — §9.)

2 The vocabulary, defined

The theorem is imported wholesale from supervisory control theory, a branch of control engineering that has been mature since the late 1980s and whose vocabulary is almost entirely unknown outside it. None of it is difficult. Every term below is a statement about strings and sets of strings, and a reader who is comfortable with regular languages already has the prerequisites. Definitions in the order they are needed, each used immediately after it is given.

An *alphabet* Σ is just the finite set of event types the system can emit — for us, `fs_write`, `net_egress`, `model_emit_token` and the rest. A *string* over Σ is one possible history of an execution, in order. Everything in this paper is a claim about which histories can happen.

A set of strings K is *prefix-closed* when every prefix of a member is itself a member: if the first ten events of a run are allowed, so are the first nine. This is a modelling choice with teeth. It says a policy can be violated only by *doing* something, never by failing to do something — a run that has not yet violated the policy is still legal. Prefix-closed languages are exactly the *safety* properties in Lamport's sense: a violation has a finite witness, and once you have violated it you can never un-violate it. Liveness properties ("the agent eventually reports") are not prefix-closed, and are out of scope for the same reason no monitor can enforce them — there is no finite prefix at which you can say it failed.

The *plant* L is the set of histories that are *physically* possible, before any policy is imposed. The name is inherited from the factory-floor origins of the theory, where the plant was a literal plant. We take $L = \Sigma^*$: any interleaving of events can occur, because nothing about the hardware forbids one. A *policy* is then a prefix-closed $K \subseteq L$: the set of histories the governance document allows. We write $\overline{K}$ for prefix closure, so $\overline{K} = K$ throughout; the bar is kept only to match the standard statement of the theorem.

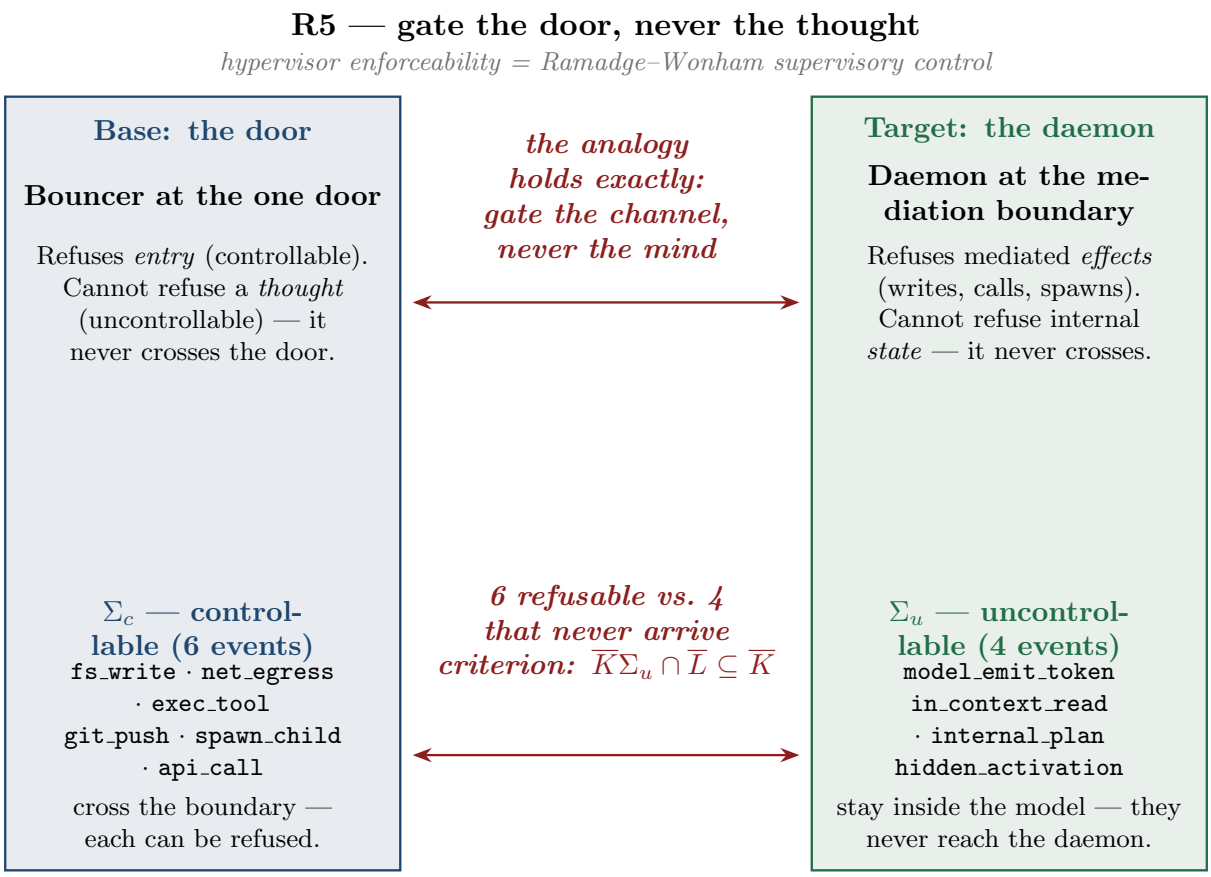

Figure 1: What a runtime can prevent is decided by *where an event crosses*, not by what the rule is about. Left: a club with one door — the bouncer refuses entry (controllable) and cannot refuse a thought (uncontrollable). Right: the daemon's mediation boundary — it can refuse fs_write, net_egress, exec_tool, git_push, spawn_child and api_call, and cannot refuse token emission, in-context reads, internal plans or hidden activations. The arrows carry the theorem: refusable-at-the-boundary ⇔ preventable (regimentable); internal ⇔ detect-only forever. The criterion under the map, $\overline{K}\Sigma_u \cap \overline{L} \subseteq \overline{K}$, is Ramadge–Wonham controllability [verified framework].

A *supervisor* is a function from the history so far to a set of events to disable next. The whole theorem turns on that function's *type*: it may disable members of Σ_c and nothing else. *Controllable* events are the ones the supervisor can refuse, because they pass through a mediation boundary before taking effect; *uncontrollable* events are the ones that complete before any mediator sees them.

The *closed loop* is the set of histories that actually occur once the supervisor is attached. A supervisor *regiments* a policy when the closed loop equals the policy exactly: nothing forbidden happens, and nothing permitted is lost. That second half is easy to overlook and is half the value — a supervisor that achieves safety by disabling everything is not a solution.

Controllability is then the one condition in the box of §5, and in words it says: no uncontrollable event, occurring after a legal history, takes you out of the policy. It is a property of the policy relative to an alphabet split, not a property of any mechanism — which is why no amount of engineering escapes it.

When a policy fails that condition, the *supremal controllable sublanguage* $\sup\mathcal{C}(K)$ is the largest sub-policy that satisfies it — the best preventable approximation of an unpreventable rule. It always exists, and for regular languages it is computable; §5 says why. This object is the reason the identification with control theory pays: §8 uses it to prove that a ban on the model's internal planning has *no* nontrivial preventable approximation at all.

Maximal permissiveness is the property that a synthesized supervisor disables nothing it does not have to. The construction in the theorem's forward direction has it by inspection, which is why regimentation costs no legal behaviour when the rule is controllable at all.

3 The model

Events and the mediation boundary. Fix a finite alphabet $\Sigma = \Sigma_c \uplus \Sigma_u$ of agent events. Σ_c (*controllable*) contains the events that pass through the runtime's mediation boundary before taking effect, so the runtime can disable any of them at any instant: in our reference alphabet, $\Sigma_c = \{$`fs_write`, `net_egress`, `exec_tool`, `git_push`, `spawn_child`, `api_call`$\}$. Σ_u (*uncontrollable*) contains the events that occur before any mediator can intervene: $\Sigma_u = \{$`model_emit_token`, `in_context_read`, `internal_plan`, `hidden_activation`$\}$. The split is a fact about the architecture, not a modeling choice: the daemon owns the syscall-and-credential surface and does not own the forward pass.

What "regiment" means, formally. A *regimentation mechanism* is a supervisor $f : \overline{L} \to 2^{\Sigma_c}$ mapping each history to a set of *disabled controllable* events; the supervised plant executes any event not disabled. It *regiments* the policy K when the closed loop equals $\overline{K}$, in the sense of §2. Anything short of that — allowing the violation and then flagging, rolling back, slashing a bond — is *detect-and-compensate*, a legitimate but categorically weaker enforcement mode.

4 The ambient result, and why the alphabet split refines it

Schneider's classical theorem says that a security policy enforceable by an execution monitor (EM) — a mechanism that watches a single execution and can terminate it — *is* a safety property [4]. The direction matters, and Schneider is explicit that only one direction holds: "the converse—that all safety properties have EM enforcement mechanisms—does not hold" [4, p. 35]. Safety is necessary for EM-enforceability, not sufficient. That result is the *ambient* case of ours: it implicitly assumes the monitor mediates *every* event, i.e. $\Sigma_u = \emptyset$. Under complete mediation, controllability is vacuous (there is no uncontrollable event to carry a history out of K), so the alphabet split stops doing any work and the only surviving obstruction is safety itself. Our theorem is at that point strictly stronger than Schneider's, and the extra strength is not ours either: restricting to regular K and L silently supplies the decidability and nonemptiness side conditions that Basin et al. [8] state separately (§10).

An LLM agent runtime does not enjoy complete mediation. The model's forward pass emits tokens, reads its context, and forms plans on hardware the daemon schedules but does not gate step-by-step; these events happen and *then* become visible, which in supervisory-control terms makes them uncontrollable. Once $\Sigma_u \neq \emptyset$, safety remains *necessary* for preventability (a liveness violation has no finite witness to block) but is no longer *sufficient*: "never `internal_plan`" is as prefix-closed a safety property as "never `git_push`," yet only one of them is preventable. The refinement that separates them is Ramadge–Wonham controllability [1, 2], developed for exactly this situation — a plant in which some events (machine breakdowns, arrivals) cannot be disabled by any supervisor. Anderson's reference-monitor tradition [5] contributes the design obligation (complete mediation *of the controllable alphabet*: every Σ_c event actually passes the boundary), and Rushby's separation-kernel analysis [6] the standard for verifying that the obligation holds; neither supplies the boundary itself. The boundary is controllability.

5 The theorem

Theorem (Regimentation = controllability). Let $L \subseteq \Sigma^*$ be a prefix-closed plant over $\Sigma = \Sigma_c \uplus \Sigma_u$, and let $K \subseteq L$ be a nonempty prefix-closed safety policy. There exists a regimentation mechanism for K — a supervisor disabling only controllable events whose closed loop realizes exactly $\overline{K}$ — if and only if K is *controllable* with respect to L and Σ_u:

$$\overline{K}\,\Sigma_u \;\cap\; \overline{L} \;\subseteq\; \overline{K},$$

i.e. no uncontrollable event, physically enabled after a legal history, exits the policy. When the condition fails, no runtime confined to disabling Σ_c can prevent the violation; the best achievable enforcement is detect-and-compensate, and the largest preventable weakening of the policy is the supremal controllable sublanguage $\sup \mathcal{C}(K)$, which exists and is computable for regular K and L [verified framework, Ramadge–Wonham 1987].

Proof. ($\Leftarrow$) Suppose K is controllable. Define the supervisor $f(s) = \{a \in \Sigma_c : sa \in \overline{L} \setminus \overline{K}\}$ for $s \in \overline{K}$: after any legal history, disable exactly the controllable events that would exit the policy. We show the closed-loop language equals $\overline{K}$, by induction on history length. The empty history lies in $\overline{K}$ (nonempty and prefix-closed). Assume the current history $s \in \overline{K}$. Any event a the closed loop can execute next is enabled in the plant ($sa \in \overline{L}$) and not disabled. If $a \in \Sigma_c$, then $a \notin f(s)$ means $sa \in \overline{K}$ by construction. If $a \in \Sigma_u$, then $sa \in \overline{K}\Sigma_u \cap \overline{L} \subseteq \overline{K}$ by controllability. Either way the closed loop stays in $\overline{K}$, so its language is contained in $\overline{K}$. Conversely every $sa \in \overline{K}$ is reachable: $a \in \Sigma_u$ is never disabled, and $a \in \Sigma_c$ with $sa \in \overline{K}$ is not in $f(s)$; so containment is equality, and f regiments K.

($\Rightarrow$) Suppose K is not controllable: there exist $s \in \overline{K}$ and $\sigma \in \Sigma_u$ with $s\sigma \in \overline{L}$ but $s\sigma \notin \overline{K}$. Let M be any regimentation mechanism for K. Since M realizes exactly $\overline{K}$, the legal history s is reachable under M — M cannot forbid it without cutting $\overline{K}$ down. At s, the plant enables σ, and M cannot disable it: $\sigma \notin \Sigma_c$, and disabling it is outside M's type — physically, the event completes before the mediation boundary sees it. So the closed loop under M reaches $s\sigma \notin \overline{K}$, contradicting the assumption that M's closed loop is contained in $\overline{K}$. Hence no regimentation mechanism exists, and any enforcement of K must tolerate the reachable violation $s\sigma$ — observe it after the fact and respond, which is detect-and-compensate by definition. The existence, uniqueness, and computability of $\sup \mathcal{C}(K)$ are imported from Ramadge–Wonham [1]: controllable sublanguages of K are closed under arbitrary union, so a supremal element exists, and for regular languages it is computed by a fixpoint iteration on the product automaton. □

Two remarks on what the proof used. First, the forward direction is constructive and *maximally permissive*: the supervisor disables nothing except the controllable exits, so regimentation costs no legal behavior — the governance rule and the agent's freedom are not in tension when the rule is controllable. Second, the reverse direction used nothing about the mechanism except its type ($f : \overline{L} \to 2^{\Sigma_c}$). No cleverness inside the runtime — better prompts, faster monitors, a smarter policy engine — escapes the theorem, because the obstruction is that σ finishes before any decision point the runtime owns.

6 The characterization, made runnable

The controllability test is decidable by a product-automaton search: walk the synchronous product of plant and policy automaton; flag any reachable product state where the plant enables an uncontrollable event that the policy disables. We implemented the test over the reference alphabet and ran it on the governance policy set. The checker[1] produces:

[1] `skills/harbor-results/scripts/b3_controllability.py`; deterministic, no seed needed; exits 0.

Policy	Event class	Regimentable?
forbid `net_egress`	controllable	**yes** (regiment)
forbid `git_push` (incl. force-push)	controllable	**yes** (regiment)
forbid `fs_write` (out-of-scope write)	controllable	**yes** (regiment)
forbid `exec_tool`	controllable	**yes** (regiment)
forbid `spawn_child`	controllable	**yes** (regiment)
forbid `model_emit_token`	uncontrollable	no (detect only)
forbid `in_context_read`	uncontrollable	no (detect only)
forbid `internal_plan` ("confident lie")	uncontrollable	no (detect only)
compound: no `net_egress` after secret `in_context_read`	mixed	**yes** (regiment)

All nine verdicts are the checker's output, verbatim in content [internal, `b3_controllability.py`]; the eight single-event rows also follow by hand from the theorem, which is the right way to first believe the table. *Numbers by hand:* take "forbid `git_push`." The policy automaton has one state with every event except `git_push` self-looping; `git_push` $\in \Sigma_c$, so the only event the policy ever disables is controllable, the premise of a controllability violation ($\sigma \in \Sigma_u$ enabled in plant, disabled in policy) is unsatisfiable, and K is controllable — regimentable. Now take "forbid `internal_plan`": the same one-state automaton disables `internal_plan` $\in \Sigma_u$, the universal plant enables it at the start state, so the empty history $s = \varepsilon \in \overline{K}$ and $\sigma =$ `internal_plan` witness $\overline{K}\Sigma_u \cap \overline{L} \not\subseteq \overline{K}$ — detect-only, forever, for any runtime with this alphabet split. *Now you try:* classify "forbid `api_call` after `spawn_child`" and "forbid `internal_plan` after `fs_write`." (Answers: the first gates a controllable event on a controllable trigger — regimentable; the second must disable an uncontrollable event in its post-trigger state, and the plant enables it there — detect-only.)

This table is the manuscript's needed re-grading of governance rules, stated correctly: *force-push, unauthorized egress, and out-of-scope writes are matters of structural authority and shall be prevented; a confident lie in a status report is semantic — its truth is not a condition the boundary can witness (boundary (vi)) — and shall be detected and compensated* — by audit, reputation, and slashing, which is the adjacent paper's machinery, not this one's. Figure 2 draws the same line as a single divider: everything above it gates at the mediation boundary, everything below it is detect-only forever, and the criterion on the divider is the whole theorem in one inequality.

7 The compound case: gate the channel, never the token

The last table row is the one the whole clean-room product line rests on. The policy *"no network egress after an in-context read of a secret"* mentions an uncontrollable event (`in_context_read`) — so a naive reading of the single-event rows predicts detect-only. The checker says otherwise:

```
COMPOUND POLICY: 'no net_egress after in_context_read of a secret'
  regimentable? YES
```

[internal, `b3_controllability.py`]. The mechanism is visible in the policy automaton: two states, `ok` and `tainted`. The uncontrollable read is *enabled in both* — the policy never tries to forbid it; it lets the read happen and records the taint transition. What the tainted state disables is `net_egress` alone, and egress is controllable.

This automaton is not new, and it is worth saying whose it is: it is Schneider's own Figure 1, a two-state security automaton for "a security policy that prohibits execution of `Send` operations after a `FileRead` has been executed" [4], with his `q_nfr`/`q_fr` relabelled `ok`/`tainted`. What the alphabet split adds is not the construction but the *grading*: under complete mediation the figure is simply one more enforceable safety policy, whereas once $\Sigma_u \neq \emptyset$ the same two states

nine-policy classification

policy	verdict
forbid `fs_write`	**regiment**
forbid `net_egress`	**regiment**
forbid `exec_tool`	**regiment**
forbid `git_push`	**regiment**
forbid `spawn_child`	**regiment**
compound: no `net_egress` after `in_context_read`	**regiment** ★
the divider carries the criterion: $\overline{K}\Sigma_u \cap \overline{L} \subseteq \overline{K}$ (Ramadge–Wonham 1987) — no uncontrollable event exits the spec	
forbid `model_emit_token`	**detect-only**
forbid `in_context_read`	**detect-only**
forbid `internal_plan`	**detect-only**

compound case: how it becomes regimentable

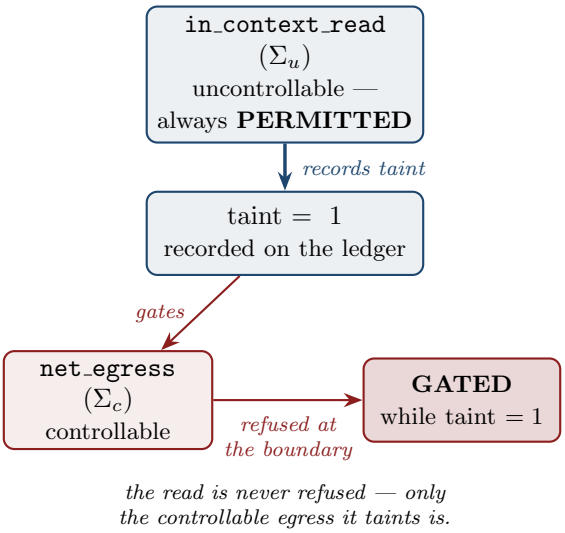

Figure 2: The regime diagram. Above the divider, the regimentable regime (Ramadge–Wonham controllable): forbid fs_write / net_egress / exec_tool / git_push / spawn, and the compound "no egress after secret read." Below, detect-only forever: forbid token emission / in-context read / internal plan / confident falsehood. The divider carries the criterion $\overline{K}\Sigma_u \cap \overline{L} \subseteq \overline{K}$ — no uncontrollable event exits the specification. Classifications reproduce from the product-automaton checker [internal, `b3_controllability.py`].

become the canonical witness that a policy may reference an unrefusable event freely so long as it never tries to refuse it. The design rule below is a reading of Schneider's example through controllability, not a new classification of it. Check the criterion: every uncontrollable event enabled by the plant is enabled by the policy in every reachable state, so $\overline{K}\Sigma_u \cap \overline{L} \subseteq \overline{K}$ holds and the theorem's constructive direction hands us the supervisor — track taint, gate the channel.

Three consequences, in increasing order of consequence.

First, the design rule. A compound trigger→effect policy is regimentable iff its *effect* events are controllable; the trigger may be as uncontrollable as it likes, provided the policy observes rather than forbids it. Uncontrollable events in the policy's *condition* are free; uncontrollable events in its *prohibition* are fatal.

Second, the clean room. The Sealed Harbor design (Paper 4) puts an agent alone in a room with a customer's secret data and promises the secret does not leave. The theorem dictates the only sound architecture: it is impossible to prevent the model from *reading* the secret (the read is uncontrollable — and useless to forbid, since reading is the job), and it is impossible to prevent the model from *incorporating* the secret into tokens and plans (uncontrollable again). What is possible — exactly and only — is to gate every controllable egress channel out of the room, with taint recorded from the moment of exposure. Gate the channel, never the token. Paper 4's companion noninterference result proves from the other direction that the gate suffices: observations outside the room are identical across secrets except at declared gate releases. Controllability says the gate is the only place a guarantee *can* live; noninterference says a guarantee *does* live there. This controllability boundary is thus the critical assumption behind the clean-room ledger's claims elsewhere in the program: "all spend is recorded" is precisely complete mediation of Σ_c.

Third, the special case that explains a folk theorem. Product reviews of agent governance converged on the slogan "a rule is enforceable iff it is a pure deterministic function of committed local DB state." That slogan is our theorem's special case $\Sigma_c = \{\text{DB writes}\}$: when the only mediated channel is the database commit, the regimentable policies are exactly those expressible over committed state. The general statement strictly dominates it — the slogan cannot express why force-push is preventable by a runtime that also owns the git credential channel, nor why a confident lie stays unpreventable no matter how much state is committed. Holding the plant fixed, widening Σ_c (owning more channels) is the only way to grow the regimentable set; no policy-language cleverness does. The plant is the other dial, and the box of §5 is where to see it: $\overline{L}$ sits on the left of the containment, so shrinking it weakens the antecedent and strictly more policies pass. Our own detect-only verdicts already say as much — both turn on "the universal plant enables `internal_plan`" (§6, §8) and are therefore conditional on the modelling choice $L = \Sigma^*$. An event that never becomes physically possible needs no supervisor at all, which is why data minimization and capability removal are governance moves and not merely hygiene: a secret never loaded into the room regiments this section's compound policy with no gate at all. What no richer rule syntax reaches is either dial — the criterion mentions only $\overline{K}$, Σ_u and $\overline{L}$.

8 A synthesis case study: the sandboxed review job

The theorem's constructive direction is not just an existence proof — it is the synthesis algorithm the Supremica / TCT tool family implements, and it is small enough here to run by hand. Consider a realistic composite policy for a sandboxed code-review job:

- P_1: no `net_egress`, ever (sandboxed);
- P_2: no `git_push`, ever (review jobs propose, humans land);
- P_3: after any `in_context_read` of customer material, no `fs_write` (findings leave through the mediated report channel, not the workspace).

Each P_i is a safety automaton; the composite policy is their synchronous product. P_1 and P_2 are one-state automata, P_3 is the two-state taint automaton of §7, so the product spec has $1 \times 1 \times 2 = 2$ states, `ok` and `tainted`, and the product with the universal one-state plant has the same two reachable states. The controllability check visits both states and tests each of the four uncontrollable events in each: all eight (state, event) pairs are enabled by the spec — neither state ever disables a member of Σ_u — so there are 0 violations and the composite is controllable [verified: eight cases, checkable by eye]. The maximally permissive supervisor falls out of the proof's construction as a two-row disablement map:

Supervisor state	Disabled ($\subseteq \Sigma_c$)	Everything else
`ok`	{`net_egress`, `git_push`}	enabled
`tainted`	{`net_egress`, `git_push`, `fs_write`}	enabled

Two states, one observed transition (`in_context_read` flips `ok` to `tainted`), three gates. That table *is* the synthesized supervisor — it is also, not coincidentally, exactly the policy engine a competent engineer would have written by hand. The value of synthesis is not that the answer is surprising; it is that the answer arrives with a maximal-permissiveness certificate (no legal behavior was sacrificed) and a controllability certificate (no uncontrollable event can force a violation), neither of which hand-written policy engines carry.

Synthesis is equally informative when it fails. Ask it to weaken the detect-only policy "forbid `internal_plan`" to its best preventable approximation: the supremal controllable sublanguage. Let $K' \subseteq K$ be any nonempty controllable sublanguage; it contains ε, the plant enables `internal_plan` at ε, so controllability forces $\varepsilon \cdot \texttt{internal\_plan} \in \overline{K'} \subseteq \overline{K}$ — contradicting that

K forbids it. Hence $\sup \mathcal{C}(K) = \emptyset$ [verified: three lines, above]. The theory refuses to pretend: there is no nontrivial preventable weakening of a thought ban — the nearest preventable policy prevents nothing — and any governance document claiming to "partially enforce" such a rule by prevention is describing detection with optimistic vocabulary.

9 Partial observation: the next boundary (future work)

Our theorem assumes the supervisor *observes* every event, even the ones it cannot disable — the daemon logs tokens after emission, reads after completion. Realistic deployments break this in a second place: some events are not only unrefusable but *unobservable* (activations never surfaced; state internal to a third-party tool; another principal's context). Supervisory control under partial observation is also solved territory: Lin–Wonham [3] characterize enforceability as controllability *plus observability* — the supervisor's decisions must be constant on observation-equivalence classes of histories, i.e. the policy may not demand different gate decisions after histories the runtime cannot tell apart. Observability strictly shrinks the regimentable set: a policy like "no egress after the model *considered* exfiltration" references unmediated internal *state* with no observable taint transition, so even though its prohibition (egress) is controllable, no supervisor can implement the gate condition. The practical reading: a compound policy is deployable only if its trigger is *witnessed at the mediation boundary* (the secret arrives via a mediated `in_context_read` the daemon performs, as in §7) — taint must be an event the daemon sees, not a state it infers. Figure 3 puts both boundaries on one grid: controllability separates its columns, observability its rows, and only the policies in the upper-right cell are preventable. Mapping the agent-governance policy corpus across the observability line, and connecting it to the assurance-level ladder (which integration tiers witness which triggers), is the natural sequel; we state it as future work rather than claim it, because the interesting part — which real triggers are witnessed at which assurance levels — is an empirical systems question, not a corollary.

10 Related work

Supervisory control of discrete-event systems originates with Ramadge and Wonham [1, 2]; we import controllability, the supremal controllable sublanguage, and the maximally permissive supervisor unchanged, and claim no new control theory. Lin and Wonham [3] supply the partial-observation refinement sketched in §9. Schneider [4] shows EM-enforceable policies are safety properties, and denies the converse (§4).

The alphabet split is not ours. Closest to our setting is Basin, Jugé, Klaedtke and Zălinescu [8], who refine Schneider by distinguishing system actions an enforcement mechanism *controls* from those it merely *observes*, and give necessary and sufficient conditions for enforceability with respect to that distinction. Their (U, O)-safety condition, in closed form $\mathrm{cl}(\mathrm{pre}^*(P \cap U) \cdot O^*) \cap U \subseteq P$, specializes on a universal plant and regular K to exactly the criterion in our box; they identify Schneider's setting as the $O = \emptyset$ case, as we do. We claim no priority for the split, for the characterization, or for the observation that Schneider is its degenerate case. What Basin et al. leave explicitly open is the identification itself: they note that "the Ramadge–Wonham framework ... has several similarities with our setting" and close with "it remains to be seen whether and how the domain of policy enforcement can benefit from the Ramadge–Wonham framework and the results around it." This paper closes that question in one direction, and the closing is the first contribution. Naming the condition *controllability* is not a relabeling: it imports the supremal controllable sublanguage, the maximal-permissiveness certificate, and decades of modular, hierarchical and tool-supported synthesis, none of which

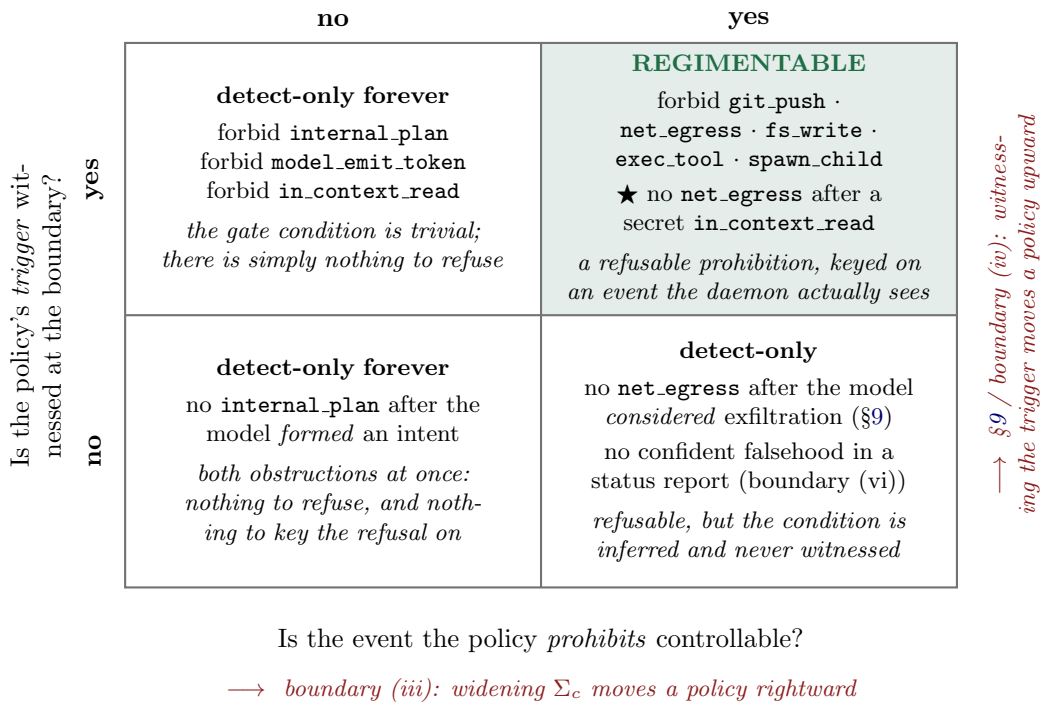

Figure 3: Controllability is only half the boundary: a policy is preventable exactly when its prohibition is refusable *and* its trigger is witnessed at the mediation boundary — the shaded quadrant, and nothing else. The theorem of §5 separates the columns; Lin–Wonham observability (§9) separates the rows. The two architectural moves the boundary box names are the two moves on this grid: widening Σ_c carries a policy rightward (boundary (iii)), witnessing a trigger carries it upward (boundary (iv)), and only a policy that makes both trips becomes preventable. The compound rule (★) is the paper's worked case of a rule whose naive reading puts it in the left column and whose correct grading puts it top-right; "no confident falsehood in a status report" is the case stuck one cell below, whatever the runtime owns. (The bottom-left entry is a constructed illustration; no rule in the corpus lands there.) Column membership reproduces from the product-automaton checker [internal, `b3_controllability.py`]; row membership is by inspection of each trigger.

the (U, O)-safety formulation provides — the $\sup \mathcal{C}(K) = \emptyset$ result of §8, which prices a thought-ban's best preventable approximation at nothing, has no counterpart in Basin et al. The second contribution is the instantiation: their canonical uncontrollable action is a clock tick, an event of the *environment*; ours are the generative model's own forward-pass steps, events of the regulated agent's own constitution — which is why the confident lie in a status report survives every widening of Σ_c: own the emission channel and the prohibition becomes controllable, but the gate condition is still a state of the agent the boundary never witnesses (boundary (vi), §9). A model whose uncontrollable events are clock ticks has no internal state of the regulated agent behind them and cannot state that corollary at all. Khoury and Hallé [11] generalize the binary split to a lattice of control levels; our two-valued split is the coarsening appropriate to a runtime that either owns a channel or does not.

Mechanism power is a different axis. The lineage descending from Schneider that classifies monitors by what they do to the trace is orthogonal to ours and should not be confused with it. Ligatti, Bauer and Walker [9] show that truncation, suppression, insertion and edit automata all *precisely* enforce exactly the safety properties, while an edit automaton *effectively* enforces any property by suppressing until a prefix is confirmed legal and then reinserting. Every one of

those operations requires the mechanism to be interposed and able to withhold the action; edit automata are powerful precisely because they can *delay*, which is the capability an uncontrollable event denies. Our detect-and-compensate is therefore not a weaker member of their taxonomy but a category outside it — all four of their mechanisms are mechanisms over Σ_c — and they name our regime as an out-of-model limitation ("the monitor is unable to interpose itself between the application and the device"), which Basin et al. later formalize. Falcone, Fernandez and Mounier [10] ask a question whose title is nearly ours and answer it on the other axis: holding complete mediation fixed and equipping the monitor with unbounded finite memory, the enforceable properties are exactly the response properties of the safety–progress hierarchy. Their refinement moves *up* the hierarchy by adding buffering power; ours stays at prefix-closed safety and removes mediation. Neither contains the other, and the joint statement — safety *and* controllable — is what a deployed agent runtime must satisfy.

The word "regimentation" is not ours either. In normative multi-agent systems it is an established term of art for precisely the distinction this paper draws: regimentation makes violation impossible by design, enforcement detects violations and responds with sanctions [12]. That community has long observed both halves of our theorem informally — that regimentation operates by mediating access to resources and the communication channel, and that "the regimentation of all actions is often difficult or impossible" [13] — and has drawn our alphabet split as a design taxonomy, separating regimentation of an agent's *mental states* (possible only when the agent is a white box) from regimentation of its *actions* (the black-box case) [14]. An LLM agent is the black-box case, and Σ_u is the formal content of its black-boxness. One paper in this community goes further than observing the distinction informally: Dastani, Sardina and Yazdanpanah [16] formalize regimentability for a norm as supervisory-control-theoretic controllability, proved by the same Ramadge–Wonham theorem we import, though they present it as an unremarked corollary and do not pursue enforceability beyond regimentation, synthesis, or classification. Our contribution relative to this literature is to replace "often difficult or impossible" with a decidable criterion, a synthesis procedure, and a machine-checked classification of a deployed governance corpus.

Adjacent and contemporaneous. Ray [15] asks what a certified runtime gate can enforce for tool-using agents and answers on a third axis: safety plus decidability of good-prefix membership relative to an oracle interface, with a fallible judge and a statistical certificate. His gate sees and decides on every proposed action — $\Sigma_u = \emptyset$ in our terms — so his boundary and ours are complements, not competitors: he asks what a gate can *represent and decide*, we ask which events a gate can *refuse at all*. The reference-monitor lineage — Anderson's complete-mediation obligation [5], Rushby's kernel-verification standard [6], and Goguen–Meseguer's policy-as-noninterference framing [7] — supplies the architecture our Σ_c presumes and the companion theory Paper 4 builds on. Closest in spirit on the synthesis side is the *shielding* line: Bloem et al. [17] synthesize a reactive shield that sits between an untrusted component and its environment and corrects its outputs at runtime, and Alshiekh et al. [18] apply the same construction to a reinforcement learner, whose policy the shield cannot inspect and does not try to. A shield is a supervisor in our sense — it owns the action channel and nothing inside the learner — so that literature is an independent instance of our Σ_c/Σ_u split arrived at from reactive synthesis rather than from discrete-event control; what it does not ask is which events *admit* a shield, which is our question.

The search trail, and its gap. We swept four literatures by hand — supervisory control of discrete-event systems, runtime verification and enforcement, normative multi-agent systems, and the reference-monitor tradition — and the citations above are what that sweep returned. It was not a systematic search: there is no protocol, no enumerated venue list, no screening

record, so "not found" here is not "proven nonexistent," and the two contribution claims of this section should be read with that discount applied. The adjacent fields most likely to hold a prior statement of our theorem and least covered by the sweep are safe reinforcement learning beyond the shielding papers above, LLM guardrail and inference-time filtering work, and the operating-systems capability literature. Boundary (vii) names the single outstanding item we already know about.

11 Honest boundary

What this theorem does NOT say. (i) It does not say detect-only policies are unenforceable or hopeless — detection feeding audit, reputation, and bond-slashing (Paper 3) is real enforcement; the theorem only says prevention is off the table, so pricing and compensation must be sized for violations that *will* occur. (ii) It does not certify any implementation: the theorem's Σ_c is the set of events *actually* mediated, and every unenumerated side channel (raw sockets, DNS, inherited file descriptors, /proc, git credential helpers, package managers, clipboard, provider callbacks) silently moves events from Σ_c to Σ_u and shrinks the regimentable set behind the operator's back — complete mediation of the claimed alphabet is an architectural obligation (Anderson; Rushby) that a red-team, not this proof, discharges. (iii) The classification is relative to the alphabet split: a runtime that gates model steps (e.g. token-level filtering with the model inside the boundary) changes Σ_u and re-grades the table — the theorem is a functor from architectures to boundaries, not one fixed verdict. (iv) The full-observation assumption is critical: policies whose triggers are unwitnessed internal state fall to the Lin–Wonham refinement (§9) even when their prohibitions are controllable. Items (iii) and (iv) are the two axes of Figure 3: (iii) is the horizontal move, (iv) the vertical one, and a policy must make both trips to become preventable. (v) The checker verifies the stated policy automata over the stated alphabet [internal, `b3_controllability.py`]; it is a faithful implementation of the test, not a mechanized proof of the theorem — an Isabelle/Coq formalization remains the submission gate. (vi) Prevention governs *effects*, never *semantics* — but for the governance rule of §1, "no confident falsehoods in status reports," the obstruction is *observability*, not controllability, and item (iii) is what forces the distinction. A runtime that gates model steps re-grades the table, and `internal_plan` is a row of that table, so the confident-lie row is re-gradable in principle; by §7's own design rule it is re-gradable in practice too, since a status report leaves through emission and then through `api_call`, `fs_write` or `net_egress` — all already controllable — which makes the prohibition controllable and the internal step a mere trigger. What no such runtime can do is *evaluate the gate*: "this report is false" is not a predicate over the event alphabet, and the condition is unwitnessed internal state rather than an event the daemon sees. The rule therefore falls to item (iv) and §9 — Lin–Wonham observability — and not to the box's inequality, which is why it stays unpreventable at *every* alphabet split and not merely at ours. (vii) Dastani, Sardina and Yazdanpanah's *Norm Enforcement as Supervisory Control* [16], obtained and read in full: their Theorem 1 proves a norm is "regimentable" iff a regimenting supervisor exists, by directly importing the same Ramadge–Wonham controllability theorem we do, in normative-multi-agent-systems vocabulary rather than ours. They do not claim the identification as a contribution — their stated novelty is extending it to sanction- and repair-based enforcement — and they do not address enforceability more generally, synthesis, or classification of a deployed corpus. This narrows but does not retract the priority claim in §10: the identification itself is folklore enough that two independent communities (Basin et al.'s security-policy line and this normative-systems line) import it in passing before this paper; what neither prior source gives is the synthesis procedure or the $\sup \mathcal{C}(K) = \emptyset$ result. The theorem and the

classification stand regardless, since neither depends on being first.

Reproducibility. The policy table and the compound-case verdict regenerate deterministically from `skills/harbor-results/scripts/b3_controllability.py` (pure Python, no dependencies, no randomness); Figures 1 and 2 regenerate from `docs/harbor-research/figures/src/r5_figures.py` (the relation map with the script's third row dropped, since the regime diagram's right panel already carries it), and Figure 3 is drawn from those same checker verdicts plus the worked policy of §9. Every number and verdict in this paper is tagged [verified] (externally recomputable from the cited literature or by hand) or [internal] (regenerates from the named script).

References

[1] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

[2] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[3] F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, 1988.

[4] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[5] J. P. Anderson. *Computer Security Technology Planning Study.* Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, 1972.

[6] J. Rushby. *Noninterference, Transitivity, and Channel-Control Security Policies.* Technical Report CSL-92-02, SRI International, December 1992.

[7] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 11–20, 1982.

[8] D. Basin, V. Jugé, F. Klaedtke, and E. Zălinescu. Enforceable security policies revisited. *ACM Transactions on Information and System Security*, 16(1):1–26, 2013.

[9] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, 2005.

[10] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14(3):349–382, 2012.

[11] R. Khoury and S. Hallé. Runtime enforcement with partial control. arXiv:1508.06525, 2015.

[12] D. Grossi, H. Aldewereld, and F. Dignum. Ubi lex, ibi poena: designing norm enforcement in e-institutions. In *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, pages 101–114. Springer, 2007.

[13] N. Criado and J. M. Such. Norm monitoring under partial action observability. arXiv:1505.03996, 2016.

[14] T. Balke. A taxonomy for ensuring institutional compliance in utility computing. Dagstuhl Seminar Proceedings 09121, 2009.

[15] S. Ray. What can be enforced? A theory of certified runtime safety for tool-using agents. arXiv:2607.22868, 2026.

[16] M. Dastani, S. Sardina, and V. Yazdanpanah. Norm enforcement as supervisory control. In *PRIMA 2017: Principles and Practice of Multi-Agent Systems*, LNAI 10621, pages 330–348. Springer, 2017.

[17] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang. Shield synthesis: runtime enforcement for reactive systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 533–548. Springer, 2015.

[18] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. Safe reinforcement learning via shielding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.