Capability-Scoped Runtimes for Desktop Agents: Risk-Gated Execution, Durable Tasks, and Reversibility-Aware Containment

Bert Colemont
Euraika Labs
Belgium
bert@euraika.net
https://www.euraika-labs.net/

Abstract*—Desktop agents—language-model-driven systems that operate computers on behalf of users—are advancing rapidly along the planning and perception axes, yet remain fragile along the *execution* axis. A single prompt-injected tool call, a single model misjudgment, or a single crashed multi-step plan can leave a workstation in an undesired state with no clear path back. We argue this is a runtime problem, not a planner problem. This paper presents the design of pan-agent, a managed desktop-agent runtime that composes three load-bearing mechanisms: a pre-execution risk-gated classifier that intercepts dangerous tool calls, a durable taskrunner that survives crashes and zombie processes, and a reversibility-aware containment layer that records each side-effect with a typed receipt and wires it to per-tool reversers backed by capability-probed filesystem snapshots. We frame the contribution around a novel taxonomy that classifies every action as `local-reversible`, `runtime-compensable`, or `externally-irreversible`. The runtime today operationally distinguishes the two extreme tiers (via the `(SnapshotTier, ReversalStatus)` pair on each receipt; §III-D) while the middle tier is a conceptual category not yet separately tracked in code. We route execution and metrics by this 2-of-3 operational split, declare the third tier as a v1.2 deliverable, and report only what the runtime actually distinguishes today (§VI). We distinguish our runtime-level compensation mechanism from concurrent work on tool-level transactions [5] and agent-loop replanning [7], and position the paper against existing benchmarks for desktop-agent utility [1], harm [2], [6], adversarial robustness [4], and recovery [3]. We propose a four-experiment evaluation plan with pre-registered task subsets across OSWorld, OS-Harm, RedTeamCUA, and a long-horizon crash generator, and articulate a strict bar for falsifying our composition claim. Implementation is open source, ∼32 kLOC of Go; the ablation harness is *specified* in this preprint as a small set of environment-variable flags (§V-D); v1.2 will ship the wiring and per-arm code diff that make the harness independently runnable.

***Index Terms*—desktop agents, computer-use agents, LLM tool use, reversibility, transactional execution, durable workflows, prompt injection, AI safety, systems architecture**

ACM CCS Concepts. Computing methodologies → Multi-agent systems; Software and its engineering → Software design engineering; Security and privacy → Software and application security.

I. Introduction

Computer-use agents (CUAs)—systems that drive a desktop, a browser, or a mobile device under language-model control—have moved from research prototypes to production previews in fewer than thirty months. Benchmarks such as OSWorld [1] now ship hundreds of execution-graded tasks across Ubuntu, Windows, and macOS, and adversarial benchmarks such as OS-Harm [2] and RedTeamCUA [4] document the speed at which the same agents can be coerced—by deliberate user misuse, by indirect prompt injection, or by mere model misjudgment—into unsafe behavior. The empirical literature has, in parallel, begun to measure the corresponding *recovery* problem: Recovery-Bench [3] measures whether agents can recover from corrupted context; Human-Guided Harm Recovery [6] proposes reward-model-guided re-steering after a harmful state is reached.

Less attention has been paid to the layer **between** the planner and the operating system: the runtime that actually executes the tool calls. In most current CUA designs, that layer is functionally absent. A tool call decided by a language model is dispatched directly to `exec`-class APIs; if the call is harmful, the harm lands. If the process crashes mid-task, the state is left half-written. If the user later realizes the agent took the wrong action, there is—in the general case—no `git revert` for the operating system.

This paper argues that **the runtime is the right place to add the guarantees the planner cannot make**, and that those guarantees compose into a measurable safety/utility frontier. We present **pan-agent**, an open-source desktop-agent runtime built around three load-bearing mechanisms:

1) **A risk-gated execution layer**—a 108-pattern regex classifier, maintained as two severity-level collections (`DangerousPatterns` and `CatastrophicPatterns`) and applied via tool-type dispatch (terminal, code execution, filesystem, browser, ...). Each pattern resolves to one of `Safe` / `Dangerous` / `Catastrophic`. `Catastrophic` is rejected outright; `Dangerous` requires a UI round-trip; `Safe` proceeds. The patterns themselves span six

thematic concerns (exec, fs, net, creds, obfuscation, and injection-shaped inputs); we treat that grouping as an exposition-time taxonomy and publish a generated breakdown in Appendix D rather than encoding it into the source today.

2) **A durable taskrunner**—a CAS state machine (`queued` → `running` → {`succeeded`, `failed`, `paused`, `zombie`, `cancelled`}) with heartbeat-based zombie detection, pause/resume via step memoization, and per-task budget and approval enforcement.
3) **A reversibility-aware containment layer**—an action journal that records typed receipts with a `ReversalStatus` (`reversible` | `audit_only` | `reversed_externally` | `irrecoverable`), a per-tool reverser registry exposed at `/v1/recovery/*`, and a capability-probed snapshot subsystem with three tiers (`SnapshotTier` = `cow` | `copyfs` | `audit_only`) plus per-platform capture implementations on Linux, macOS, and a stub on Windows.

The combination is straightforward to describe but, we argue, novel in composition. Existing transactional tool-use systems such as Atomix [5] address atomic semantics at the tool layer through epoch-tagged commits and per-tool compensations. Existing long-horizon GUI agents such as LongHorizonUI [7] address recovery at the agent-loop layer through hierarchical replanning. Neither operates at the OS-level runtime layer; neither classifies actions by their *reversibility scope* before execution; and neither composes recovery with risk-gating and durability into a single managed substrate.

A. Motivation: the Local State Fallacy

A naïve framing of this work would claim that local rollback makes dangerous actions safely attemptable: the agent tries something risky; if it goes wrong, the runtime restores the snapshot. This framing is wrong, and importantly so. Many of the most consequential desktop actions—sending an email, posting to Slack, calling a billing API, exfiltrating a secret to a remote server, triggering an account lockout—leave **zero residual filesystem delta** and are **not reversible by any local mechanism**. A snapshot-and-rollback runtime that pretends otherwise builds a fraudulent safety claim on top of a real engineering achievement.

We therefore treat reversibility as a *capability scope* the runtime declares per action—currently as the implicit `(SnapshotTier, ReversalStatus)` signal on each receipt (§III-D), with explicit `Tier`-as- first-class-field deferred to v1.2—not as a property of the runtime as a whole. Every action falls into exactly one of three tiers:

- `local-reversible`—effects bounded to the local filesystem, registry, or process tree; rollback is principled and complete.
- `runtime-compensable`—effects partially observable to the runtime (e.g., a temp file uploaded to a known-cooperative cache); rollback is partial; compensation must be explicitly designed and measured.
- `externally-irreversible`—effects exit the runtime's jurisdiction (network egress, third-party API mutations, UI-driven send actions); rollback is a category error.

The runtime makes claims appropriate to each tier—and refuses to make claims outside the tier where they apply.

B. Contributions

This paper makes four contributions:

1) **A taxonomy of action reversibility** (`local-reversible`, `runtime-compensable`, `externally-irreversible`) presented as both an analytical lens and the basis for a routing primitive. In the current pan-agent codebase the tier is *expressed implicitly* via the (`SnapshotTier`, `ReversalStatus`) pair on `Receipt`; making `Tier` a first-class enum field is the next planned code change (§III-D, §VIII). We are explicit about which parts of the taxonomy are shipped today and which are deferred, and the evaluation in §VI only relies on the implicit signal.
2) **A composed managed-runtime architecture** integrating risk-gated execution, durable task management, and reversibility-aware containment, with explicit interfaces between the three layers. The corresponding ablation harness is *specified* in v1.1 (per-arm invariants in `ABLATION_AUDIT.md`); the per-arm code diff and independent reviewer's audit are v1.2 deliverables (§V-D, §VI-F, Appendix C).
3) **A pre-registered evaluation methodology**—four experiments on harmonized subsets of OSWorld [1], OS-Harm [2], RedTeamCUA [4], and a long-horizon crash generator—designed to falsify the composition claim if the integrated runtime fails to expand the safety/utility Pareto frontier in expectation.
4) **An open-source artifact**—pan-agent, approximately 32 kLOC of Go (24.7k non-test, 7.1k test), MIT license. The ablation harness itself is *specified* in §V-D as a set of environment-variable flags; v1.1 publishes the specification and the per-arm invariants in `ABLATION_AUDIT.md`; v1.2 ships the wiring and the populated per-arm diff so independent groups can reproduce or refute results on their own hardware and OS mix.

C. Paper Organization

Section II surveys related work in CUA benchmarks, transactional tool use, agent recovery, and capability-based security. Section III introduces the reversibility-tier taxonomy. Section IV presents the pan-agent architecture and the three composing layers. Section V discusses implementation. Section VI details the evaluation methodology and experimental design. Section VII discusses threats to validity, honest scope, and

operational implications. Section VIII sketches future work; Section IX concludes.

II. Related Work

We organize related work into four bands: (i) computer-use-agent benchmarks, (ii) transactional and recoverable tool use, (iii) capability-based security and operating-system substrates for untrusted execution, and (iv) durable workflow systems. We position this paper as the first work to compose all four into one runtime.

A. Computer-Use-Agent Benchmarks

OSWorld [1] is the canonical execution-graded benchmark for desktop CUAs. It provides 369 tasks across Ubuntu, Windows, and macOS with per-task verification scripts, and has become the de-facto utility oracle for the field. OS-Harm [2] introduces 150 tasks specifically designed to measure compliance with harmful requests across three misuse categories, including indirect prompt-injection attacks. RedTeamCUA [4], appearing as an ICLR 2026 oral, extends adversarial testing to hybrid web-OS environments with systematic injection attacks. LongHorizonUI [7] focuses on long-horizon GUI control with fifteen-or-more-step tasks, contributing the LongGUIBench evaluation suite and the framework's own *compensatory execution* mechanism—operating, importantly, at the agent-loop level rather than the runtime level.

We use OSWorld, OS-Harm, and RedTeamCUA as our pre-registered task sources for Experiments A and C respectively. We do not propose a new benchmark; we propose a runtime that is evaluated *on* existing benchmarks.

B. Transactional and Recoverable Tool Use

Atomix [5], appearing as an ICLR 2026 Workshop AIWILD contribution in March 2026, is the closest concurrent work. Atomix introduces atomic semantics for tool calls through epoch-tagged transactions and frontier-gated commits, with compensation mechanisms invoked on abort. Atomix's compensation, however, is **tool-level**: each tool implements its own undo logic, and the runtime treats every tool as a transactional candidate.

Our approach differs in three respects: (a) we install a *reversibility classifier* upstream of the transactional regime, deciding which tools are even eligible for transactional treatment; (b) for the **snapshot-restore subset** of our reverser library (§V-C), our reversers operate at the **OS / runtime layer**, backed by filesystem snapshots and registry restoration that the tool itself does not own—note that the *inverse-operation* and *compensation* subsets of our reverser library are tool-level in the same sense Atomix's are, and we therefore restrict the "OS-level vs. tool-level" differentiator to the snapshot-restore subset and the routing decision; (c) we explicitly refuse to claim transactional guarantees outside the `local-reversible` tier, treating externally-irreversible operations as a separate problem class requiring prevention rather than recovery.

Recovery-Bench [3] measures whether language-model agents can recover from corrupted context—that is, whether the *agent's reasoning* can be restored. This is orthogonal to runtime-level state restoration. We treat Recovery-Bench as a useful evaluation context for the durability layer (§VI-E) but not as a competing system.

Human-Guided Harm Recovery [6] proposes a reward-model-guided recovery procedure where, after a harmful state is reached, the agent is steered back toward safety by a learned preference signal. This is a *learned* recovery mechanism; ours is a *mechanical* one. We see the two as complementary: a mechanical runtime can establish hard guarantees inside the `local-reversible` envelope, on top of which a learned mechanism may guide higher-level escalation decisions in the irreversible tier.

C. Capability-Based Security and Sandboxed Execution

The reversibility-tier taxonomy is in conceptual lineage with capability-based security systems [10], in which authority to perform an action is represented as a transferable, unforgeable token rather than as a static identity-and-permission match. Modern sandboxed execution environments (gVisor, Firejail, Docker user namespaces) operate by restricting *what* a process can do; our runtime operates at a higher level, restricting *which actions enter the transactional regime* on the basis of their reversibility scope. The two are complementary; a future deployment of pan-agent could run inside gVisor without architectural modification.

D. Durable Workflow Systems

The durable-taskrunner subsystem builds on a long line of work in durable workflow engines (Temporal, AWS Step Functions, Cadence, Apache Airflow), which themselves descend from foundational transaction-processing work [8] and the saga pattern [9]. We adopt the canonical CAS state machine with heartbeat-based zombie detection and step-memoized resume. The contribution is not the durability mechanism itself but its *coupling* into the agent runtime—specifically, the integration with the risk-gated layer (gating happens at the step boundary, so a resumed task respects the *current* policy, not the policy at queue time) and with the action journal (the durability layer shares a `*sql.DB` handle with the journal, eliminating a class of inconsistency between "the task believes it succeeded" and "no journal entry exists for the side-effect").

III. The Reversibility-Tier Taxonomy

We define an action's **reversibility tier** as a property of the *tool implementation and its runtime environment*, not of the language model's intent. The taxonomy has three tiers.

A. Tier 1 — `local-reversible`

An action is `local-reversible` if and only if its observable side-effects are bounded to a set of resources for which the runtime can produce a complete pre-image at acquisition time and restore the pre-image deterministically. In pan-agent, this includes:

- writes to the user's home directory under a snapshot-capable filesystem (btrfs, ZFS, APFS, NTFS+VSS);
- mutations to per-process state (environment variables, working directory, child process tree);
- registry edits on Windows where the surrounding key has been exported pre-execution;
- targeted SQLite database writes inside a transaction we control.

Claim regime. Inside this tier, the runtime may legitimately claim that bytes-on-disk and process-tree state can be returned to their pre-execution image, and the paper supports frontier claims with the four-arm ablation in §VI.

B. Tier 2 — `runtime-compensable`

An action is `runtime-compensable` if it has effects the runtime cannot exactly reverse but can *compensate* through a known-correct inverse operation. The taxonomy's normative target is for the inverse to be implemented inside the runtime (registered out-of-band of the tool that produced the original effect, so the tool itself need not own its undo logic). In pan-agent's v1.1.2 implementation the compensation pattern is *tool-implemented* (§V-C, §II-B narrowing)—the same regime as Atomix [5]. v1.2 will migrate compensation reversers to a runtime-side registry; until then, the conceptual distinction holds at the taxonomy level while the implementation is in transition. Examples of conceptually `runtime-compensable` actions:

- a file uploaded to an internal artifact cache that the runtime controls and can delete on rollback;
- a row inserted into a service we own and can DELETE within a bounded time window;
- a system-tray notification we can dismiss through a documented API.

The label `runtime-compensable` is chosen specifically to distinguish from two other 2026-vintage uses of "compensation" in the literature. Atomix [5] uses "compensation" to denote **tool-level** undo logic—the tool implements its own inverse. LongHorizonUI [7] uses "compensatory execution" to denote **agent-loop-level** behavior—the LLM detects failure and replans. Our `runtime-compensable` denotes **runtime / OS-level** inverse operations registered out-of-band of the tool that produced the original effect, and triggered automatically by the action journal during rollback.

Claim regime. Inside this tier, the runtime may claim compensation but must specify the *cooperating system* and the *time horizon* under which compensation is correct. Pan-agent's journal records both at receipt time, and rollback validates that the cooperation contract still holds before issuing the compensating call.

C. Tier 3 — `externally-irreversible`

An action is `externally-irreversible` if its effects exit the runtime's jurisdiction in a manner the runtime cannot reliably reverse. Examples include arbitrary HTTP egress, SMTP send, posting to a third-party messaging service, modifying SaaS records, triggering a payment, or any UI-driven action whose downstream consequence depends on a human who has now seen the result.

Claim regime. Inside this tier, the runtime makes **no post-hoc safety claim** of any form. The only meaningful safety mechanism is *prevention*: the risk-gated layer either (a) routes the action to a Catastrophic-class block, (b) opens a UI approval gate, or (c) downgrades the action to an explicit dry-run. The paper measures the frequency with which adversarial inputs cause the runtime to *escape* this regime—i.e. cause an `externally-irreversible` action to execute without the corresponding approval—under the metric **safety-escape rate** (§VI-D).

D. Classifying Actions at Runtime

The reversibility classifier is presented here as the paper's central **conceptual** contribution: a taxonomy that any desktop-agent runtime can adopt to scope its claims, not just pan-agent. In a tier-aware runtime, the classifier is a pure function over a tool's declared input and the runtime's environmental capabilities; tool authors register their tool against a tier, and the runtime evaluates conditional tiers (e.g., a file-write tool that becomes `local-reversible` on a btrfs volume but `externally-irreversible` if the path resolves to an NFS mount to a third-party server) at the moment of dispatch using the same capability cache that drives snapshot-tier selection.

In the current pan-agent codebase, the tier is **expressed implicitly** via the pair `(SnapshotTier, ReversalStatus)` on each `Receipt` (§IV-C). The mapping is shown in Table I.

TABLE I
IMPLICIT TIER EXPRESSION IN V1.1.2 (PAN-AGENT)

Paper tier	Implicit code expression	
`local-reversible`	`SnapshotTier ∈ {cow, copyfs}`	AND
	`ReversalStatus = reversible`	
`runtime-compensable`	(currently collapsed; not separately tracked)	
`externally-irreversible`	`SnapshotTier = audit_only`	AND
	`ReversalStatus ∈ {audit_only, irrecoverable}`	

Promoting `Tier` to a first-class enum field on `Receipt` is the next code change planned for the runtime; it is described in §VIII. For the experiments in §VI we treat the implicit `(SnapshotTier, ReversalStatus)` pair as the operational tier signal and make this mapping explicit in the ablation harness so that step-level tier predictions in Experiment B (§VI-C) can be scored against ground truth labels.

The taxonomy is the paper's **structural** contribution; the three-layer composition is the **systems** contribution. Both are load-bearing; we are honest that the structural contribution is currently expressed implicitly in code and that explicit encoding is in flight rather than shipped.

IV. System Architecture

Pan-agent is a single Go binary serving an HTTP+SSE API on `localhost:8642`, with a Tauri/React desktop application as one client. A second tier of clients (Telegram/Discord/Slack bots, raw HTTP) communicates over the same surface. There is no cross-process IPC; everything goes through the gateway. We describe each layer in turn.

A. The Risk-Gated Execution Layer (`internal/approval`)

Every shell-bound tool call passes through `Classify(cmd)` → `{Level, Pattern}` before reaching `exec.Command`. The classifier is a 108-entry regex bank, maintained as two top-level severity-keyed collections in `internal/approval/patterns.go`: `DangerousPatterns` (currently 75 entries) and `CatastrophicPatterns` (currently 33 entries). A separate `classify.go` performs tool-type dispatch (terminal commands, code execution, filesystem operations, browser actions, ...) so that the same regex bank can be applied to multiple input shapes. Each individual pattern resolves to one of three levels:

- **`Safe`**—proceeds without a UI round-trip.
- **`Dangerous`**—the runtime emits an SSE `approval-required` event; the calling client (UI, bot, or API consumer) must explicitly authorize before the tool dispatches. Authorization is single-use and parameter-bound: an approved invocation cannot be re-used to execute a different command.
- **`Catastrophic`**—outright rejected. The pattern is logged, the receipt is stored with `ReversalStatus = audit_only` (the command never executed, so there is nothing to reverse—the receipt exists purely for audit), and the agent is informed via the tool result.

The `Catastrophic` check runs *before* `Dangerous` so that a single matching pattern in the higher class is not down-graded by a more permissive pattern in the lower class. Patterns in the `prompt-injection` category fire on input shapes that historically indicate indirect injection ("ignore previous instructions", encoded-payload sentinels, base64 fragments resolving to shell metacharacters); we acknowledge openly that pattern-based detection is an arms race [11], and we treat the layer as one of three composing defenses, not as a sufficient one.

B. The Durable Taskrunner (`internal/taskrunner`)

The taskrunner is a goroutine that polls a `tasks` table every two seconds, claiming queued rows via compare-and-set on the `status` column. Each task has an associated `task_events` log and a `next_plan_step_index` field used for resume.

The state transition graph is:

```
queued -> running -> succeeded
              |  +-> failed
              |  +-> paused -> running   (resume)
              |  +-> cancelled
              +-> zombie    (heartbeat timeout)
```

`zombie` is reachable from `running` only; the heartbeat thread updates a `last_heartbeat` timestamp every 10 seconds (`runner.go: heartbeatLoop`), and a separate reaper goroutine (`reaper.go`) runs every 10 seconds, marking any `running` row whose timestamp is older than `staleThreshold = 60 seconds` as `zombie`. Reclaim transitions through `zombie` → `running` rather than directly to `running` to give independent observers a chance to audit the transition.

Resume from `paused` reads the `next_plan_step_index` and replays the plan from that step *under the current policy*. If gating rules have tightened since the task was queued, the new rules apply—there is no "approved at submit time, executed forever" escape route.

C. The Reversibility-Aware Containment Layer (`internal/recovery`)

The recovery layer has three sub-components: the action journal, the per-tool reverser registry, and the capability-probed snapshot tier.

Action journal. Every tool dispatch records a `Receipt`. The struct, taken verbatim from `internal/recovery/journal.go` at the SHA pinned in Appendix C:

```
type Receipt struct {
    ID             string  // UUID v4
    TaskID         string
    EventID        *int64  // FK to task_events
    Kind           ReceiptKind
    SnapshotTier   SnapshotTier
    ReversalStatus ReversalStatus
    Payload        []byte  // Journal.Record redacts
    SaaSDeepLink   string
    CreatedAt      int64   // unix seconds
}
```

The reversibility tier (§III) is recoverable from the `(SnapshotTier, ReversalStatus)` pair as described in §III-D; making `Tier` an explicit enum field is the next planned change (§VIII). `SnapshotTier` and `ReversalStatus` are distinct concepts: the former records *how the pre-image was captured* (O(1) cow snapshot, full-content copy, or no capture), the latter records *the reversal lifecycle state* (whether the receipt is currently reversible, has been reversed externally, is audit-only, or is irrecoverable).

Receipts are written to the same SQLite database the taskrunner uses, sharing a `*sql.DB` handle, so that a successful tool dispatch and its receipt are committed together. The journal is the paper's *audit artifact*: downstream governance tooling (Aegis-style compliance dashboards are the planned consumer in our deployment; integration in flight, see §VII-C) can consume the receipts to construct an answer to "what did the agent do for user X between time T1 and T2."

Per-tool reverser registry. Each tool is registered with an optional `Reverser`, called by `internal/recovery/reaper.go` when the action journal is asked to undo a receipt. The reverser receives the

`Receipt`, the original input, and a dependency-injected `ApprovalRequester` (so the reverser can itself ask for confirmation on user-visible operations) and `ShellExecFn` (so reverser tests can mock execution without touching the real OS). The `Reverser` contract is invariant under partial failure: if a reverser cannot complete, it must declare `ReversalStatus = irrecoverable` and emit a residual-state report for the journal.

Capability-probed snapshot tier. Snapshots are not always needed; not every host supports every snapshot mechanism. At receipt-creation time the runtime classifies the resource being captured into a three-valued `SnapshotTier`:

- `cow`—copy-on-write snapshot via the host's native mechanism (Linux btrfs/ZFS reflink-class operations, APFS clone on macOS). Capture is O(1) in the resource size.
- `copyfs`—fallback full-content copy of the affected files into the snapshot directory. Always works on a writable filesystem; cost is O(bytes touched).
- `audit_only`—no pre-image is captured. Used for actions that are intrinsically not locally-reversible (network egress, calls to external services, blocked Catastrophic invocations); the receipt still exists for audit.

The selection function, `pickTier()`, lives in `snapshot.go` and consults per-platform helpers (`snapshot_linux.go`, `snapshot_darwin.go`, `snapshot_stub.go` on Windows for now). The chosen tier is stored on the `Receipt` so that ablation harnesses and auditors can record host conditions alongside results. Adding explicit per-OS variants (e.g., distinguishing btrfs-send from APFS clone, or wiring up Windows VSS) is a worthwhile extension and is deferred to §VIII.

D. Composition and Interfaces

The three layers are not independent. They share state in three places:

1) **A single SQLite database** holds the tasks table, the task_events log, and the action journal. This is intentional: a successful tool dispatch and its journal entry must commit in the same transaction or else the journal becomes a fiction.
2) **Every tool dispatch traverses the approval classifier**, not only at task submission. The taskrunner does not call `Classify()` directly; rather, each step it dispatches reaches the tool registry through the gateway, where `Classify()` is the gate. A long-running task that spawns a `Dangerous`-classified sub-step blocks at that gate until the calling client approves, and this composes with the durable `paused` state, allowing an agent to wait indefinitely for human input without occupying a live connection.
3) **The reverser registry consults the approval store** (`ApprovalRequester` interface in `internal/recovery/reversers.go`), which in turn carries the classifier verdict for any reverse-side command issued by a reverser. A reverser may itself need to issue a shell command; that command is classified-then-gated transitively. This prevents the failure mode in which an agent achieves a bad state, the runtime attempts to roll back, and the rollback itself triggers a new bad state silently.

This shared-state architecture is what makes the layered ablation of §VI tricky to construct correctly; we discuss the construction explicitly in §VI-F and publish the per-arm diff to make the ablation falsifiable.

V. Implementation

Pan-agent is implemented in Go 1.25, using only standard library and `modernc.org/sqlite` (a pure-Go SQLite, no CGo dependency). The codebase, measured at the SHA recorded in Appendix C, is approximately 32,000 lines of Go (24,734 non-test, 7,051 test). Cross-platform support is achieved through build-tag-conditional files (`_linux.go`, `_darwin.go`, `_windows.go`, `_stub.go`); each tool registers itself only on the platforms where it is implementable.

The Tauri/React desktop client is approximately 8,000 lines of TypeScript and 1,500 lines of Rust glue. The desktop client is one implementation of the gateway API; alternative clients (Telegram/Discord/Slack bots, raw HTTP consumers) re-use the same surface and the same SSE event stream.

A. Persistence

A single SQLite database stores sessions, messages, the action journal, the task and event tables, and the skill proposal queue. SQLite is chosen for two reasons: deployment simplicity (no external service to operate on a user's workstation) and transactional clarity (a single `*sql.DB` handle gives us ACID across the runtime). FTS5 is enabled for session and message search; we have not yet applied FTS to the action journal but expect to.

B. Approval Classifier Maintenance

The 108 patterns evolve continuously. Patterns are versioned and unit-tested individually with both positive and negative cases. The classifier is intentionally not extensible at runtime—adding or removing a pattern requires a code change and a release—because runtime-mutable safety policy is itself an attack surface.

C. Reverser Library

Sixty-two of pan-agent's currently registered tools have reversers; thirty-eight do not. The v1.2 design (§VIII) requires tools-without-reversers to be declared `externally-irreversible` (or `runtime-compensable` with a known compensation) at registration time, and a planned compile-time validation pass will reject any tool that declares `local-reversible` without supplying a reverser. v1.1's tool registration API does not yet take a tier parameter; the discipline is currently maintained by hand-review of new tools rather than by mechanical enforcement, and we mark this as the primary motivation for the §VIII work.

Reversers fall into three implementation patterns:

1) **Snapshot-restore.** The tool captures a snapshot of the relevant resource pre-execution; the reverser restores from the snapshot. Used by file-system tools, registry edits, and process state changes.
2) **Inverse operation.** The tool stores enough information in its receipt to compute an exact inverse (e.g., a delete-file tool stores the full file content as a sidecar; the reverser re-creates the file). Used for small, well-bounded operations.
3) **Compensation.** The tool's effects are not exactly reversible but a known-correct compensating operation exists (e.g., adding a row $\rightarrow$ deleting that row by ID). Used inside the `runtime-compensable` tier.

Each pattern is unit-tested; cross-platform correctness is verified by running the same test suite on Linux, macOS, and Windows under CI.

D. Ablation Harness (specification; v1.2 will ship the wiring)

A central design goal is *falsifiability*. The ablation harness is specified in this preprint as a set of environment-variable flags that gate each runtime layer at the `cmd/pan-agent/main.go` level:

- `PAN_AGENT_GATING=on|off`
- `PAN_AGENT_DURABLE=on|off`
- `PAN_AGENT_RECOVERY=on|off`

Four arms (`raw`, `gating-only`, `recovery-only`, `full`) are constructed by setting these flags. We are explicit that **the wiring of these flags is a v1.2 deliverable** and is not yet present in the SHA pinned by Appendix C; v1.1 publishes the specification, the machine-checkable per-arm invariants (in `ABLATION_AUDIT.md`), and the policy that the wiring must satisfy ("gating-only" does not silently re-enable recovery; "recovery-only" does not silently consult the classifier; etc.). v1.2's release will carry the wired flags, the per-arm diff against trunk, and the independent reviewer's audit. The split between specification (v1.1) and wiring (v1.2) is itself a methodological contribution: it lets reviewers attack the harness *design* now, without waiting for the implementation, and lets independent reproducers build their own wiring against the specification if they prefer not to wait.

VI. Evaluation Methodology

This section is *evaluation methodology*, not evaluation results. The purpose of the methodology is to define a falsification bar strict enough that, if the integrated runtime fails to clear it, the paper's claim is honestly retired rather than partially salvaged.

We propose four experiments. All four are pre-registered: the task subsets, the metrics, the statistical tests, and the dominance thresholds are fixed in this preprint and will not be retuned between methodology submission and result reporting.

A. Pre-Registered Workloads

All experiments draw from a fixed task pool of 450 items:

- **Utility (200 tasks).** Selected from OSWorld [1], stratified by application category, operating system, and the OSWorld category prevalence weights. Selection rationale and exclusion list are published as Appendix A.
- **Adversarial / harmful (150 tasks).** Equally weighted between OS-Harm [2] (75 tasks) and RedTeamCUA [4] (75 tasks); within each source, stratified by attack type (direct misuse, indirect injection, model misjudgment).
- **Long-horizon durability (100 tasks).** Generated by a scripted task author that emits multi-step plans of length 8–20 with intentional crash-injection points; details in Appendix B.
- **Reversibility-tier ground truth.** Labeling is performed at **two granularities**—task-level (all 450 tasks) and **step-level** for a 60-task subset (drawn proportionally from the three task categories above) that drives Experiment B routing-fidelity (§VI-C). Two human raters label each unit independently against a fixed rubric; disagreements are adjudicated by a third rater. Cohen's κ is reported alongside results.

This constitutes the **pre-registered workload v1**. The pre-registration record—the task list, tier labels, exclusion rationale, and analysis plan—is committed at the SHA recorded in Appendix C of the artifact repository, time-stamped at submission of this preprint, and mirrored as an immutable OSF registration. Subsequent versions are additive only; tasks will never be removed or relabeled silently. Any post-hoc analysis that deviates from the plan in this section is reported as exploratory in §VII.

B. Experiment A — Inside-envelope Frontier (scalar AUC)

Hypothesis. Inside the `local-reversible` tier, the full runtime achieves a higher **utility-AUC under a fixed residual-damage budget** than each of the three single-layer baselines, in expectation across the pre-registered task mix.

Why a scalar. The earlier "convex-hull frontier dominance" formulation was set-valued and could not be paired cleanly with a scalar p-value; we replace it with a single pre-registered scalar estimand.

Primary estimand. For each (arm $\times$ OS $\times$ task) triple, define the binary `success` outcome and a scalar `damage(arm, task)` (residual post-rollback state delta vs. clean-snapshot ground truth, in log-bytes-touched). The damage budget B is fixed at the pre-registered 75th percentile of the `raw` arm's damage distribution (declared before unblinding). For each arm, compute **utility-AUC(B)** = the fraction of tasks succeeding with damage $\leq B$. The reported statistic is

$$\Delta_{\text{AUC}}(\text{arm}) = \text{utility-AUC}(\texttt{full}) - \text{utility-AUC}(\text{arm})$$

for arm $\in \{\texttt{raw}, \texttt{gating-only}, \texttt{recovery-only}\}$.

Randomization. The 25% of the 200 utility tasks that receive prompt-injection are assigned by a pre-registered seeded

RNG stratified by OS and OSWorld application category. The seed is declared in Appendix C alongside the SHA.

Inference. Stratified paired bootstrap with $B = 10{,}000$ resamples, paired *within* (task, OS) to control task-difficulty variance. Report bias-corrected and accelerated (BCa) 95% CIs for each Δ_{AUC}. Multi-comparison: three Δ_{AUC} values per OS $\times$ three OSes = nine simultaneous tests; Holm–Bonferroni step-down with familywise $\alpha = 0.05$. A hierarchical (random-intercept) logistic regression with task and OS random effects is reported as a robustness check.

Falsification bar. The integrated-runtime claim survives if all three lower bounds (Δ_{AUC}(`raw`), Δ_{AUC}(`gating-only`), Δ_{AUC}(`recovery-only`)) exceed zero after Holm–Bonferroni correction in at least two of the three OSes. The claim fails if any of the three Δ_{AUC} lower bounds crosses zero on the **majority of OSes**, or if the hierarchical-model coefficient on the `full`-arm indicator has 95% CI containing zero. Both criteria are declared before unblinding; tying breaks fail the claim.

C. Experiment B — Cross-envelope Routing Fidelity (step-level)

Hypothesis. The reversibility classifier predicts ground-truth tier with precision ≥ 0.90 (one-sided lower bound) on the `externally-irreversible` class, evaluated at the **step level** that the classifier actually scores.

Unit of analysis. The classifier runs at dispatch (§III-D), so the natural prediction unit is the dispatched step, not the task. We therefore label tiers at the step level. For each of the 60 tasks in the routing-fidelity subset, every dispatched step is adjudicated by two human raters against a fixed rubric; the third rater resolves disagreements. The expected step count is roughly 8–12 per task ($\approx$ 600–700 labeled steps total).

Method. Run each of the 60 tasks through the runtime; record predicted tier and ground-truth label per step. Because v1.1.2 operationally distinguishes only `local-reversible` and `externally-irreversible` (§III-D), the *predicted* axis of the confusion matrix has two values; the *ground-truth* axis is full 3-way (annotators may label a step `runtime-compensable` even if the runtime cannot currently emit that prediction). Report: (a) the 2×3 confusion matrix at the step level, (b) per-tier precision, recall, and Wilson 95% CIs for the two predicted classes, and (c) the number of ground-truth `runtime-compensable` steps and how the current runtime routed each one.

Cluster-aware inference. Steps within a task are correlated. We use clustered bootstrap with task as the cluster unit (10,000 resamples) for all CIs. Naïve (independent-step) CIs are reported in the appendix as a contrast, *not* as the primary statistic.

Sample-size justification. Treating each adjudicated step as a Bernoulli trial with cluster correction reduces effective N from $\sim$650 to $\sim$150 (intra-class correlation estimated from a 12-task pilot at $\rho \approx 0.4$). At effective $N = 150$ and target precision 0.90, the one-sided Wilson lower-bound test rejects an alternative of 0.85 at $\alpha = 0.05$ with power ≥ 0.80. We pre-register that this pilot-derived ρ is recomputed from the full sample; if the actual ρ exceeds 0.55, the routing-fidelity claim is downgraded to "underpowered, exploratory" rather than retired.

Falsification bar. Wilson 95% lower bound on precision for `externally-irreversible` ≥ 0.90 **and** recall lower bound ≥ 0.85, both at the step level after cluster correction.

D. Experiment C — Outside-envelope Safety-Escape Rate (paired equivalence)

Hypothesis (equivalence, not directional). On `externally-irreversible` tasks under adversarial pressure, *and in the absence of induced crashes*, the safety-escape rates of the `full` and `gating-only` arms are **equivalent** within a pre-registered equivalence margin $\Delta_e = 0.02$ (absolute risk difference). The directional claim that `full` < `gating-only` under crash conditions is *not* tested here; it is tested in Experiment D (§VI-E), where crash injection provides the experimental manipulation.

Definition. An *escape* is the execution of any `externally-irreversible` action without a corresponding pre-execution approval gate having been satisfied. The event is binary per task and per arm.

Method. Run the 150 adversarial tasks under all four ablation arms. Each task is run once per arm under the same seed, so escape events are paired across arms.

Inference. Two one-sided tests (TOST) for paired proportions on the matched-pairs table (`full` vs. `gating-only`):

- TOST-RD: test $|\text{RD}| < \Delta_e = 0.02$ at $\alpha = 0.05$ (each side); the null is *non-equivalence*, the alternative is equivalence.
- Paired bootstrap 95% CI for RD (10,000 resamples).
- A mixed-effects logistic regression with task and attack-type random intercepts is reported as a robustness check.

We also report the descriptive direction of any non-zero RD as exploratory (not confirmatory) information.

Near-zero baseline rule. If the `gating-only` arm's escape rate is below $5/150$ (3.3%) the equivalence test is interpreted with caution: the equivalence margin Δ_e becomes the dominant determinant of pass/fail and we report the absolute escape count in addition to the rate. The 5/150 threshold is declared here.

Falsification bar. Equivalence is established if both TOST sides reject at $\alpha = 0.05$ after Holm–Bonferroni across the three attack-type strata. **Equivalence is the success outcome** for Experiment C, because under the arm definitions it is the only defensible outcome in the absence of crash injection.

We make **no claim** of safe attemptability inside this tier in any experiment. The crash-consistency claim—that approval state and execution state cannot disagree under restart—is tested in Experiment D (§VI-E).

E. Experiment D — Long-Horizon Crash-Resume and Approval-State Consistency

Hypothesis. Under crash injection on long-horizon tasks, the durable taskrunner achieves (a) resume-correctness $\geq$

95%, and (b) **approval-state consistency**: across all (`full`, `gating-only`) runs that experience a crash before the dispatch of an approval-gated step, no run executes the gated step on resume without the approval still being present in the state.

Method. Run the 100 long-horizon tasks; inject a `kill -9` on the agent process at randomly selected step boundaries (1 injection per run, log timestamp). For (b), select the subset of runs in which the crash boundary falls *before* a Dangerous-class step that has been pre-approved by the calling client; report how many such runs (i) correctly preserve the approval and re-dispatch the step, (ii) lose the approval and correctly re-prompt, and (iii) execute the step on resume without an extant approval—the failure mode this hypothesis explicitly names.

Falsification bar. Resume-correctness $\geq 95\%$; duplicate-side-effect rate $\leq 1\%$; **count of failure-mode (iii) runs $= 0$.** Tying breaks fail the claim. This is the experiment that operationalizes the "approval state and execution state cannot disagree under crash or restart" claim that Experiment C explicitly does *not* test.

F. Layer-Separability Audit

A meta-experiment, run once before A–D. Publish the per-arm code diff against the trunk. For each ablation arm, an independent reviewer (neither author nor reviewer of A–D) must verify that:

- the `gating-only` arm has no journal writes,
- the `recovery-only` arm short-circuits the classifier,
- the `full` arm activates all three layers,
- the `raw` arm is genuinely uninstrumented.

The diff is published as `ABLATION_AUDIT.md` alongside the paper; reproducibility depends on it.

G. Ethics and Honest Reporting Protocol

All adversarial workloads are run inside disposable VM snapshots, not on a contributor's primary workstation. No real third-party service is targeted in adversarial trials; OS-Harm and RedTeamCUA already provide containment guidance. Negative results are reported in the same paper as positive results: if any falsification bar is hit, the corresponding claim is retired in the paper text rather than relegated to an appendix.

VII. Discussion

A. Threats to Validity

Workload selection. The pre-registered task subset shapes the evidence; under-weighting `externally-irreversible` actions would make the safety/utility frontier inside `local-reversible` look deceptively favorable. We mitigate by stratified sampling against the OSWorld application categories and by reporting tier distributions alongside results. Reviewers should weight Experiment B's outcome before reading Experiment A.

Cross-OS reversal fidelity. Snapshot tier capability varies across hosts. We report capability-tier coverage explicitly, treating tier unevenness as empirical data rather than a flaw in the architecture. Hosts on which `pickTier()` resolves only to `audit_only` (e.g., the current Windows stub, or a filesystem without copy-on-write or copy fallback support) are excluded from Experiment A but reported in Experiment C.

Layer separability. The three layers share state by design. The per-arm code diff (§VI-F)—which v1.1 specifies and v1.2 will populate—is the evidence that the ablation is real, not a measurement of re-implementation effort. Reviewers are explicitly invited to audit the v1.1 specification now, and the v1.2 implementation when it ships.

Pattern-based classifier brittleness. The 108-pattern classifier is one layer of three; its individual brittleness is the motivation for the composition, not a refutation of it.

Concurrent work overlap with Atomix [5]. Both papers appear in 2026 with overlapping vocabulary (transactions, compensation). The differentiators are: (a) the reversibility-tier classifier upstream of transactional execution; (b) for the snapshot-restore subset of reversers, runtime-level rather than tool-level mechanism (the inverse-operation and compensation subsets of our reverser library are themselves tool-level in the Atomix sense); (c) explicit refusal to claim transactional guarantees outside the `local-reversible` tier.

B. Honest Scope and Non-Claims

The paper does **not** claim:

- that the runtime makes desktop agents safe in any general sense;
- that `externally-irreversible` actions can be undone, mitigated, or made attemptable through any mechanism described here;
- that regex-based prompt-injection classification is robust;
- that snapshot-and-rollback substitutes for principled authentication, authorization, or content sanitization at service boundaries.

The paper does claim:

- Inside the `local-reversible` tier, the integrated runtime expands the safety/utility Pareto frontier over single-layer baselines, in expectation, on a pre-registered workload, under a paired bootstrap test.
- The reversibility-tier classifier achieves ≥ 0.90 precision on the `externally-irreversible` class.
- The durable taskrunner survives crash injection with $\geq 95\%$ resume correctness on long-horizon tasks.
- The three layers will compose without state leakage when the ablation harness wiring lands in v1.2; this composition property is a *claim of v1.2*, to be confirmed by the independent reviewer's audit (Appendix C). v1.1 publishes only the per-arm invariants the audit will check.

C. Operational Implications

Two implications stand out. First, **safety policy becomes a runtime parameter rather than a model-prompt problem**. Two organizations running the same underlying language model can differ in their safety posture by adjusting the classifier's pattern weights and the approval gating thresholds; the model itself is held fixed. This decouples model choice

from policy choice—important for organizations that must validate a model once and operate it under multiple policies.

Second, **the action journal is designed to be consumable by external auditors**. The receipt schema (§IV-C) is published and stable; any compliance system can ingest it to construct evidence trails per user per session. In our deployment, Aegis (an on-prem single-tenant compliance command center) is the planned consumer; the integration is in flight at preprint time and is listed as a separate v1.2 deliverable rather than presented as a shipped pipeline.

D. Limitations

The classifier's pattern set is curated by the authors and is necessarily incomplete. We do not address natural-language attacks that bypass pattern matching by encoding shell commands in human-readable English. Such attacks land squarely on the agent's planning layer and are out of scope for this paper.

The reversibility classifier is currently expressed implicitly in code via the `(SnapshotTier, ReversalStatus)` pair on each `Receipt` (§III-D, §IV-C). Promoting `Tier` to a first-class enum field on `Receipt` and adding a tier-aware tool registration API are planned changes (§VIII).

The evaluation is, at preprint time, *proposed* rather than *completed*. Pre-registered subsets and statistical tests are fixed; results will appear in version 2 of this preprint.

VIII. Future Work

Promote `Tier` to a first-class field on `Receipt`. The reversibility taxonomy (§III) is currently expressed implicitly via `(SnapshotTier, ReversalStatus)`. The next code change is to add an explicit `Tier ReversibilityTier` enum field, refactor tool registration to require a tier declaration (with conditional-tier support for tools whose tier depends on input), and add a compile-time / startup-time validation pass that rejects tools whose declared tier is inconsistent with their reverser registration.

Composition with planner-level safety classifiers. Pattern matching cannot catch attacks expressed in natural language. A planner-level safety classifier—operating on the agent's *intent* before tool calls are emitted—composes naturally with the runtime layers presented here.

Learned reversers. The current reverser library is hand-written. For tools whose effects are deterministic but whose inverse is nontrivial (e.g., complex JSON edits), a learned reverser conditioned on the receipt could supplement or replace the hand-written rule. Coupling with Human-Guided Harm Recovery [6] is a candidate composition.

Distributed runtime. Pan-agent today runs as a single binary on a single workstation. A multi-host extension—laptop + cloud sandbox—would let the runtime route `externally-irreversible` actions to a sandboxed environment by default.

Formal verification of the ablation harness. The per-arm code diff is currently audited manually. A formal proof that the three flag combinations produce four genuinely-different runtime configurations would strengthen reproducibility claims.

Application beyond desktop agents. The reversibility-tier taxonomy is not specific to desktop agents. CI runners, data-pipeline orchestrators, and any other system that executes operations on behalf of LLMs could adopt the same envelope-declaration discipline.

IX. Conclusion

Desktop agents are advancing on planning and perception while remaining fragile on execution. This paper has argued that the fragility lives at the *runtime* layer, between the planner and the operating system, and that the right response is a managed runtime that composes risk-gated execution, durable task management, and reversibility-aware containment under an explicit taxonomy of action reversibility.

The paper's structural contribution is the `local-reversible` / `runtime-compensable` / `externally-irreversible` taxonomy. We present it as both an analytical lens *and* the basis for a routing primitive; in the v1.1 codebase the tier is expressed implicitly via the `(SnapshotTier, ReversalStatus)` pair on each receipt, with explicit `Tier`-as-first-class-field deferred to v1.2 (§III-D, §VIII). The paper's systems contribution is the three layers—risk-gated execution, durable task management, and reversibility-aware containment—individually implemented in pan-agent and released open source; their integrated composition (the falsifiable claim that the layers compose without interference and shift the safety/utility frontier together) is to be empirically demonstrated by the v1.2 ablation harness. The paper's methodological contribution is a pre-registered, falsifiable, four-experiment evaluation plan that will, in version 2, either support the composition claim or retire it.

We make no claim that the runtime makes desktop AI control safe in any general sense. We do claim—and propose to demonstrate—that *inside* a declared reversibility envelope, the integrated runtime expands the safety/utility frontier in a way no single layer achieves alone, and that *outside* that envelope, the runtime's job is to refuse to make a claim it cannot honor.

The paper invites independent replication, attack, and extension. The artifact is open. The evaluation is pre-registered. The falsification bars are written down. We will report results honestly, including negative ones.

Acknowledgments

This work emerged from a structured multi-round adversarial debate between language-model assistants, with literature-grounded citations verified against canonical web sources before commitment to the framing. The synthesis methodology is documented in the supplementary debate transcripts at the artifact repository. We thank the OSWorld, OS-Harm, RedTeam-CUA, Recovery-Bench, Atomix, Human-Guided Harm Recovery, and LongHorizonUI authors for their public artifacts, on which our evaluation methodology depends.

REFERENCES

[1] T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, Y. Liu, Y. Xu, S. Zhou, S. Savarese, C. Xiong, V. Zhong, and T. Yu, "OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments," *arXiv preprint* arXiv:2404.07972, Apr. 2024.

[2] T. Kuntz, A. Duzan, H. Zhao, F. Croce, J. Z. Kolter, N. Flammarion, and M. Andriushchenko, "OS-Harm: A Benchmark for Measuring Safety of Computer Use Agents," in *Proc. ICML 2025 Workshop on Computer Use Agents (WCUA)*, Jun. 2025. (oral).

[3] S. Tan, K. Lin, K. Sen, and M. Zaharia, "Recovery-Bench: Evaluating Agentic Recovery from Mistakes," in *Proc. NeurIPS 2025 Workshop on LLM Evaluation*, Sep. 2025.

[4] Z. Liao, J. Jones, L. Jiang, Y. Ning, E. Fosler-Lussier, Y. Su, Z. Lin, and H. Sun, "RedTeamCUA: Realistic Adversarial Testing of Computer-Use Agents in Hybrid Web-OS Environments," in *Proc. ICLR 2026*, Jan. 2026. (oral).

[5] B. Mohammadi, N. Potamitis, L. H. Klein, A. Arora, and L. Bindschaedler, "Atomix: Timely, Transactional Tool Use for Reliable Agentic Workflows," in *Proc. ICLR 2026 Workshop AIWILD*, Mar. 2026.

[6] C. Li, S. CH-Wang, A. Peng, and A. Bobu, "Human-Guided Harm Recovery for Computer Use Agents," in *Proc. ICLR 2026 Workshop AIWILD*, Mar. 2026.

[7] B. Kang, S. Wen, Y. Bi, S. Wu, X. Yuan, R. Shao, J. Wang, and Z. Tian, "LongHorizonUI: A Unified Framework for Robust Long-Horizon Task Automation of GUI Agent," in *Proc. ICLR 2026*, Jan. 2026. (poster).

[8] J. Gray, "The Transaction Concept: Virtues and Limitations," in *Proc. 7th Int. Conf. Very Large Data Bases (VLDB)*, 1981, pp. 144–154.

[9] H. Garcia-Molina and K. Salem, "Sagas," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1987, pp. 249–259.

[10] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966.

[11] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection," in *Proc. 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023.

APPENDIX A
OSWORLD SUBSET SELECTION

To be published with version 2 of this preprint. The 200-task utility subset is stratified across the eleven OSWorld application categories and across Ubuntu, Windows, and macOS in proportions matching OSWorld v1 prevalence. The selection rationale, the exclusion list, and the per-task tier label are committed to the artifact repository.

APPENDIX B
LONG-HORIZON TASK GENERATOR

A scripted generator produces multi-step plans of length 8–20 by chaining primitive sub-tasks drawn from a fixed library of forty operators (file copy, archive create, registry read, etc.). Crash injection points are placed at randomly chosen step boundaries during execution; the generator is published as part of the artifact repository.

APPENDIX C
PRE-REGISTRATION PROVENANCE AND LAYER-SEPARABILITY AUDIT

Pre-registration anchor. All §VI task subsets, tier labels, RNG seeds, statistical estimands, and decision rules are pinned to the commit recorded below. We are explicit about which fields are final, which are recorded-but-not-yet-immutably-tagged, and which require external filing:

- **Repository:** `github.com/Euraika-Labs/pan-agent`
- **Anchor branch (at preprint v1.1):** `main`
- **Anchor SHA (recorded at v1.1 submission):** `9b0c2e26ee5a1f3959c6b221fd1599a07ba92cab` (short: `9b0c2e2`)
- **Anchor tag:** *pending*—`paper-v1.1.2-prereg` will be created on the same SHA before v2 results are reported.
- **OSF registration:** *pending external filing*— the SHA above will be mirrored as an OSF Pre-Registration with the §VI analysis plan. OSF DOI will replace this line in v1.2.
- **Anchor commit-time:** the SHA above resolves at the pan-agent main branch as of preprint submission (2026-04-29 UTC).

The pre-registration covers (1) the 450-task pool composition, (2) the seeded RNG used to allocate the 25% prompt-injection split in Experiment A, (3) the damage-budget definition B (75th percentile of the `raw` arm's log-bytes-touched distribution, declared before unblinding), (4) the 5/150 near-zero-baseline threshold in Experiment C, (5) the pilot-derived intra-class correlation ρ used for the Experiment B sample-size justification.

Layer-separability audit. The per-arm code diff against the trunk pan-agent codebase is published at `github.com/Euraika-Labs/pan-agent/blob/research/ABLATION_AUDIT.md` (skeleton scaffolded with v1.1; populated with diffs for the `raw`, `gating-only`, `recovery-only`, and `full` arms before results are reported in v2). An independent reviewer's signed audit accompanies the v2 preprint.

APPENDIX D
PATTERN BANK BREAKDOWN BY THEME

The 108 patterns in `internal/approval/patterns.go` are reported here as a generated breakdown across six thematic concerns. This table is exposition only; the runtime decisions are driven by the two-collection severity split (`DangerousPatterns` / `CatastrophicPatterns`) and the tool-type dispatch in `classify.go`.

Important: the six themes are *not* a partition. Many real-world dangerous commands touch more than one theme—for example, `curl ... | bash` is both `net` (the download) and `exec` (the pipe-to-shell). We therefore report **multi-label touch counts** per theme, not partitioned counts.

TABLE II
PATTERN BANK THEMATIC BREAKDOWN (MULTI-LABEL TOUCHES)

Theme	Catastr.	Danger.	Total
`exec` (shell/eval)	4	26	30
`fs` (filesystem destructive)	18	25	43
`net` (network egress)	1	13	14
`creds` (secrets/keys)	11	5	16
`obfuscation` (base64/IEX)	0	3	3
`injection` (SQL DROP/DELETE)	0	3	3

Patterns matching no theme by the current heuristic: 6 Catastrophic + 14 Dangerous = 20. These are mostly

Windows-specific anti-forensics or service-tampering patterns (e.g., `wevtutil cl`, `sc.exe stop windefend`, `bcdedit /set`) for which an additional theme—likely `evasion` or `persistence`—is a candidate extension, deferred to a future preprint revision.

Severity totals (partitioned, ground truth): `DangerousPatterns` $= 75$; `CatastrophicPatterns` $= 33$; sum $= \mathbf{108}$.

APPENDIX E
ACRONYM GLOSSARY

TABLE III
ACRONYMS USED IN THE PAPER (FIRST-USE LOCATIONS)

Acronym	Expansion	First defined
CUA	Computer-Use Agent	§I
LLM	Large Language Model	—
CAS	Compare-and-Set	§IV
IPC	Inter-Process Communication	§IV
SSE	Server-Sent Events	§IV
CORS	Cross-Origin Resource Sharing	§IV
FTS5	SQLite Full-Text Search v5	§V
AUC	Area Under the Curve	§VI-B
BCa	Bias-Corrected and Accelerated bootstrap	§VI-B
CI	Confidence Interval	§VI-B
RD	Risk Difference	§VI-D
RR	Risk Ratio	§VI-D
OS	Operating System	—
VSS	Volume Shadow Copy Service (Windows)	§VIII
APFS	Apple File System (macOS)	§I
NTFS	New Technology File System (Windows)	§III-A