Design write-up

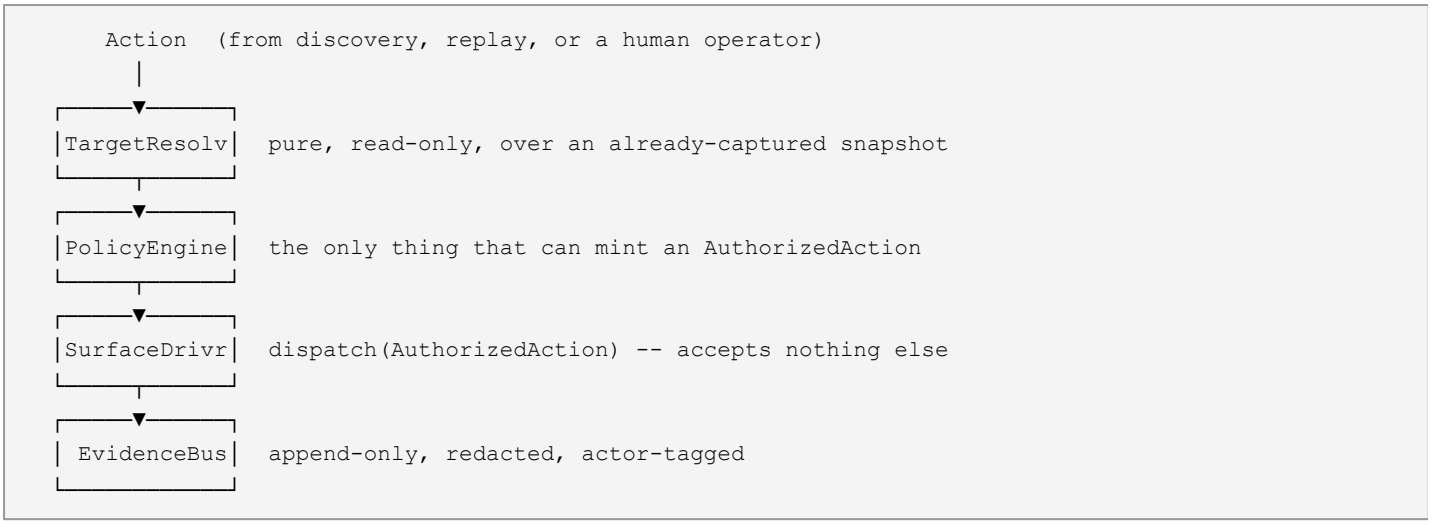
> **The model is allowed to be uncertain exactly once.** That uncertainty is compiled into a typed artifact that never needs it again, and every run after the first is deterministic.

Every claim below is backed by something runnable: an enforced import contract, a test, or a file in `evidence/`. What is not built says so in §7.

---

1. Architecture

An LLM drives a live legacy UI to a goal **once**. That run is compiled into a typed, versioned **capability artifact**. From then on the artifact is replayed with no model in the decision loop, and an AI agent invokes it like a typed function.



Discovery, replay, and human intervention share one path, differing only in who originates the action, so a guardrail cannot be present on one and missing from another. **Resolution runs before policy**: risk classification's second signal is *what the control says it does* ("Transfer Funds"), which needs the resolved node, so authorizing first would mean authorizing half-blind. The invariant is not "policy runs first" but *nothing reaches a surface without authorization* — enforced three ways: an **import contract**, a **type guard** (`dispatch()` accepts only an `AuthorizedAction`), and an **AST scan** that has caught violations the import graph missed.

Stack. Pydantic v2 lets the artifact schema *be* the type system. Playwright over CDP, since the accessibility tree is the closest browser analogue to UIA/AX. The model sits behind an `LlmPort` protocol at `temperature=0`. YAML artifacts, so a bank reviewer reads one without running anything.

2. Artifact schema

`cua.capability/v1` — Pydantic, YAML, immutable, content-addressed. Examples in `evidence/capabilities/`.

Targets are semantic, never selectors. A `TargetDescriptor` is role, accessible name, structural anchor, scope, ordinal. A CSS selector may appear only under `hints` — a cache accepted only if the node it finds *also* satisfies the semantic assertion. `role: button, name: "Search"` means the same in Windows UIA as in ARIA; `input.btn1` does not.

Business outcomes, recovery, and failure are three distinct top-level keys — the brief names conflating the first with the third as the most common mistake here, so the schema makes it structurally impossible rather than trusting the executor.

The capability is immutable; everything a *run* produces lives elsewhere — `RunRecord`, `CapabilityApproval`, `TenantBinding`, keyed by `id@version` — since a definition that accumulates telemetry stops being reviewable, and "which version ran?" stops being answerable. **`capability.id` is namespaced by vendor product, not tenant**, and `surface.driver_capabilities` declares what a driver must *be able to do* rather than naming a library: both preconditions for reuse across institutions running the same product.

The compiler emits a **draft** and says so: one run saw only the happy path, so it declares no outcomes and no recovery, with review notes naming what a reviewer must add — inventing detectors a run never observed would produce error handling that is fiction.

3. Determinism & error handling

Two type families, deliberately separate — *what we observed* and *how the run ended*:

OBSERVATIONCLASS	MEANING	LEADS TO
`EXPECTED`	matches a precondition/checkpoint	proceed
`BUSINESS_OUTCOME`	matches a declared outcome	terminate, **exit 0**
`RECOVERABLE`	matches a declared recovery rule	bounded remedy, then retry
`UNEXPECTED_STATE`	matches **nothing** declared	fail closed

`RunStatus` is terminal and separate: `SUCCESS | BUSINESS_OUTCOME | NEEDS_HUMAN | FAILED`.

"No such member" **exits 0** — a successful execution returning a negative answer, not a failure; collapsing the two turns a routine result into a 2am page and trains everyone to ignore the alert that also fires for real defects.

Fail-closed is the default. A state matching nothing declared stops the run rather than proceeding on the assumption a click probably worked; two surviving candidates is `TARGET_AMBIGUOUS`, and the resolver **refuses** rather than breaking the tie on a similarity score — a tunable threshold is a knob that eventually gets turned. Determinism itself is enforced, not asserted: `cua.replay` may not import `cua.agent` or any HTTP client, the resolution ladder scores purely with ties broken by document order (never dict iteration, clock or random), and the `vision` rung is hard-disabled in replay. Where a hard failure routes — pause for a human, or terminate — is `escalation.triggers`' decision, not the executor's.

Drift is measured as descent. A target recorded at `semantic_exact` that now resolves only at `structural_anchor` has moved, even though the run still succeeded, so `run_record.drift_score` rises before replays begin to fail rather than after — `make eval` reports determinism as a boolean per capability, since "94% deterministic" is not a property anyone can act on.

4. Heterogeneity & multi-tenant

The seam is `UiSnapshot` — a normalized `{node_id, role, name, value, states, bounds, scope}` tree. Drivers populate it from whatever their surface offers: a CDP accessibility tree today, Windows UIA tomorrow. **`cua.targeting` and `cua.perception` may not import Playwright or `cua.surfaces`** (enforced), so neither the artifact nor the resolver can grow web-shaped assumptions. `surfaces/desktop_uia/` is an interface-only stub documenting the UIA→`UiSnapshot` mapping.

Tenants are an **overlay, never a fork**: `TenantBinding` supplies vars and per-step overrides, resolved at load into an effective capability whose content hash covers the overlay.

The no-CSS claim is **measured**, in `evidence/evals/cross_tenant.md`. Variant B relabels controls *and* restyles markup, so every `hints.css` in the artifact names a selector that does not exist there — and the runs succeed anyway: 100%, zero wrong actions. Markup churn is carried by `semantic_exact`; rebranding by `structural_anchor` plus the overlay — both load-bearing, which is why the ladder has both. `structural_anchor` does not itself survive rebranding, since the anchor text *is* the label cell and gets relabeled too, so the honest claim is markup change, not vocabulary change; vocabulary is what the overlay is for. One limit that leaves: drift is computed against the *effective* capability, so an overlay reports zero drift for a surface that genuinely differs.

5. Escalation & handoff

`NEEDS_HUMAN` is a **first-class terminal status**, not an exception — which is what makes escalation architectural rather than bolted on.

The operator works the **same live session**, guarded by a `Lease{holder, epoch}`. Automation checks it on every dispatch — who holds the session, and has it changed hands since this run was authorized — either returns `LEASE_LOST`; both matter, since the epoch alone once left a replay running to `SUCCESS` while an operator held the session, because the check was opt-in and unasked.

Human input does not bypass policy. The console never injects events into the page; it submits `raw_input` to the broker, through the same resolve → authorize → dispatch → evidence path under a **`HUMAN` profile** — escalation widens authority deliberately and auditably, and the allowlist and redaction apply unchanged.

Control comes back. `ReplayExecutor.resume()` re-anchors against the live screen rather than `run()` with an offset: it skips the entrypoint navigation (which would reload the frameset and discard the operator's state), re-adopts the epoch the handoff produced, carries forward outputs and spent recovery attempts, and fails closed when the screen matches no step. A step is skipped only when its **postcondition** holds — without one, resume would re-submit the search the operator just ran. `cua console --capability <ref> --arm-fault <fault>` runs a capability against the supervised session, so the queue a reviewer opens has something waiting in it.

6. Safety

One chokepoint, three enforcements (§1), applied identically to all three actors.

Allowlist — domains, URL patterns, action types, step and duration caps, in `config/policy.yaml`. Navigation outside it is `NAVIGATION_BLOCKED`, for humans too; a capability's own `allowed_domains` only ever *narrows* the global list,

checked both on a `navigate`'s declared URL and again against where the session actually landed, since checking intent is not checking outcome. "Landed" means every frame, not the top-level document — on a frameset the shell never moves — and the driver also refuses an off-list document navigation in any frame, redirect hops included, before it is fetched.

Risk tiers — `safe → elevated → irreversible`, from three independent signals (action type × target semantics × artifact annotation), since any one alone is fooled. Automation is refused an irreversible action unless the step is declared irreversible **and** the run carries an approval of that exact content (or of its sealed base, through a `--base-url`-only binding) **and** a caller opt-in — checked by `PolicyEngine` at the dispatch itself, not only before the run, because either gate alone is one accident away from a wire transfer.

Redaction at every sink, including outbound model prompts. Writing a run record *requires* a redactor, and the secret resolver registers each value as it issues it, closing a path where an unregistered secret ran the redaction pass empty. Tested in both directions — `id@version` once matched the email pattern, so a run record named its own capability `<redacted:email>`. The operator's live view is redacted from the same `sensitive` declarations an evidence screenshot is, being the sink most likely to sit on a second monitor. `make secrets` scans every commit in CI.

Limits, stated rather than claimed away. Free-text redaction is patterns plus registered literals, so a member *name* is protected only where a capability declares that field `sensitive`, not by shape. Prompt injection is scoped, not solved: a closed action space and policy evaluated *outside* the model, but a model talked into a *permitted* action on a *permitted* target still performs it. Fixture sign-in (`--sign-in`) drives the browser directly — the one documented carve-out, opt-in and scoped to the mock app's fixed credentials. No real credentials or PII exist in this repository.

7. Cuts

Scope the brief did not ask for. §3.6 scopes a co-browsing console out and invites a mock; I built a working one, because control transfer was the part I most wanted to prove end to end — ~5,000 lines against the brief's own preference for "small, correct, well-argued", and the first thing I would cut. **Not built:** continuous pixel streaming (a still frame on connect and after each policed gesture, not co-browsing); a real desktop surface beyond the interface-only UIA stub; multi-operator routing, SSO, audit sign-off.

Stretch goals: one taken seriously. §8 invites at most one or two; I took cross-tenant reuse (§4) — the hardest to fake, and the only one testing whether the artifact is portable or merely a recording. The rest fell out of making that claim checkable: a catalog and tool schema (§2) to invoke a capability by name, `cua eval` to measure portability, `cua catalog approve` to gate what is not yet measured (refusing evidence that does not support it, losing it the moment the artifact is edited), and `cua codegen`, emitting a runnable Playwright test as the same argument aimed outward — CI runs the generated file rather than trusting it. Assisted recovery (`cua replay --assist`) is the one I would have left out: one model call after a deterministic run has failed, dispatched through the chokepoint from `cua.assist`, above both `cua.agent` and `cua.replay` so it cannot make §3's central claim false. A plain replay still makes zero model calls, asserted rather than described.

What the evidence shows. `evidence/discovery/disc-e0aa86b951/` is a genuine LLM-driven run: six model calls, 10,215 tokens, each carrying the gateway's own provider and request id (`make verify-live` re-checks any run; this one predates latency tracing, so reconcile its request ids with the gateway's logs). The artifact compiled from it, `memberdesk.savings_balance@1.0.0`, replays at zero drift for members that run never saw.

Known weaknesses, stated rather than hidden. The compiler reads a value by the label beside it — right on a label/value table, wrong in an n-column grid, where it cannot yet read a *column header* and falls back to a flagged positional descriptor. `raw_input` is classified by what it lands on — a click is hit-tested against the controls' geometry, a key against the focused control — so a click on "Transfer Funds" needs the operator's explicit confirmation; a gesture on something the semantic tree does not expose falls back to its action type's tier. Approval pins the sealed base; a `--base-url` binding is re-derived at the gate and covered only because it fills a declared slot — a tenant overlay that overrides steps has no approval path of its own yet, so an irreversible capability behind one fails closed.

Two failures of the same shape: a check existed, the test was green, and nothing was actually checked. `make invariants` ran import-linter through `python -m importlinter.cli` — not a runnable entry point, so every "mechanically enforced" claim here held only once that became `lint-imports`. And the compiler wrote the discovery host into `entrypoint.url_pattern` verbatim, so `--base-url` substituted nothing, working only on the recording machine — now parameterised at compile time, and pinned under a test.

Then the question was whether the suite could fail at all. A hardening pass wrote about 1,800 edge-case tests — boundary, malformed, hostile and out-of-order input for every subsystem, the command line included — and the defects they exposed were real, and are all fixed: an allowlist that raised on a malformed URL instead of denying it; the headline wrong-action metric ignoring the expected status; a sealed artifact that reloaded as *tampered* after declaring an empty list; an identifier that was all digits one time in 300 and was redacted as an account number in its own trace; one redaction pattern that took minutes on a single long token. `make mutation-check` then breaks the code on purpose —

each of the ten invariants, and a guard per fix — and records whether the suite noticed: 22 of 22 deliberate violations are caught. `make edge-cases` keeps 22 scenarios as saved runs, so the boundary conditions can be opened and read rather than only asserted.

What to look at

The artifact	`evidence/capabilities/`
The handoff, actors and epochs	`evidence/escalation/demo-escalation/trace.jsonl`
All four terminal statuses	`evidence/replay/`, `evidence/escalation/`
The invariants, enforced	`.importlinter`, `tests/invariants/`
The whole story, one command	`make demo`
The numbers behind §3 and §4	`evidence/evals/`
The edge cases, run and saved	`evidence/edge-cases/`
Does the suite fail when it should?	`evidence/tests/mutation-report.md`
One page of results	`evidence/SUMMARY.md`

Repository