Garbage Collecting the External Effects of AI Agents

Jinhao Hu
jinhaohu@mpi-sws.org
Max Planck Institute for Software Systems
Saarbrücken, Germany

Ashvin Goel
ashvin@eecg.toronto.edu
University of Toronto
Toronto, Canada

Laurent Bindschaedler
bindsch@mpi-sws.org
Max Planck Institute for Software Systems
Saarbrücken, Germany

Abstract

AI agents open pull requests, allocate cloud resources, charge cards, and post status updates to team channels. They increasingly delegate their work to trees of sub-agents and helper processes that the model shapes at runtime. When these computations end before their work is done, for example because the user cancels them, their token budget runs out, or they crash, their processes stop, but their effects remain. We argue that managing these effects is a garbage collection problem over external services. Remove an effect too early, and something that depends on it breaks. Never remove it, and it leaks. Unlike memory, this heap belongs to other services, the code that creates effects is chosen at runtime and cannot be instrumented, other people start depending on effects without telling anyone, and every part of the system can crash.

We present effect recovery, an abstraction for disposing of the external effects a computation leaves behind when it ends. It guarantees that a computation leaves no dangling effects and no silent leaks. It records every effect before it happens and decides what happens to each one after the computation ends. Requests leave the computation only through a monitor, which records each one before forwarding it. Because the computation can stop at any point, a controller that outlives it works from this record. It removes an effect only when removal is known to be safe, and otherwise keeps it or escalates it to a human. Effects that no later step needs, such as the final pull request, wait until the user completes the task, so an aborted task never creates them. Our evaluation shows that our implementation of effect recovery mediates real agent workloads without meaningfully reducing their task success or adding noticeable latency, and that its guarantee holds when computations fail at any point.

1 Introduction

AI agents increasingly act on services outside the machine they run on by opening pull requests, allocating cloud resources, charging cards, and posting to team channels. They also delegate. A single task typically grows into a tree of sub-agents and helper processes that can each call a service provider or start further processes. The model shapes this tree dynamically while the task runs.

These computations often end before their work is done for ordinary reasons, for example because the user cancels the task, its budget runs out, it times out, or its host restarts. Budgets and deadlines alone stop many runs. For example, 31% of SWE-agent's runs on SWE-bench exhausted their $4 cost budget [62]. In AgentBench, running out of interaction rounds is the predominant cause of failure [37]. GitHub's Copilot cloud agent stops any session after 59 minutes, finished or not [23]. The model side fails too. An agent that fans out into many sub-agents can hit the LLM provider's rate limits, so the run stalls or fails mid-task.

When such a computation ends, its processes stop, but the changes it made at external services remain. These can be a pull request, a test customer, a half-applied deployment, or a status message already posted to the team's channel. Neither the user nor the system knows which computation created these effects, or whether they should be kept. Claude Code, which lets agents start nested sub-agents, recently added limits on nesting depth and spend [8], but these limits bound only what a computation consumes.

We present *effect recovery*, which guarantees that a computation leaves no dangling effects and no silent leaks. Every request leaves the computation through a *monitor*, like a reference monitor [7, 51]. The monitor records the request in a durable *ledger* before forwarding it. A *controller* that outlives the computation reads the ledger and ensures that each effect reaches an intended outcome, one of the settled or owned outcomes of §3.3, whether or not the computation finishes successfully. It removes an effect only when the removal is known to be safe. Otherwise, it keeps the effect or escalates it to an owner outside the computation, such as a human or a supervising agent. Effects that no later step needs, such as a pull request that is the task's final deliverable, are *held back*, recorded but not sent, until the user completes the task, so an aborted task never creates them. In our adapters, only pull requests and subscriptions are held back. Holding an effect back follows output commit [22, 41], which releases output only once it is known to be final. Here, the user's decision to complete the task is what makes it final. Small per-provider *adapters*, a few dozen lines of code per *effect type*, such as a GitHub branch or a Stripe customer, tell the monitor and the controller what each request does and what the provider guarantees about removing its effect.

Providing these guarantees raises three challenges. First, the ledger cannot be based on what the agent reports. An agent reports only the effects still in its context. We measured this on agents from four model families. Once its older

turns were summarized, an agent listed only half of its effects and none of the effects a sub-agent made for it. Its helper processes, such as a `curl` command, a shell script, or a package installer, also make calls it never sees, which is why the monitor records each request at the point where it leaves the computation. Second, removal is risky, because the references that make a removal unsafe often live at the provider or with other people, where the controller cannot see them. Third, the computation can stop at any point. A provider can also execute a request even though its response is lost on the way back. Both the ledger and the controller's decisions must survive these failures.

We argue that this is a garbage collection problem. Each effect is like an object in a heap that lives at external services and must be reclaimed once nobody needs it. Reclaiming it too early leaves something dangling, as when deleting a leftover branch closes a pull request that a teammate opened from it. Never reclaiming it leaks it. These are the two failures our guarantee rules out. A garbage collector does not rely on the program to report what it allocated. Similarly, our controller does not depend on the computation but uses the ledger that the computation cannot skip, and unlike a collector it keeps working after the program has stopped. It tracks who still needs each effect through *claims* in the ledger, much as a distributed collector tracks which clients hold a reference [12].

Existing systems address parts of this problem. Durable execution engines such as Temporal, Restate, and DBOS [20, 49, 58] survive crashes and make programmer-written compensations reliable for work that enters their API, but a script's subprocess can bypass that API. Kubernetes owner references and finalizers [60] govern the cleanup of objects in Kubernetes's own API, and external calls that the API does not represent fall outside that mechanism. Transactional outboxes [50] hold messages until commit, but only for code that routes messages through them. Tool-call authorization [52, 66] decides whether a call may happen, and does nothing about the effect afterward. Each of these covers only the work that passes through its own interface. None takes responsibility for effects created outside it. Finally, OpenAI's Codex cloud, a hosted coding agent, avoids the problem by default. It runs each task in a cloud container with the agent's internet access turned off [43] and returns its work, such as a proposed pull request, for the user to review [45]. Once users turn internet access on for an environment [43], or let an agent call external tools through OpenAI's platform [44], the problem resurfaces. A disconnected default can suit some coding tasks, but an agent that charges a card, posts to a channel, or provisions infrastructure has to reach those services while it runs.

Evaluation. We built two implementations. The first runs the monitor, a durable ledger, and the controller as user-level Python processes behind a container's network confinement, with adapters for GitHub, Stripe, Slack, and AWS. The second enforces the same boundary in the kernel. We also model-check the contract's core invariant over a simplified model. Our evaluation answers five questions. First, do the options people use today leave dangling effects or silent leaks where effect recovery leaves none? Second, is each of its components necessary? We remove one component at a time under a scenario built to exercise it. Third, does the guarantee hold when the monitor or the controller is killed, or a response is lost, at any step? Fourth, do real agents still complete their tasks at a rate comparable to running without the system? We fixed a two-point margin for this comparison in advance. Finally, how many effects does the controller dispose of without a person, and what does mediation cost?

Results. Removing any one component produces the failure it exists to prevent, while the complete system never fails the guarantee in 380 trials. Under 4,400 fault injections at 22 points on each of macOS and Linux, no effect is left dangling or leaked. None of the 22,000 effects in these trials that needed a decision is disposed of incorrectly. The monitor adds 3.8 to 4.7 ms per call against emulators and 2.4 to 4.4% of a call's round trip against real providers. One monitor sustains 565 calls per second across 500 concurrent domains. Real agents complete 1.1 percentage points fewer tasks under effect recovery. The estimate is inside the two-point margin we fixed in advance. An attempt takes 0.1 more turns and 3 s more on average, with the same tokens. An abort leaves 42% of real agents' effects settled without a person.

Contributions. This paper makes three contributions.

- We identify the lifetime of agents' external effects as a garbage collection problem and define effect recovery and its contract.
- We present its design and implementation.
- We show that each component is necessary within this architecture, that tested existing constructions do not reach the same completeness without equivalent machinery, that agents complete nearly as many tasks, and that the cost is acceptable.

2 Background and Motivation

How agents reach providers. An agent's requests reach a provider along several paths. Some go through tools that the agent's harness exposes and can log. Many do not. The agent runs command-line clients such as `gh` or the Stripe CLI, writes scripts that call a provider's SDK, and starts helper processes that send raw HTTP requests with `curl`. Providers also differ in what happens when something is removed. Some refuse to delete a resource that others still use, e.g., S3 refuses to delete a bucket that still holds objects [3]. Some check a condition in the same operation as the delete. Others delete whatever they are asked to, e.g., GitHub deletes a branch even while a pull request is open on it. Finally, other people work in the same accounts and repositories, so

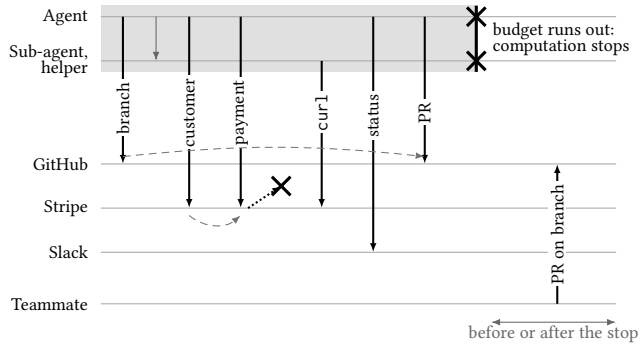


Figure 1. One run of the running example. Dashed arcs mark the only fixed orders. The dotted arrow is the lost response.

a leftover branch or test customer can become someone's dependency without anyone telling the agent.

Running example. An on-call engineer asks a coding agent to fix checkout failures that started after a payment SDK upgrade. The only deliverable the engineer expects is a pull request with the fix. Other engineers' agents use the same test accounts and credentials. Figure 1 shows one run. The agent creates a branch, a customer and an uncaptured payment in the payment provider's test mode to reproduce the failure. It delegates dependency analysis to a sub-agent, whose helper creates a second customer by calling the provider directly with `curl`. The agent then posts a status update to the team's Slack channel and opens the pull request. Before the task finishes, the token budget runs out and the computation is stopped, possibly just after the provider executed the payment and before its response arrived. A teammate opens a pull request of their own from the agent's branch, either before or after the stop. The agent chose these steps and their order while it ran. Another run might skip the status update or create a third customer. Only data dependencies fix an order, since the payment needs the customer's identifier and the pull request names the branch.

What today's options do. Today's options each miss part of this run. Doing nothing leaves the branch, the customer, the payment, and the status update behind. Cleaning up by hand later risks deleting the branch under the teammate's pull request. The agent cannot clean up after itself, because the budget stopped it and because its own list omits the helper's customer. A durable execution engine [49, 53, 58] journals the steps it runs and compensates them after a crash, but the helper's `curl` call runs inside a script and never becomes a step. Tagging every resource and sweeping by tag misses whatever the helper created without the tag. Giving each run its own credentials lets provider records attribute effects [6], but here the runs share credentials. None of these options can keep the pull request of a stopped run from reaching GitHub or check whether the teammate is using the branch before removing it. A sandbox with the network turned off [42, 61] avoids all of this only by not doing the task. Resource managers such as Kubernetes [60], Crossplane [18], and Terraform [26] manage only resources declared through their own interfaces. Transactional outboxes [17, 39, 50] publish only the messages that code routes through them. Neither sees the helper's call.

Why this is hard. The run shows the three problems of §1. The record of what the run did cannot come from the agent, whose summarized context has lost half of its effects and which never saw the helper's `curl` call. Whether an effect may be removed depends on facts the run does not hold, such as the teammate's pull request at GitHub. The customer is scratch and the pull request is the deliverable, so keeping everything leaks and deleting everything removes the deliverable. The agent cannot tell them apart for us, since its own success is in question. Finally, the stop can fall between any two steps, and a provider can execute a request whose response is lost, or even execute it after a lookup has come back empty. None of this needs an adversary. A summarized context, an ordinary model error, a budget, or a teammate is enough.

3 The Effect Recovery Abstraction

Any system that takes responsibility for these effects needs four things. It needs a complete record of every effect, taken where each request leaves the computation instead of reported by the agent. It needs to track who still depends on each effect and never remove one that anyone might still need, including a dependent it cannot see. It needs a rule for when removing an effect is safe, with a safe fallback when it is not. It should avoid creating effects the task may not keep.

3.1 Model and Assumptions

A *computation* is an agent together with every sub-agent and helper process it starts. A *provider* is an external service that the computation acts on, such as GitHub, Stripe, or Slack. An *effect* is a mutation that a request from the computation makes, or would make, to state at a provider. Its *effect type* names the kind of mutation per provider, such as a GitHub branch or a Stripe customer. Reads are not effects. A *domain* is one computation together with every effect it causes. A domain is *live* from its start until it is *completed* or *aborted*. The *user* is whoever starts the task and decides whether it succeeded. A domain can also start a child domain for delegated work. When a child completes, the claims on its deliverables pass to its nearest live ancestor.

Effect recovery consists of four trusted parts that run outside the computation: the monitor, the ledger, the controller, and the per-provider adapters. It also trusts the launcher, which starts each computation in its domain and holds the domain's lease. We do not trust the computation to report or decide anything about its effects. It may forget or misreport what it did, start processes that call providers directly or ignore proxy settings, and stop at any time. We assume that

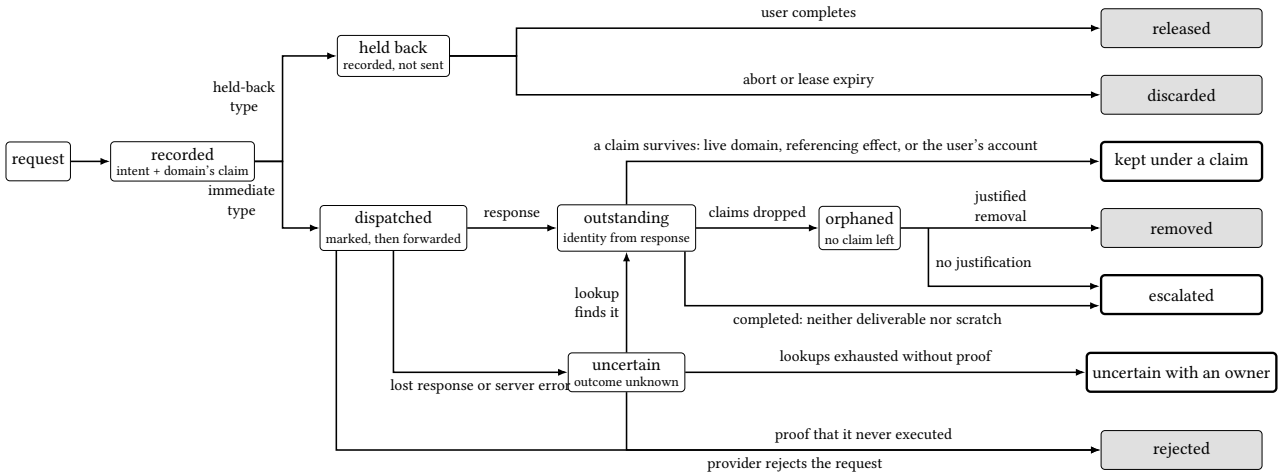


Figure 2. Effect lifecycle. Shaded outcomes are *settled*, meaning nothing remains to be done. Outcomes with a thick border are *owned*, meaning a named party is responsible for what remains. An orphaned effect first waits a short grace period (not shown). §4.2 defines intents, dispatch marks, and outstanding effects. A released effect is sent like any other request, so it can still be rejected or become uncertain (§4.8).

it is not malicious (§7). The trusted parts behave as specified but may crash. Providers follow their documented API semantics. Other people act on the same provider state at any time and can come to depend on an effect through paths that effect recovery never sees.

3.2 Ending a Domain

The computation holds no provider credential, so it can reach a provider only through the monitor. When the task has succeeded, the user *completes* the domain by naming its *deliverables*, the effects that the task was meant to produce, such as the pull request. The launcher holds a *lease* on the domain and renews it until the domain ends. If the lease expires, because the launcher died or its host restarted, the domain is aborted.

What adapters declare. Each provider's adapter declares three things per effect type. It declares whether the type is *held back*, meaning recorded but not sent to the provider until the user completes the domain. Every other type is *immediate*. It declares whether the type is *scratch*, meaning intermediate work that should not outlive a successful task, such as the test customer. It declares the conditions, if any, under which the controller may remove an effect of the type (§4.6). Which types to hold back is a policy choice. We use one simple default (§4.5) and do not claim it as a contribution.

3.3 Claims, Outcomes, and the Contract

A *claim* is a record in the ledger stating that someone still needs an effect. Claims have three kinds of holders. A live domain holds a claim on each of its effects. An effect holds a claim on every effect it requires, e.g., the payment holds one on the customer it charges. After the user completes a domain, the *user's account* holds a claim on each deliverable and released effect. An effect that no claim holds is *orphaned*.

Figure 2 follows an effect from its request to one of seven outcomes. An outcome is *settled* when nothing remains to be done. The effect was *released* when the user completed its domain, *discarded* before any provider saw it, *removed* by the controller, or *rejected* by the provider. An outcome is *owned* when a named party is responsible for what remains. The effect is *kept* under a claim, *escalated* to its owner, or *uncertain with an owner* because nobody can establish whether its request executed. The owner is the user unless the deployment names someone else.

Why this works. The abstraction treats the lifetime of effects as garbage collection. Effects are the objects and claims are the references. Live domains and the user's account are the roots, and the controller is the collector.

Unlike a memory collector, the controller cannot see every reference. Reclaiming an effect is also a request to a provider, which may refuse it, delete dependents along with it, or lose the response. When the budget stop aborts the running example's domain, the controller deletes the bot's status message and cancels the uncaptured payment. It escalates the branch and both customers, because GitHub and Stripe delete such objects even while others depend on them [25, 55].

The contract. Effect recovery guarantees four properties.

P1 Complete recording. Every request from the computation that changes provider state passes the monitor and is recorded durably, attributed to its domain, before it is forwarded, and each recorded request is forwarded at most once.

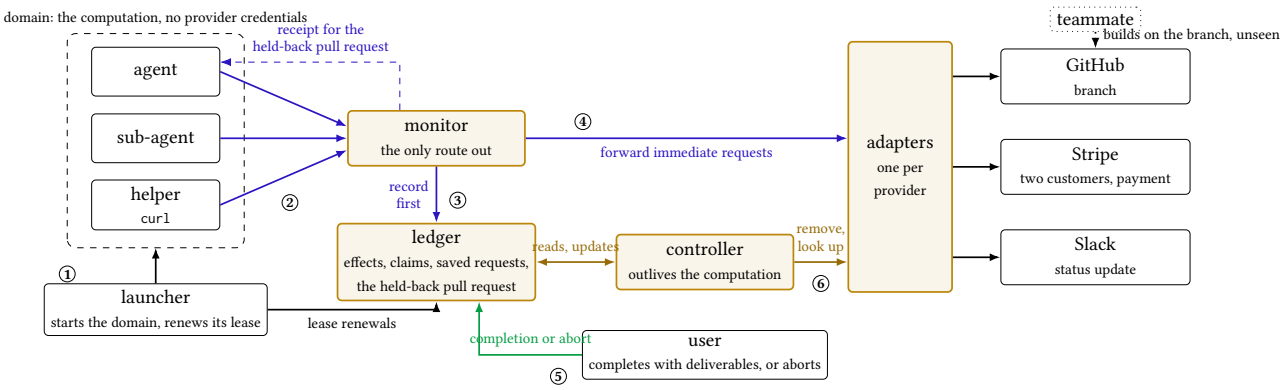


Figure 3. One run of the running example. Blue arrows are the computation's requests, gold arrows the controller's work, and the green arrow the user's decision. The teammate's dependency on the branch is invisible to the monitor. Numbers mark the steps of §4.1.

- **P2 Outside decision.** Only the user or an expired lease ends a domain. The user completes it by naming its deliverables or aborts it. The computation can do neither.
- **P3 Complete disposition.** If the controller eventually runs and providers eventually answer, each recorded effect reaches exactly one settled or owned outcome.
- **P4 Safe removal.** The controller removes an effect only while no claim holds it, and only under one of the justifications of §4.6. Every other effect is kept or escalated.

Figure 2 shows where each property applies. P1 covers the step from request to recorded. P2 covers the transitions that complete or abort a domain. P3 states that every recorded effect reaches one of the seven outcomes on the right. P4 governs the step from orphaned to removed. Undo is attempted wherever a justification allows it, but it is not guaranteed.

4 Design

4.1 Overview

Figure 3 follows the running example. (1) A launcher starts the computation in a domain whose only route to the network leads to the monitor, and takes the domain's lease. (2) Every request, including the helper's `curl` call, reaches the monitor. (3) The monitor asks the provider's adapter which effect type the request creates and records it in the ledger, with the domain's claim on the new effect, before doing anything else. (4) It forwards the branch, the customers, the payment, and the status update. It holds back the pull request and answers it with a receipt, so the computation can continue. The payment's response is lost, so the ledger marks the payment uncertain. (5) When the budget runs out, the launcher aborts the domain for the user. (6) The controller reads the ledger and brings every effect to an outcome through the adapters (§3.3).

4.2 The Monitor and the Ledger

Observation at the network. The helper's `curl` call never appears among the agent's tool calls. Hooks around the agent's tools [9] see only the calls made through them [66], which breaks complete mediation [51]. The monitor therefore sits where requests leave the computation, as the only destination the domain can reach. The computation holds a domain credential in place of any provider credential. The monitor resolves it to the domain on every request and substitutes the real credential only when it forwards. The monitor, the ledger, and the controller can run in user space behind a container's network confinement, or behind a boundary that the kernel enforces (§5).

Classification. For each request, the monitor asks the provider's adapter, a module it loads, whether it is a read or creates an effect of some type, and which recorded effects it references. Reads pass through unrecorded. A mutating request of an unrecognized type is recorded under an unknown type, which the controller later escalates. A gap in an adapter can thus cause an escalation, but it never leaves an effect unrecorded. A request that changes or deletes a recorded effect is recorded as a new effect that references the original.

Record before forwarding. For each mutating request, the monitor writes an *intent* to the ledger in one transaction. It holds the saved request, the domain's claim on the new effect, and a claim on each referenced effect. The monitor then durably marks the request as dispatched in a second commit and only then forwards it. When the response arrives, the adapter extracts the resource's identity, such as the branch name, and the effect becomes *outstanding*. A rejection makes it rejected. A response without an identity, a transport error, or a server error makes it *uncertain*. This order makes P1 hold under crashes.

The ledger. The ledger is a durable, transactional store of domains, effects, claims, saved requests, and an event log of every effect's state transitions. Each effect carries a version that every state change checks and increments, so the monitor and the controller never change one effect at the same time.

4.3 Adapters

An adapter holds one provider's specific knowledge. It classifies each request by effect type, extracts identities and references from real requests and responses, and states whether a reference requires the referenced effect to persist. It implements removal and a lookup that finds the effect of a request whose response was lost. It declares the provider facts the controller relies on, namely the idempotency key and its retention [56], whether the provider refuses to delete an object that others depend on, and whether it offers a conditional delete. It also states whether the type is scratch and whether the deployment's workloads can need it later.

An adapter is written once per provider and reused for every task and agent. It needs no cooperation from the agent. Adapters are trusted. A wrong declaration, such as calling a type dependency-free when others can depend on it, weakens P4 for that type.

4.4 The Controller

The controller is a separate process that keeps all its state in the ledger, so it survives the computation and its own restart. Like a collector's marking pass, each periodic pass aborts the domains whose lease has expired, drops their claims, and marks every effect left without a claim as orphaned. An orphan first waits a short grace period, during which a request from another live domain can claim it again. The controller then applies the adapter's removal rule to each orphan (§4.6), looks up uncertain effects (§4.8), and finishes removals in progress.

An effect holds a claim on every effect it requires and drops these claims only when it is removed, discarded, or rejected. The controller therefore disposes of dependents before their dependencies. An escalated effect keeps its claims. When a child domain completes, the claims on its deliverables pass to the nearest live ancestor, or to the user's account at the root.

4.5 Holding Effects Back

For a held-back type, the monitor records the request, answers it with a receipt that carries the effect's ID, and does not forward it. A later request can use this ID, which the monitor replaces with the provider's identity on release. When the user completes the domain, the monitor releases the held-back effects in recorded order, so a referenced effect is sent first. When the domain is aborted, they are discarded. An aborted task therefore never creates these effects, while a released effect is disposed of like any other. This is output commit, with the user's decision in place of durability.

Which types to hold back. A later step that needs an effect's response, or needs the effect to exist, fails if the effect is held back. Our default holds a type back if and only if no later step of the workload can need it and no justification allows its removal. Under the prototype's declarations, only GitHub pull requests and Stripe subscriptions are held back, because no later step needs them and no justification removes them. The same type can be a deliverable in one task and a prerequisite in another, so we treat the rule as policy. We measure it against holding back every type that can be held and holding back nothing, which trade task success against protection (§6.5).

Why discard on abort. A domain is usually aborted because its computation died, so a discarded request might have succeeded. We do not keep it for later release, because it was composed against a world that may have changed. A deployment may declare escalation instead.

Held-back effects hold claims. A held-back pull request references its branch. Without a claim, the controller could remove the branch before release and make a successful task fail. A held-back effect therefore holds its claims from the moment it is recorded, and its release rechecks the fence in the same transaction. A client that retries a held-back request would otherwise create two held-back effects, so the ledger merges a retried request with an identical held-back request of the same domain.

4.6 Safe Removal

Others can start using an effect at any time without the monitor seeing it. The controller removes an orphan only under one of three justifications. Under *the provider refuses*, the provider rejects the delete while any dependent exists, as AWS does for a VPC in use [4, 5]. Under *conditional delete*, the provider checks a precondition and deletes in the same operation. Under *dependency-free*, the adapter declares that nothing can depend on the type, as for an uncaptured payment [54]. An effect of any other type is kept or escalated.

Listing the dependents and then deleting does not qualify, because a dependent created in between is destroyed. GitHub deletes a branch without any precondition and closes its open pull requests [25], so no justification applies to a GitHub branch, and the controller never removes one. Branch protection [24] forbids deletion altogether. A deployment may allow list-then-delete as a declared *bounded-risk* policy, which we never count as safe.

Observed references only add protection. A claim from a referencing effect can stop a removal but can never cause one. Claims protect the dependents the monitor records, and justifications protect those it cannot see. The controller starts a removal only by a compare-and-set on the effect's version while no claim holds it. In the opposite order, the *fence* makes the monitor reject any request that references an effect being removed.

4.7 Completion

When the user completes a domain and declares its deliverables, the user's account claims each deliverable and the held-back effects are released. The domain's claims on scratch

effects are dropped, so they are removed under a justification or escalated. Every other effect is escalated with its full record. A scratch effect stays as long as a kept effect requires it, as the branch stays under the released pull request.

4.8 Fault Tolerance

The computation stops. The user or the launcher aborts the domain. If the launcher dies too, its lease expires and the controller aborts the domain.

The monitor crashes. When the monitor restarts, an intent without a dispatch mark was never forwarded, so it is rejected, and a held-back request stays held. An intent with a dispatch mark may have executed, so its effect becomes uncertain and is looked up. The monitor holds a lock on the ledger for its lifetime, and the controller handles unfinished intents only when it can take that lock. At restart, the monitor releases the held-back effects of domains that completed before the crash.

The controller crashes. A restarted controller resumes from the ledger and repeats any removal in progress, which is safe because removal treats an absent resource as removed. A second lock ensures that only one controller acts on a ledger.

A response is lost. As with a remote procedure call [13], the caller cannot tell whether the request executed. The system never sends it again, since that could create a second effect. The controller asks the adapter to look up the uncertain effect. A found resource makes the effect outstanding. An empty lookup is not enough, because the request may still execute later. The controller writes off an uncertain effect as rejected only with proof that the request cannot still execute. The proof is an idempotency key whose outcome is final [56], an expired provider-bounded request lifetime, or a record that the request was never forwarded. Otherwise the effect is escalated after a bounded number of failed lookups.

The provider refuses or a release fails. A refusal to remove is a definite answer, so the controller escalates the effect with the refusal as evidence. A rejected release is reported, and one with an unknown outcome follows the lost-response path. Released effects stay released, since effect recovery attempts no compensation.

4.9 Model-Checked Invariant

We state the core invariant in TLA+ [34] and check it with TLC [65]. The invariant states that every effect reaching a provider is recorded first and ends in exactly one outcome, and that no effect is removed while a recorded effect requires it or an unrecorded dependent exists, given justified removals and correct dependency-free declarations. The model has a root and a child domain and includes lost responses with late execution, the three justifications, outside dependents, held-back effects, completion, and one controller crash. TLC checks it exhaustively on instances of three and four effects, and checks progress under fairness with one crash.

Provider	Effect types	Removal after abort
AWS	VPC, S3 bucket	provider refuses
AWS	instance, CodePipeline	dependency-free
AWS	subnet, security group	bounded risk
AWS	bucket policy	escalate
GitHub	comment	dependency-free
GitHub	branch	bounded risk
GitHub	issue, pull request	escalate
Stripe	uncaptured payment	dependency-free
Stripe	customer, subscription	escalate
Slack	message	dependency-free
Slack	channel	escalate

Table 1. Effect types in our adapters and their removal after abort. Unknown types are escalated. In the guaranteed configuration, bounded-risk types are escalated too.

Three weakened models fail. With list-then-delete, a 12-step trace destroys an unrecorded dependent (§4.6). With write-off on an empty lookup, a 6-step trace ends with an effect that exists after its write-off (§4.8). With held-back effects that hold no claims, a 17-step trace ends with the failed release of a successful task (§4.5). The implementation must discharge five obligations. The intent, its claims, and the fence check commit in one transaction before forwarding. A removal starts only by compare-and-set with no claim present. An effect becomes outstanding only with an identity from the real response or a lookup. Removal is idempotent. Release rechecks the fence in the releasing transaction.

5 Implementation

We built two implementations. The evaluation uses the Python one unless it says otherwise. Its monitor and controller are two processes sharing one SQLite ledger in write-ahead-log mode with synchronous commits.

Confinement. A launcher starts each computation in a Docker container whose entry point installs firewall rules that drop all outgoing traffic except to the monitor, including DNS and IPv6, and then drops to an unprivileged user. Provider host names resolve to the monitor. The domain credential is a bearer token, or for AWS the access key inside the SigV4 signature, which the monitor re-signs with the real key. The monitor never saves authorization headers.

Controller and completion. The controller runs a pass every 50 ms and escalates an uncertain effect after three failed lookups. The user completes or aborts a domain through a ledger operation unreachable from the container.

Adapters. Table 1 lists the effect types. The adapters share a base class of about 100 lines. The Stripe and Slack adapters take about 50 and 60 lines, the GitHub adapter about 170, and the AWS adapter, with six effect types, about 230. Our harness adds bucket policies as a seventh AWS type. The GitHub adapter targets Gitea's GitHub-style API. A separate 60-line adapter targets real GitHub.

Kernel implementation. The second implementation enforces the boundary in the kernel. A Linux security module of about 5,000 lines of C binds every new process to its parent's domain. Members of a domain can reach only the monitor, here written in Rust. It records every request in a checksummed append-only ledger before acting, holds effects back, and enforces the lease.

6 Evaluation

The evaluation answers five questions.

RQ1 Do today's options leave dangling effects or silent leaks where effect recovery leaves none?

RQ2 Is each component needed, and do the removal rules keep effects that others still use?

RQ3 Does the guarantee hold when the monitor or controller crashes, or a provider's response is lost?

RQ4 Do real agents still complete their tasks under effect recovery?

RQ5 How many effects does effect recovery dispose of without a person, and at what cost?

6.1 Methodology

Providers and hosts. Experiments run against local emulators of AWS (Moto), GitHub (Gitea), Stripe (stripe-mock), and Slack (a small fake), and against real test accounts on Stripe, GitHub, Slack, and AWS S3. Correctness experiments run on two Linux servers with 16 cores each, two macOS laptops, and a Mac mini. Per-call latency and throughput come only from Linux, where a synchronous commit reaches the device.

Models. Real-agent experiments use Claude Sonnet 5 through Claude Code, with the agent's own tools removed, so that every action goes through one tool that runs a command in the domain.

Configurations. In the guaranteed configuration, every removal carries one of the three justifications of §4.6. The prototype also implements the bounded-risk policy of §4.6 for GitHub branches and AWS subnets and security groups. Where that policy changed an outcome, we report the guaranteed configuration beside it. We never count a bounded-risk removal as safe.

What counts as a failure. We read provider state and raw provider responses, never the agent's own account. A run fails the guarantee if it removes an effect that someone else still uses (*harm*), leaves an effect at a provider that no record names (*silent leak*), loses a record, releases a held-back effect early or twice, discards one that should have been released, removes a deliverable, or leaves an effect with neither an outcome nor an owner. An unresolved effect with a named owner is a permitted outcome.

Option	Recall	Precision	Harm	Silent leaks
Agent cleans up	0%	0%	0	153
Durable execution	0%	0%	0	153
Tag and sweep	91.2%	92.5%	0	12
Inventory, shared account	98.8%	2.0%	12,833	3
Per-run credentials*	98.8%	100%	0	3
Container teardown	0%	0%	0	153
Logging proxy	100%	46.8%	178	0
Logging proxy, given deliverables	100%	100%	0	0
Effect recovery	100%	100%	0	0

Table 2. Real agents under ordinary pressure: what each option selects and leaves behind over 120 attempts with 153 intermediates. Harm counts selected resources that another run still uses or that the task delivered. Leaks are silent leaks. *Analytic, computed from ground truth.

6.2 RQ1: Today's Options

Options. We compare agent-side cleanup from the agent's own list, durable execution with compensations (Temporal), tag and sweep, container teardown, a container per run with all calls routed through a logging proxy, and our system. In the scripted workload, holding every effect back until the user completes the task stops at the first request that needs the provider's answer, with no deliverable. §6.5 measures it on real tasks. Per-run credentials with provider records are tested on real accounts below.

Real agents under pressure. Agents perform real tasks with instructions to finish quickly and cheaply, and we inject no failure. Four tasks each leave an intermediate and each runs 30 times under effect recovery. When the agent stops, we score every other option on the same attempt (Table 2). Agents completed 115 of the 120 tasks and removed none of their 153 intermediates. Durable execution leaves all of them because nothing registered a compensation. With one credential per run, an inventory difference would select only the run's own resources, but attributing a run takes 1.2 s against 8 ms for effect recovery. A logging proxy's log cannot tell a deliverable from scratch. Given the user's declared deliverables, it matches effect recovery here, because these tasks involve no teammate and nothing to hold back.

What agents leave behind. Without any system, the real-agent runs of §6.5 left 6,571 new objects at the providers in 2,706 attempts, mostly the deliverables the tasks asked for. The 283 failed attempts left 604 objects, and 47 of them left at least one object that no part of the task asked for, such as a payment on the wrong customer. Only 5 of the 2,423 successful attempts did so. These counts are a lower bound, because the snapshots cover only what each task's check reads.

The situations of the running example. Six scripted situations test recording and removal: a stop between steps (Stop), an effect missing from the agent's list (Forgotten), a helper's direct provider call (`curl`), a resource the helper creates without the run's tag (Untagged), concurrent runs

Option	Stop	Forgotten	`curl`	Untagged	Shared acct.	Teammate	Hold back	Deliverable
Agent cleans up	fail	fail	fail	pass	pass	fail	fail	pass[2]
Durable execution	pass	pass	fail	pass	pass	fail	fail	fail
Tag and sweep	pass	pass	pass	fail	pass	fail	fail	pass
Container teardown	fail	fail	fail	fail	fail	fail	fail	fail
Logging proxy	pass	pass	pass	pass	pass	fail	fail	fail[3]
Effect recovery	pass	pass	pass	pass	pass	pass[1]	pass	pass

Table 3. The situations of the running example, five rounds per cell, with effect recovery in the guaranteed configuration. A cell passes under the rule in the text with none of the other failures of §6.1. [1] It escalates the branch, and the review survives in 5 of 5 rounds. Under the bounded-risk policy the review is destroyed in 5 of 5. [2] Passes when the agent names its deliverable correctly. Misnaming it fails in 5 of 5. [3] Passes when given the user's declared deliverables.

in the same account, each with its own credentials (Shared account), and outside use of an effect before cleanup (Teammate). Two more test decisions about the final pull request (Hold back) and the user's declared deliverable (Deliverable). A cell passes with mean recall at least 95%, mean precision at least 99%, no harm, and no silent leak.

Table 3 reports 240 scripted runs. Temporal misses the helper's direct call, which is not one of its steps. Tag and sweep misses the untagged resource, and teardown leaves external effects in place. The logging proxy publishes the final pull request before completion. When the runs share one credential per provider, every option gives its Shared account result. On real accounts, provider records can attribute effects by GitHub App or Stripe key while each run has its own credential (§2).

Takeaway. Every other option in Table 3 fails at least one situation, and effect recovery passes all eight. With real agents under ordinary pressure, it named every leftover without selecting anything in use and left no silent leak (Table 2).

6.3 RQ2: Necessity of Each Component

For RQ2, we remove one component of §4 at a time and run a scenario built to need it.

Removing one component at a time. Each scenario runs 20 or 40 times with the component and as many times without it. We predicted each failure before the run.

Table 4 shows that each removal produces the failure the component exists to prevent. Two removals fail only in a second scenario. Recording after the response loses nothing for good when only the computation dies, because the surviving monitor still records it. Until then the effect exists at the provider with no record, in all 20 trials without the component and none with it. When the monitor itself is killed, the removal leaves an unrecorded branch in 5 of 5 trials. Removing claims held by effects changes nothing for an S3 bucket, which the provider refuses to delete while it holds objects. On a provider that permits the delete, it deletes a resource a live pipeline needs in 20 of 20 trials.

Prop.	Component removed	Outcome without it
P1	Monitor as the only way out	20 unrecorded effects, all leaked
P1	Lookup before repeating a request	two objects in 40 of 40 runs; one with it
P1	Record from observation, not the agent's report	40 leaks, 20 outside resources deleted
P1	Record before forwarding	20 effects present with no record at the cut
P2	Decision outside the computation	20 failed runs' effects kept for good
P2	Completion rule (keep everything)	20 runs leak their scratch
P2	Completion rule (remove everything)	20 runs lose their deliverables
P3	Controller outside the computation	20 runs with nothing decided
P3	Write-off only with proof	20 effects leak after a late execution
P4	Claims held by effects	20 resources removed under a live pipeline
P4	Claims of held-back effects	20 releases of a successful run refused

Table 4. Removing one component of the design. The full system passed all 380 of its trials with no prohibited outcome.

A kept pull request also holds a claim on its branch, so the branch remains in 20 of 20 runs where only the pull request is declared. Without that claim, cleanup escalates the branch, again in 20 of 20.

The monitor sees every request. Eleven client stacks, including `curl`, the Stripe SDK, and a subprocess with an empty environment, each attempt 20 mutating requests. With the container's network confined to the monitor, the monitor saw and attributed 220 of 220 attempts. With a proxy set through environment variables, 20 clients making 400 attempts were seen only 65% of the time, and 140 attempts bypassed the proxy with no record. Here these hosts did not resolve, so none executed. Seven of the 20 clients ignored the proxy variables.

Observed references only add claims. Over 31 dependency situations, the monitor sees a reference from one effect to another for about two thirds of the real dependencies: 62% by our labeling, and 70% and 65% by two model families labeling blind. References therefore only add claims (§4.6). A harness places an outside pull request on the branch between the list and the delete in every trial. The guaranteed configuration escalates the branch in 40 of 40 trials and never sends the delete. List-then-delete destroys the dependent in 40 of 40, each recorded in the ledger as bounded risk. With the GitHub branch wrongly declared dependency-free, the same race harms the dependent in 40 of 40 trials. Every removal is recorded under its declared justification, and every other effect keeps its correct outcome. A wrong declaration therefore breaks P4 only for the type it covers.

Summary. Every component is needed, and the complete system has no prohibited outcome in 380 trials. The monitor sees every request that leaves the domain, and removal stays safe although the monitor sees only two in three dependencies.

6.4 RQ3: Failures

RQ3 injects failures into the trusted parts and checks that every effect still ends in an outcome with its record intact.

Fault injection. We stop the monitor or the controller, or lose or delay a provider's response, at 22 points of an effect's lifecycle, 200 times each with random timing: 4,400 trials on each of Linux and macOS. After the system recovers, an audit compares the ledger with provider state and checks every failure category of §6.1. The audit detects each defect we planted in passing trials: a deleted record, an effect without owner and outcome, and an effect left at the provider after its recorded removal.

Results. No trial on either host has a prohibited outcome in any category. With 0 of 200 per point, the one-sided 95% upper bound on the failure rate is 1.5% per point and 0.07% pooled. Every trial recovered within the 10 s bound. On Linux, this takes 0.10 s at the median, 0.96 s at the 99th percentile, and 1.44 s at most. The four points at which the controller removes two resources account for the tail. We also scored each of the 22,000 effects in these trials that needed a decision against provider ground truth. None was removed while still needed, left without an owner, or left unclassified.

Further campaigns. Appendix A reports crashes before and after every durable write (1,520 trials), 500 kills at random times, checks of specified outcomes and of a liveness deadline, and all four provider families with every computation in its own container. All 220 trials reached their specified outcomes, a lapsed liveness deadline ended the run exactly once in 40 of 40 trials, and the four-provider container matrix had no prohibited outcome. The one-sided 95% upper bounds are 0.20% for the 1,520 enumerated crashes, 0.60% for the 500 kill attempts, and 0.68 to 0.80% per provider family in the container matrix. In one interval, after the monitor durably marks a held-back message as dispatched but before it forwards it, the unsent message is escalated to a named owner, as the contract permits (P3).

Takeaway. Across the 22 selected crash points, no effect was left unrecorded, dangling or leaked.

6.5 RQ4: Real Agents

The introduction bounds the acceptable loss at two percentage points of task success. RQ4 tests that bound with real agents on real tasks, and measures what mediation costs them in turns, tokens, and time.

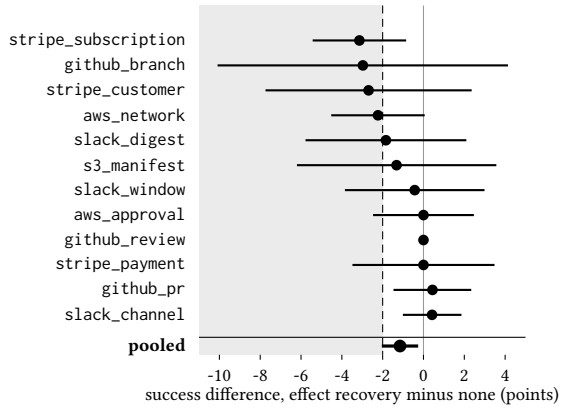


Figure 4. Task success with and without effect recovery in the 24-turn test, 2,699 matched pairs. Per task: difference and 95% interval. Pooled: the difference averaged over tasks, with the interval whose lower end is the one-sided 95% bound. The shaded region lies beyond the two-point margin.

Tasks and design. Twelve tasks span GitHub, Stripe, Slack, and AWS, three per provider. Their later steps need nothing back from a request, the identifier it created, or the provider's answer. Stripe runs against a real test account and the others against emulators. The campaign schedules 318 repetitions of each task with effect recovery under its default policy and without any system, as matched pairs started together on the same account. Success is judged from provider state after the run. The two-point margin, the analysis, and the task bank were fixed before the campaign. Each attempt may take up to 24 turns, a budget we tuned so that agents have room to finish the tasks. Fewer than 0.1% of attempts in either arm reach the turn cap.

Results. Over the 2,699 pairs of the 24-turn test (Figure 4), agents complete 88.2% of tasks with effect recovery and 89.4% without it, averaged over tasks. The difference is −1.15 points. Its one-sided 95% bound is −2.04 points, so the test misses the two-point margin by 0.04 points. A bootstrap over pairs within each task gives a bound of −1.98 points. An attempt with effect recovery takes 0.11 more turns, 9.15 against 9.04, and 2.9 s more on average, with the same median of 93 s. Both arms use about 104,000 tokens per attempt.

Where mediation costs a task. Only one task has a difference whose interval excludes zero. On the task that creates a Stripe subscription, which is held back, agents succeed in 96.9% of pairs with effect recovery and in all of them without it. In six of its seven failures, the agent could not read an ID from the receipt and sent the request again. The monitor merged the retry with the held-back request (§4.5). The agent could not tell, so it stopped and asked for review. Receipts that report the merge would remove this case.

Held-back effects arrive. In the default arm, completed attempts held back 447 effects, 229 pull requests and 218

subscriptions. All 447 were released and are present at the provider after the run.

Completed with evidence. A task that needs the provider's answer counts as completed only if the agent read that answer before it acted. Every attempt of these tasks that passed its check had received the required value in an earlier response, 898 of 898 without a system and 896 of 896 with effect recovery.

The two policy extremes. We ran both extremes on the twelve tasks, ten repetitions each, next to runs without a system. Holding nothing back matches running without a system, with 109 and 105 successes of 120. Holding everything back succeeds in 44 of 120. It passes every task whose later steps need nothing back from a request. It fails every task that needs the provider's answer and 36 of the 40 attempts at tasks that need an identifier, because a receipt stands in for the provider's object.

A second harness. We also ran the twelve tasks with Codex, five times each with and without effect recovery. Both arms succeed in 55 of 60 attempts. In each arm, all five failures are on the task that links a new Stripe customer to a payment, so effect recovery costs no task. The held-back subscription task succeeds in all five mediated attempts. We audited every mediated attempt and found no prohibited outcome.

Takeaway. Agents lose about one point of task success under effect recovery and spend nearly the same turns, tokens, and time.

6.6 RQ5: Automation and Cost

RQ5 measures how many effects the controller disposes of without a person, and what mediation costs next to an agent's own work.

Disposed of without a person. We take every effect recorded in the real-agent runs of §6.5 and apply the adapters' declared rules to it under an abort. The efficiency-pressure runs of §6.2, in which the controller disposes of real agents' effects at the providers, check this. Of the 8,619 effects recorded in 2,701 attempts, an abort leaves 41.7% settled without a person (Table 5). The controller removes 34.1%, mostly bot messages and uncaptured payments. Another 5.3% were held back and are discarded, and 2.4% were rejected by the provider and never existed. The rest goes to an owner, 1.7 effects per aborted attempt, each with its record in the ledger. Types without a justification, such as issues, channels, customers, and bucket policies, account for 21.4%, and bounded-risk types, such as branches, subnets, and security groups, for 7.6%. Another 24.2% are requests recorded without a removable identity, which the controller escalates. Most change an effect already recorded, such as adding a rule to a security group. The rest are git pushes, S3 object uploads, and edits to objects that existed before the run. Finally, 5.2% are kept because an escalated effect still claims them. Under the bounded-risk policy, the settled share rises to 49.2%. On completion, the user's account keeps the task's deliverables, 70.8% of the recorded effects. In the efficiency-pressure runs, the controller removed 30 of the 153 intermediates at the providers, all working branches under the bounded-risk policy, within 0.2 s of completion at the median.

Provider	Effects	Removed	Discarded	Rejected	Kept	Escalated
AWS	2,081	0	0	0	444	1,637
GitHub	2,382	0	229	135	0	2,018
Slack	2,698	2,460	0	3	0	235
Stripe	1,458	477	224	65	0	692
All	8,619	2,937	453	203	444	4,582
Share		34.1%	5.3%	2.4%	5.2%	53.2%

Table 5. What an abort would leave to the controller for every effect recorded in the default arm of §6.5, under the adapters' declared rules in the guaranteed configuration. Escalated includes 651 effects of bounded-risk types and 2,087 requests recorded without a removable identity.

Cost per call. We pair each mediated call with a direct call to the same provider. Against the emulators, the monitor adds 3.8 to 4.7 ms at the median. As a share of an emulator call, overhead ranges from 7% for Gitea to 355% for stripe-mock, which measures the emulators' speed rather than the monitor. The monitor adds 4.4 to 8.2 ms for real Stripe, Slack, and S3. Across all four real providers, the added median is 2.4 to 4.4% of the round trip, over 1,624 mediated calls that all succeeded and were all cleaned up. GitHub's added median, 16 to 31 ms, has an interval that includes zero, because its own response time varies by hundreds of milliseconds.

At scale and in the kernel. With 500 domains, 10,000 calls at 64-way concurrency completed at 565 calls per second, with added latency at the 99th percentile below the 250 ms bound we set before the run. As the ledger grows to 1 GiB, every step grows at most linearly: the monitor starts in 2.2 s, the controller's first pass after a restart takes 3.3 s, and a forwarded call's added time stays near 4.8 ms at the median. The kernel implementation starts a domain in 2.7 ms alone or 202 ms among 500 and holds it in 160 KiB, against 1.3 s, about 101 s, and 7.5 MiB for a Docker container. A whole kernel run is ready in 18 ms, comparable to restoring a Firecracker microVM from a snapshot [1]. With ledgers on tmpfs, the kernel implementation's monitor adds 0.09 ms to a call, against 1.8 to 2.2 ms for the Python monitor. A completed run's effects reach their outcomes in under a second, and a stopped run's in about 3 s, mostly its lease. A held-back pull request reaches GitHub 0.57 s after completion. Start-up, kernel, and disposition results are in Appendix B.

Adapters. An adapter takes a few dozen lines per effect type on a shared base class (§5).

Takeaway. In the guaranteed configuration, an abort settles 42% of real agents' effects without a person and hands the rest to an owner with their record. The monitor adds a few milliseconds per call, a few percent of a real provider's round trip, next to a median agent attempt of 93 s. The kernel implementation starts and holds runs at a fraction of the cost of a container.

7 Discussion and Limitations

What the contract does not cover. Effects that never pass the monitor are outside the contract. These include effects that a provider creates on its own, such as a CI run that a push triggers, and the agent's calls to its model. Processes within one domain are not separated from each other. Effect recovery does not control information flow, so confidentiality is out of scope.

Malicious computations. We assume that the computation is not malicious. Within its confinement, it can still cause any effect its task allows, such as a damaging message, and the monitor records that effect and the controller hands it to an owner but does not prevent it.

Trusted declarations. The guarantee rests on the adapters' declarations. An adapter that classifies a mutating request as a read lets an effect pass unrecorded. An adapter that declares a type dependency-free when others can depend on it allows a removal that harms them. Effect recovery does not check these declarations.

Release and staleness. Releasing held-back effects is not atomic across providers, so if one release fails, the effects already released stay released. A held-back request is sent as it was recorded, so a long delay can release a request that no longer matches the world, and checking it against the world as Bayou checks the dependencies of a tentative write [28, 59] is future work.

Agents are not trained to be monitored. In our runs, some agents stopped when a held-back request returned a receipt instead of the provider's object. Today's models are not trained to work behind a monitor. Receipts that say more, and a way for an agent to list its held-back effects, should reduce this cost.

8 Related Work

Effect recovery builds on systems that make a program's work durable or reversible, manage the lifetime of resources, or confine agents and mediate their traffic. None records every effect of an unmodified computation and takes responsibility for it after the computation ends.

Durable execution and transactional agents. Workflow engines such as Temporal, Restate, and DBOS [20, 49, 58] journal a program's steps so that it resumes after a crash. Atomix [39] and Cordon [17] bring transactional semantics to agents' tool calls. All of them cover the work that code routes through their API. Effect recovery records effects where requests leave the computation, and disposes of each according to what the provider guarantees.

Resource lifetime and cleanup. A CloudFormation stack rolls back a failed deployment [2]. Terraform and Pulumi destroy the resources in their state [27, 48]. These tools remove whatever their declared scope contains, including what a teammate has come to depend on. Effect recovery removes only observed effects that a justification makes safe.

Garbage collection. Effect recovery adapts ideas from garbage collection [32]. Like a conservative collector that cannot see every pointer [14], it works over a heap whose references live at providers and with other people. It therefore removes an effect only when the provider guarantees or the adapter declares that nothing depends on it, and hands every other effect to an owner.

Speculation and output commit. Speculator [40] and Remus [19] release output once a speculation is validated or the state behind it is durable or replicated. Speculative agents [29, 63] run actions early to save time. In all of them the system decides when output is final. Effect recovery releases output only when the user completes the task.

Undo across services. Retro [33], Warp [15], and Aire [16] undo an attack and re-execute the legitimate actions. GoEX [47] proposes undo and damage confinement for the actions an LLM generates. These systems repair the damage of actions known to be wrong, or offer undo for the actions they run. Effect recovery disposes of the effects of computations that ended.

Mediating agents and their traffic. Egress proxies and gateways [31, 57] route and log outgoing traffic. Such a log supplies observation but no record written before forwarding, claims, justified removal, handling of lost responses, or held-back effects. Agent frameworks let a person approve a tool call [35, 46]. Sandboxes confine an agent's processes [1, 61, 64]. ActPlane [66] and revocable capabilities [52] decide whether a call may happen. These systems decide what a computation may do. Effect recovery decides what happens to what it did, and can run behind any of them.

Accountability and provenance. Audit trails [10], bonded recourse [11], and irreversibility budgets [38] record, insure, or ration agents' external actions. Externalization barriers [30] sketch holding back agents' effects as an OS abstraction, without an implementation or an evaluation. Effect recovery adds claims that outlive the computation, provider-backed removal, handling of lost responses, and an implemented and evaluated system.

9 Conclusion

Effect recovery takes responsibility for the effects that agents leave at external services. It treats their lifetime as garbage collection over a heap that the collector cannot fully see. A

monitor records every effect before it leaves the computation, and a controller that outlives the computation removes an effect only when removal is known to be safe. Every component is necessary, the guarantee holds under crashes at every step, and the monitor adds only a few milliseconds per call. Agents complete nearly as many tasks through it.

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, USA, 419–434.

[2] Amazon Web Services. 2026. Choose how to handle failures when provisioning resources. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/stack-failure-options.html. AWS CloudFormation User Guide. Accessed 2026-09-25.

[3] Amazon Web Services. 2026. DeleteBucket. https://docs.aws.amazon.com/AmazonS3/latest/API/API_DeleteBucket.html. Amazon S3 API Reference. Accessed 2026-09-25.

[4] Amazon Web Services. 2026. DeleteVpc. https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_DeleteVpc.html. Amazon EC2 API Reference. Accessed 2026-09-25.

[5] Amazon Web Services. 2026. Error codes for the Amazon EC2 API. https://docs.aws.amazon.com/ec2/latest/devguide/errors-overview.html. Defines DependencyViolation. Accessed 2026-09-25.

[6] Amazon Web Services. 2026. What Is AWS CloudTrail? https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-user-guide.html. AWS CloudTrail User Guide. Accessed 2026-09-25.

[7] James P. Anderson. 1972. *Computer Security Technology Planning Study, Volume II.* Technical Report ESD-TR-73-51, Vol. II. Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, USA.

[8] Anthropic. 2026. Claude Code Changelog. https://github.com/anthropics/claude-code/blob/main/CHANGELOG.md. Versions 2.1.172 to 2.1.224.

[9] Anthropic. 2026. Hooks reference. https://code.claude.com/docs/en/hooks. Claude Code documentation. Accessed 2026-09-25.

[10] Laurent Bindschaedler, Quentin Botha, and Christoph Siebenbrunner. 2026. Agent Flight Recorder: Tamper-Evident Audit Trails with On-Chain Anchoring for Long-Horizon Tool-Using Agents. In *2026 8th International Conference on Blockchain Computing and Applications (BCCA 2026)*. IEEE, Barcelona, Spain. To appear. arXiv:2609.01931.

[11] Laurent Bindschaedler, Quentin Botha, and Christoph Siebenbrunner. 2026. Bonded Recourse for Smart-Contract Settlement of Compensable Agent Side Effects. In *2026 8th International Conference on Blockchain Computing and Applications (BCCA 2026)*. IEEE, Barcelona, Spain. To appear. arXiv:2609.01939.

[12] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. 1993. Network Objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM, New York, NY, USA, 217–230. doi:10.1145/168619.168637

[13] Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems* 2, 1 (1984), 39–59. doi:10.1145/2080.357392

[14] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Software: Practice and Experience* 18, 9 (1988), 807–820. doi:10.1002/spe.4380180902

[15] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. 2011. Intrusion Recovery for Database-backed Web Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 101–114. doi:10.1145/2043556.2043567

[16] Ramesh Chandra, Taesoo Kim, and Nickolai Zeldovich. 2013. Asynchronous Intrusion Recovery for Interconnected Web Services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 213–227. doi:10.1145/2517349.2522725

[17] Zheng Chen, Hanqing Liu, Duling Xu, Dong Dong, Jialin Li, Bangzheng Pu, and Jidong Zhai. 2026. Cordon: Semantic Transactions for Tool-Using LLM Agents. arXiv:2606.17573 [cs.OS] doi:10.48550/arXiv.2606.17573

[18] Crossplane. 2026. Crossplane. https://www.crossplane.io. Accessed 2026-09-25.

[19] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*. USENIX Association, San Francisco, CA, USA, 161–174. https://www.usenix.org/conference/nsdi-08/remus-high-availability-asynchronous-virtual-machine-replication

[20] DBOS, Inc. 2026. DBOS Transact: Workflows and Steps. DBOS documentation, https://docs.dbos.dev/typescript/reference/workflows-steps. Accessed 2026-09-25.

[21] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)* (Lausanne, Switzerland). Association for Computing Machinery, New York, NY, USA, 467–481. doi:10.1145/3373376.3378512

[22] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *Comput. Surveys* 34, 3 (2002), 375–408. doi:10.1145/568522.568525

[23] GitHub. 2026. About GitHub Copilot Cloud Agent. https://docs.github.com/en/copilot/concepts/agents/cloud-agent/about-cloud-agent. Accessed 2026-09-25.

[24] GitHub. 2026. About protected branches. https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-protected-branches/about-protected-branches. GitHub Docs. Accessed 2026-09-25.

[25] GitHub. 2026. Managing branches within your repository. https://docs.github.com/en/pull-requests/how-tos/commit-changes/managing-branches-within-your-repository. GitHub Docs. Accessed 2026-09-25.

[26] HashiCorp. 2026. Terraform. https://developer.hashicorp.com/terraform. Accessed 2026-09-25.

[27] HashiCorp. 2026. terraform destroy command reference. https://developer.hashicorp.com/terraform/cli/commands/destroy. Terraform documentation. Accessed 2026-09-25.

[28] Pat Helland and Dave Campbell. 2009. Building on Quicksand. In *Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research (CIDR '09)*. Asilomar, CA, USA. https://www.cidrdb.org/cidr2009/Paper_133.pdf

[29] Coleman Hooper, Minwoo Kang, Suhong Moon, Nicholas Lee, Eric Wen, John Wawrzynek, Michael W. Mahoney, Yakun Sophia Shao, Amir Gholami, and Kurt Keutzer. 2026. Speculative Interaction Agents: Building Real-Time Agents with Asynchronous I/O and Speculative Tool Calling. arXiv:2605.13360 [cs.LG] doi:10.48550/arXiv.2605.13360

[30] Jinhao Hu, Bardia Mohammadi, Ashvin Goel, and Laurent Bindschaedler. 2026. Externalization Barriers: An OS Abstraction for Untrusted Agent Exploration. In *2nd Workshop on Operating Systems Design for AI Agents (AgenticOS 2026), co-located with SOSP 2026*. Prague, Czechia. https://os-for-agent.github.io/ Non-archival workshop.

[31] Istio Authors. 2026. Egress Gateways. https://istio.io/latest/docs/tasks/traffic-management/egress/egress-gateway/. Istio documentation. Accessed 2026-09-25.

[32] Richard Jones, Antony Hosking, and Eliot Moss. 2023. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (2nd ed.). Chapman and Hall/CRC. doi:10.1201/9781003276142

[33] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2010. Intrusion Recovery Using Selective Re-execution. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. USENIX Association, Vancouver, BC, Canada, 89–104. https://www.usenix.org/conference/osdi10/intrusion-recovery-using-selective-re-execution

[34] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

[35] LangChain. 2026. Interrupts. https://docs.langchain.com/oss/python/langgraph/interrupts. LangGraph documentation. Accessed 2026-09-25.

[36] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, USA, 53–68.

[37] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2024. AgentBench: Evaluating LLMs as Agents. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*. OpenReview.net, Vienna, Austria. https://proceedings.iclr.cc/paper_files/paper/2024/hash/e9df36b21ff4ee211a8b71ee8b7e9f57-Abstract-Conference.html

[38] Bardia Mohammadi and Laurent Bindschaedler. 2026. The Irreversibility Budget: Fleet-Level Risk Accounting and Admission Control for Agent Operating Systems. In *2nd Workshop on Operating Systems Design for AI Agents (AgenticOS 2026), co-located with SOSP 2026*. Prague, Czechia. arXiv:2609.00275 https://os-for-agent.github.io/ Non-archival workshop.

[39] Bardia Mohammadi, Nearchos Potamitis, Lars Henning Klein, Akhil Arora, and Laurent Bindschaedler. 2026. Atomix: Timely, Transactional Tool Use for Reliable Agentic Workflows. In *Proceedings of the ICLR 2026 Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD)*. OpenReview.net, Rio de Janeiro, Brazil. https://openreview.net/forum?id=UeRbEpSVUz

[40] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. 2005. Speculative execution in a distributed file system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 191–205. doi:10.1145/1095810.1095829

[41] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the Sync. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Seattle, WA, USA, 1–14. https://www.usenix.org/conference/osdi-06/rethink-sync

[42] OpenAI. 2026. Agent Approvals & Security. https://learn.chatgpt.com/docs/agent-approvals-security. Accessed 2026-09-25.

[43] OpenAI. 2026. Agent internet access. https://learn.chatgpt.com/docs/cloud/internet-access. Codex documentation. Accessed 2026-09-25.

[44] OpenAI. 2026. Agents API. https://developers.openai.com/api/docs/guides/agents-api/overview. Public beta announced September 10, 2026. Accessed 2026-09-25.

[45] OpenAI. 2026. Codex cloud. https://learn.chatgpt.com/docs/cloud. Codex documentation. Accessed 2026-09-25.

[46] OpenAI. 2026. OpenAI Agents SDK: Human-in-the-loop and Guardrails. https://openai.github.io/openai-agents-python/human_in_the_loop/. See also https://openai.github.io/openai-agents-python/guardrails/. Accessed 2026-09-25.

[47] Shishir G. Patil, Tianjun Zhang, Vivian Fang, Noppapon C., Roy Huang, Aaron Hao, Martin Casado, Joseph E. Gonzalez, Raluca Ada Popa, and Ion Stoica. 2024. GoEX: Perspectives and Designs Towards a Runtime for Autonomous LLM Applications. arXiv:2404.06921 [cs.CL] doi:10.48550/arXiv.2404.06921

[48] Pulumi. 2026. pulumi destroy. https://www.pulumi.com/docs/iac/cli/commands/pulumi_destroy/. Pulumi CLI commands. Accessed 2026-09-25.

[49] Restate. 2026. Restate. https://restate.dev. Accessed 2026-09-25.

[50] Chris Richardson. 2026. Pattern: Transactional Outbox. https://microservices.io/patterns/data/transactional-outbox.html. Accessed 2026-09-25.

[51] Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (1975), 1278–1308. doi:10.1109/PROC.1975.9939

[52] Igor Santos-Grueiro. 2026. Lingering Authority: Revocable Resource-and-Effect Capabilities for Coding Agents. arXiv:2606.22504 [cs.CR] doi:10.48550/arXiv.2606.22504

[53] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2022. DBOS: A DBMS-oriented Operating System. *Proceedings of the VLDB Endowment* 15, 1 (2022), 21–30. doi:10.14778/3485450.3485454

[54] Stripe. 2026. Cancel a PaymentIntent. https://docs.stripe.com/api/payment_intents/cancel. Stripe API Reference. Accessed 2026-09-25.

[55] Stripe. 2026. Delete a customer. https://docs.stripe.com/api/customers/delete. Stripe API Reference. Accessed 2026-09-25.

[56] Stripe. 2026. Idempotent requests. https://docs.stripe.com/api/idempotent_requests. Stripe API Reference. Accessed 2026-09-25.

[57] Stripe. 2026. Smokescreen. https://github.com/stripe/smokescreen. GitHub repository. Accessed 2026-09-25.

[58] Temporal Technologies. 2026. Temporal Workflow. Temporal documentation, https://docs.temporal.io/workflows. Accessed 2026-09-25.

[59] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 172–182. doi:10.1145/224056.224070

[60] The Kubernetes Authors. 2026. Garbage Collection. https://kubernetes.io/docs/concepts/architecture/garbage-collection/. Accessed 2026-09-25.

[61] Cong Wang and Yusheng Zheng. 2026. Sandlock: Confining AI Agent Code with Unprivileged Linux Primitives. arXiv:2605.26298 [cs.CR] doi:10.48550/arXiv.2605.26298

[62] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, Vol. 37. Curran Associates, Inc., Vancouver, BC, Canada, 50528–50652. doi:10.52202/079017-1601

[63] Naimeng Ye, Arnav Ahuja, Georgios Liargkovas, Yunan Lu, Kostis Kaffes, and Tianyi Peng. 2026. Speculative Actions: A Lossless Framework for Faster AI Agents. In *Proceedings of the Fourteenth International Conference on Learning Representations (ICLR 2026)*. OpenReview.net, Rio de Janeiro, Brazil. https://openreview.net/forum?id=P0GOk5wslg

[64] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA, USA. https://www.usenix.org/conference/hotcloud19/presentation/young

[65] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods (CHARME '99) (Lecture Notes in Computer Science, Vol. 1703)*, Laurence Pierre and Thomas Kropf (Eds.). Springer, 54–66. doi:10.1007/3-540-48153-2_6

[66] Yusheng Zheng, Tianyuan Wu, Quanzhi Fu, Tong Yu, Wenan Mao, Tao Ma, Dan Williams, Wei Wang, and Andi Quinn. 2026. ActPlane: Programmable OS-Level Policy Enforcement for Agent Harnesses. arXiv:2606.25189 [cs.OS] doi:10.48550/arXiv.2606.25189

A Further Fault Injection (RQ3)

Section 6.4 crashes the monitor or the controller, or loses a response, at 22 selected points of one scenario with two providers. This section widens that test in four ways: crashes around every durable write, crashes at random moments, a stricter check of the outcome each crash must reach, and all four providers with every computation in its own container.

Beyond the selected crash points. We also crashed a representative run before and after each of its 38 durable writes (76 positions), 20 times per position (1,520 trials), and made 500 kill attempts on the monitor or the controller at uniformly random moments, 428 of which hit a process. No trial had a prohibited outcome. The one-sided 95% upper bound is 0.20% for the enumeration and 0.60% for the kill attempts. Crashes between the monitor's durable dispatch mark for a held-back message and its forward ended in a permitted escalation. Such crashes occurred at two of the positions (40 enumerated trials) and in one random kill, and we targeted the interval with 50 further injections. In all 91 crashes, provider state shows that the message never reached the provider, and the ledger retains its dispatch mark and records its effect as escalated to a named owner. Because the mark must be durable before the forward and the message has no idempotency key, the controller escalates it rather than risk a duplicate send, and the contract permits this escalation (P3).

Two further checks. At each of the 22 crash points, all 220 trials reached the exact outcome we specified for that point before the run. A run whose liveness deadline expired while the controller or the monitor was down ended exactly once, in 20 of 20 trials each.

All four providers, in containers. The fault matrix of Section 6.4 uses two providers and runs each computation as a plain process. We repeated its crash injection with all four providers and with every computation in its own container, as in deployment, 20 seeds per crash point. Each trial with S3 (Moto) also holds Slack messages (a small fake), and 440 such trials cover 22 crash points. GitHub (Gitea) ran 399 trials over 20 points with the bounded-risk policy enabled, so that crashes also hit branch removal, and Stripe, against a real test account, ran 373 trials over 19 points. Every trial reached its crash point, except 20 Stripe trials of a dependency scenario that has none and is judged by its own assertions. No trial had a prohibited outcome. The one-sided 95% upper bounds are 0.68% for S3 and Slack, 0.75% for GitHub, and 0.80% for Stripe.

B Starting, Holding, and Disposing of Runs (RQ5)

Section 6.6 reports the cost per call, the automation share, and the headline scale and start-up numbers. This section gives the details behind them: what limits the monitor at 500 domains and as its ledger grows, how quickly effects reach their dispositions, what starting and holding a run costs against containers and microVMs, and the kernel implementation's cost per call.

At scale. In the 500-domain run, 10,000 calls with at most 64 in flight completed at 565 calls per second, about 18 s, with no error, and the domains then reached their dispositions in 153 s. During the call phase, added latency was 129–133 ms at the median and 209–233 ms at the 99th percentile, below the 250 ms bound we set before the run. The monitor commits its ledger writes in batches from one writer thread, so its event loop stays free to serve calls. The tail is the wait for that thread, which is busy 95% of the time under this load. At the 99th percentile, 276 of 296 ms are spent in the call's three ledger writes, most of it queued, and only 16 ms in `fsync` and 16 ms at the provider. As the ledger grows from 1 MiB to 1 GiB, every step grows at most linearly: the monitor starts in 2.2 s at 1 GiB, the controller's first pass after a restart takes 3.3 s, disposing of the leftovers of about 1,000 unfinished runs takes 2.5 s, and a forwarded call's added time stays near 4.8 ms at the median at every size.

Time to disposition. After a user completes a run, its effects reach their dispositions in under a second. After a run stops without completion, this takes about 3 s, mostly waiting for the 3 s liveness lease. The controller's own work takes 0.14–0.39 s. Separately, a held-back pull request appears at the provider a median of 0.57 s after the user's declaration. In the RQ1 runs, effect recovery takes a median of 3.01 s to reach its reported outcome (3.26 s at the 95th percentile). The other options take 0.42 s for agent cleanup, 0.58 s for durable execution, 0.52 s for the logging proxy, 0.38 s for container teardown, and 44.6 s for tag and sweep.

Starting and holding runs. Table 6 compares startup time and memory. A Docker container takes 1.3 s to start alone, about 101 s when 500 start together, and 7.5 MiB per run. A whole run of the Python implementation, including its own monitor, answers its first request 2.9 s after launch, and the median of 500 launched together after 133 s. The same confinement with runc and a proxy takes 86 ms and 6.9 s. A kernel domain starts in 2.7 ms alone or 202 ms among 500, and a whole run admitted through its monitor takes 18 ms or 320 ms, comparable to restoring a Firecracker microVM [1] from a snapshot. A domain holds 160 KiB, against 3.1 MiB for a restored microVM and 50 MiB for a cold one. Snapshot restore and lightweight secure containers [21, 36] narrow their cost gap.

Most of the start cost is Docker's own. With no launcher options and a smaller image, `docker run` alone takes 1.1 s, and a rootful Docker daemon starts containers 11–15% faster than the rootless one measured here. Setting up the firewall (0.19 s) and waiting for readiness (0.15 s) come after that start.

	Unit started per run					Whole run	
Concurrent runs	Kernel domain	runc + proxy	Docker (Python)	MicroVM, cold	MicroVM, snapshot	Portable	Kernel
1	2.7	85.9	1,292	312	38.4	2,913	18.1
10	6.4	118.0	3,408	360	41.8	6,565	33.6
50	17.9	375.9	11,272	621	58.9	15,820	126.1
100	33.5	757.8	21,875	1,092	78.0	29,978	142.3
250	95.0	2,445	52,959	4,729	132.4	63,498	270.7
500	202.0	6,921	101,040	7,850	353.8	133,051	319.5
Teardown, 1 / 500 runs	26.6 / 44.2	139.9 / 1,705	1,914 / 134,283	126.6 / 1,113	117.9 / 975.8	1,914 / 134,283	24.4 / 49.6
Memory added per run, at 500	160 KiB	256 KiB	7.5 MiB	50.5 MiB	3.1 MiB	42.4 MiB	225 KiB

Table 6. Cost of starting and holding runs. All times are in milliseconds, medians over five repeats: the time to start one unit when the given number start at once, and the time to tear down one and 500. Memory is what each run adds with 500 running. A whole run is ready when its first request is answered: the Python implementation starts a monitor per run inside its launcher, and the kernel implementation admits the run through its one shared monitor. The Python implementation's teardown, stopping and removing the container and ending its monitor, is the same in both of its columns. A whole run's memory includes its monitor and the growth of shared processes, the Docker daemons or the kernel implementation's monitor, but not their fixed size, which for that monitor and its 16 adapter processes is about 440 MiB.

Of the 7.5 MiB per run at 500 runs, half is the container's cgroup and half its shim, excluding the monitor that both implementations run. The runc figure counts only the container process.

Mediation in the kernel implementation. To compare mediation without the cost of storage, we ran both implementations with their ledgers on tmpfs. There, the kernel implementation's monitor adds 0.09 ms to a call, its adapter included, of which about 0.05 ms is its own work, against 1.8–2.2 ms for the Python implementation's monitor. In a separate measurement at 500 concurrent runs of four effects, the kernel's checks cost 9–12% of throughput: 184 against 209 runs per second with 16 adapter processes, and 299 against 329 with 32.