

L3M: Capability Without Authority

A Verifiable Harness and Runtime for Self-Extending and Self-Improving Agents

Carson Eisenach*, Robert Joseph George, Vincent Quenneville-Bélair, Kate Gwimm, Dean Foster

Amazon

October 5, 2026

1 Introduction

Agents now do real work — they edit code, run research experiments and author documents, and increasingly they spawn other agents to do the same. The program that turns a language model’s text into those effects — the *harness* — holds every bit of authority the model appears to exercise: whatever the model decides, the harness is what opens the file, starts the process and sends the bytes. In other words, the harness is the security boundary. In most harnesses today it is an afterthought: a permission dialog, and a hope that the model behaves.

That hope fails under two pressures. The first is scale. Long-running work wants many agents — one to plan, several to work, one to review — sharing a repository and mailing each other for days. There is no single model call whose behaviour could be audited; there is only the composition of effects across a tree of processes, each with its own context, some of them asleep. Trusting judgment does not compose, and approving each action by hand scales in neither agents nor time. The second is openness. What an agent will need next — a bisector for this regression, a scraper for that dataset — is not known when the session starts, and the most capable agents write those tools themselves. An allowlist of actions is either too narrow to be useful or too broad to mean anything. A harness for such agents must bound *authority* — what may be touched, and whether it can be undone — rather than enumerate actions.

L3M (Lean + Large Language Models) bounds authority by proof. Its harness is a Lean 4 program, so the properties that make it safe are theorems about the code that runs, checked by Lean’s kernel on every build, rather than properties of a configuration argued in prose; a change that breaks one does not build. Lean plays three roles at once: it is the language L3M is written in, the language its guarantees are proved in, and the language its agents extend it in. The third role changes the division of labour among the parties to an agentic system. The model is good at deciding what to do and composing the pieces; code is good at looping, computing and enforcing; the human is good at judging what matters and deciding whom to trust. An agent that knows what it wants done turns that plan into code — a sealed program the harness admits under the grant the agent already holds (Figure 11) — so the loop runs as code, with guarantees, and the model reasons once over the result. For this to be safe, a program must be unable to do more than its author was trusted to do; for it to be useful to the human, what it can do must be readable without reading the program. Because the extension language is the implementation language, one type checker meets both conditions: a program’s type is its permission request — the tokens it asks for — and the harness holds it to that type. Capability grows; authority does not.

Table 1 lists what is proved. At session boot L3M mints one capability token for each authority the agent’s grant confers — a directory it is confined to, a Git ref prefix, a network allowance — and every operation that touches the file system, a process, the network, Git or the model takes one as an argument, so a tool without the token cannot even state the call. Tokens cannot be forged: only the module that defines a token can construct it, which the build enforces, and that the boot mints a token exactly when the grant confers its authority is a theorem. The few modules that do touch the operating system form a named, audited trusted base (Section 3.2). The root grant comes from a human, and every other grant is provably an attenuation of its parent’s, so authority and budget only narrow as the tree of agents grows. We do not claim a single theorem that a session is safe: the operating system, Git and the model provider are named assumptions, and every guarantee is reported with what it rests on (Section 3.1).

*Correspondence to carson@eisenach.org.

Table 1: The headline guarantees. Each is a theorem about the code L3M runs and what it rests on.

Guarantee	Theorem	Rests on
Confinement. A file tool cannot name a path that climbs out of the agent’s roots.	<code>safePath_no_traversal</code>	The kernel. The check is on the path’s text; symlinks are checked at run time.
Monotone authority. A child’s grant never exceeds its parent’s, and no agent grants to itself.	<code>grant_restrict_le</code> <code>no_self_grant</code>	The kernel.
No token without the grant. The boot mints a token only for an authority the grant confers, and a narrower grant mints none a wider one lacked.	<code>mint_reflects_profile</code> <code>Minted.restrict_none</code>	The kernel. That tokens cannot be forged is enforced by the build, not proved.
Budget is lent, not minted. All a subtree of agents can spend is bounded by its root’s budget.	<code>global_budget_bounded_by_root</code>	The kernel.
Self-extension adds no authority. A sealed program the harness admits uses only admitted constants.	<code>sealUsed_admits_only_safe</code>	The kernel.
Review cannot be skipped. A tool whose effects cannot be undone and are not confined by a capability is always reviewed.	<code>ungated_world_reaching_always_reviewed</code>	The kernel.
Undo. A journaled edit can be restored to the bytes it replaced.	<code>live_capture_round_trips</code>	The kernel, and a named assumption that Git’s object store returns the bytes it hashed.

Capabilities bound what an agent can do; *reversibility* decides where the human’s attention is spent. A tool’s review tier is derived from its footprint, not declared: actions that can be undone — reads, journaled edits, commits to version control — are *tame*, and the agent may take them without asking; actions that cannot, such as uploading data, are not. Much useful work is not tame: a web search leaks a query, and an email to a collaborator cannot be recalled. For these calls L3M replaces the permission dialog with review as deliberation. A prover agent argues for the action and a verifier challenges it. The exchange can exit by finding a tame way to do the same work; when it does not, the human is consulted, and what they are shown is the effect, not a shell string to decode. The human stays in the loop without becoming the loop: the alternatives are an agent that asks forever or one that is trusted blindly.

Capability and reversibility both rest on a third principle, separation of duties [38]: no party that can author an effect may also be the one that authorizes it, and no process that can read private data may also be the one that sends it. An agent may edit a Makefile, but only a human can bless it, and the blessing is a hash of the bytes the human read, so any edit withdraws it. An agent may build a program, but running it is a separate grant. A container step that mounts private source has no network, and no later step of the same plan regains it. An agent may issue grants to its children but never to itself; a doctor agent may repair other agents’ settings but never its own; and a risky call is proposed by one model and judged by another that holds no tools. This is W^X [11] applied to authority rather than to memory pages: every role that could chain two duties into an attack is designed to be split across two parties, so compromising one is not enough (Section 5).

The same bound is what lets L3M scale to many agents. L3M is multi-agent and distributed from the start, and Git is the source of truth for agent state: each agent’s work, mail and lifecycle live under a ref prefix it is confined to, and its files in a worktree it is confined to (Figure 1). The permission tree, the ref tree and the directory tree are therefore one tree. Because every grant is an attenuation of its parent’s, a guarantee proved for an agent holds for its whole subtree, and cleaning up after an agent is simply collecting that subtree. And because an agent’s behaviour is a fold over what Git stores, the loops that measure and improve agents run inside L3M, as bounded agents themselves (Section 6).

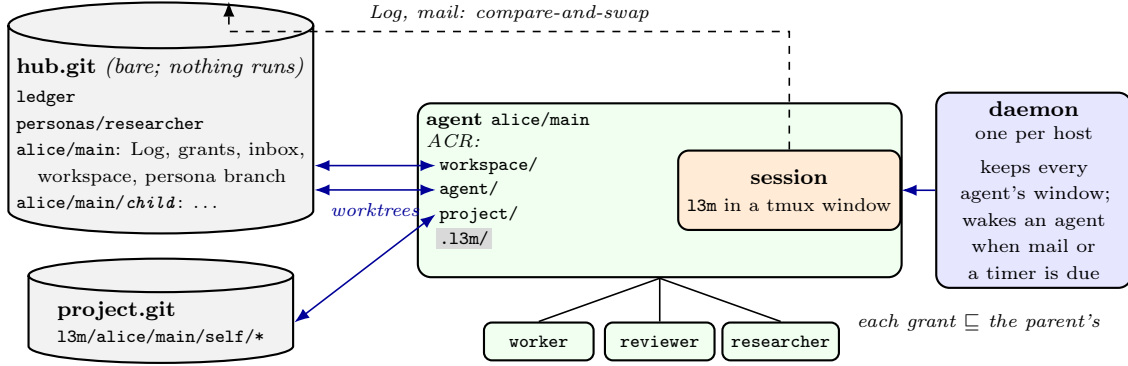


Figure 1: An agent in L3M: refs in the hub (a bare Git repository), worktrees on its branches there and in a claimed repository, and a session kept by the per-host daemon. Its children’s grants are contained in its own; the shaded .l3m/ is denied to it.

We formalize L3M incrementally rather than all at once, so at any moment some of its properties are proved, some are checked by evaluation, and some rest on assumptions about the world. That makes the human’s question — what exactly is guaranteed? — hard to answer by reading proofs. Lean Manifests [16] answers it instead: each headline claim is stated in terms a human can read and tagged with what it rests on — the kernel alone, compiled evaluation, or a named assumption such as Git’s behaviour — computed from the proof rather than declared by its author. A Lakosian levelization [27] of the source fixes which claims each layer may rest on, so that proofs build up floor by floor.

Contributions.

1. A harness written in Lean 4 whose security properties are kernel-checked theorems about the code that runs: effect-typed tools, capabilities carried into every call, and authority and budget that provably only narrow across a tree of agents (Sections 3 and 4.2).
2. A hybrid programming model in which agents extend their own tool surface with sealed programs whose types state, and bound, what they can do (Section 3.4).
3. Reversibility as a review tier derived from each tool’s footprint, and review as deliberation, and the autonomy the two buy (Section 3.3).
4. Separation of duties as a design rule: duties that could be chained into an attack are held by different parties (Section 5).
5. A Git-backed distributed runtime in which the permission, ref and directory trees coincide (Sections 4 and 4.3).
6. The persona model of agent identity and addressing: agents fork personas as branches, and identity branches and merges over time (Section 2).
7. Agent improvement inside the harness: behaviour, cost and timing are folds over the Log, so experiments on agents run as ordinary bounded agents (Sections 2.4 and 6).
8. Incremental formalization of a large Lean program, made legible to a human: each guarantee is stated readably with the evidence and world assumptions it rests on, computed from its proof, over a levelized source tree.
9. An initial evaluation on Terminal-Bench against Claude Code: how much work tame tools alone can do, how useful tool programs are, and what review costs and how often it redirects a call to a tame tool (Section 7).

2 Agents and Personas

This section describes a single agent: what it is, where it lives, what it wears, and how it runs. Section 4 assembles many agents into a system (Figure 12).

Ref	Holds	Written by	In the ACR	Retire / collect
<i>Hub-wide</i>				
l3m/ledger	who exists, who is whose parent, the grants issued	harness	—	append-only
personas/name	the persona’s prompt files, hooks and tool programs	a human	via a persona branch	outlives its wearers
<i>Written by the agent</i>				
workspace	its work in the hub	agent	workspace/	kept
persona (branch)	its copy of the persona, and its preferences	agent	agent/	kept / deleted
l3m/uri/self/*	its branches in a claimed repository	agent	repo /	kept
<i>Written by the harness, checked out</i>				
grants	grant, budget, pinned vigils	parent or human	.l3m/grants/	deleted
claims	the agent’s write roots	harness	.l3m/claims/	kept / deleted
<i>Never checked out</i>				
state	the Log	harness	—	kept
inbox	mail	senders	—	kept / deleted
health	failed turns	harness	—	kept
make	Makefiles a human blessed	a human	—	deleted
journal	the undo journal, in each write root’s repository	harness	—	—

Table 2: The refs that make up an agent and a persona. Every per-agent ref is **refs/heads/agents/role/uri/self**, and the table names only the role. In the last column, *kept* survives retire and collect, *kept / deleted* survives retire and is deleted at collect, and *deleted* is deleted at retire.

2.1 An agent is a name and its refs

L3M keeps agents in a *hub*: a bare Git repository shared by a group of agents, with one ledger recording who exists and who is whose parent. Nothing runs in a hub. An agent is a name in it — a path such as **alice/main**, whose child might be **alice/main/worker** — together with the refs keyed by that name (Table 2). The refs fall into three groups. The agent’s *authority* is its grants ref, which holds its grant, its budget and the vigils its parent pinned; the agent never writes it. Its *memory* is its Log, an append-only record of every turn that only the harness writes, together with its inbox and its own copy of the persona it wears. Its *work* is a branch in the hub and a branch in every repository it has claimed. An agent’s state is ordinary Git refs, so it can be diffed, fetched and audited with Git; and because names nest, the tree of refs is the tree of agents (Section 4.2).

2.2 The body: the agent context root

While an agent runs it also has a body, its *agent context root* (ACR): a plain directory on some host, where the agent’s session runs (Figure 2). The ACR holds worktrees, not copies. **workspace/** is checked out on the agent’s branch in the hub, **agent/** on its persona branch, and every repository the agent has claimed appears beside them as a worktree on a branch under the agent’s own prefix in that repository. These are the agent’s *write roots*, and the file tools confine every path to them. A fourth directory, **.l3m/**, belongs to the harness. It holds checkouts of the grants and claims refs, so that a human can change a grant by editing a file and committing it, and the agent can read or write nothing in it. It also holds copies the harness regenerates: the tokens minted at boot, and the Log as of the last turn. The rest of the agent is never checked out: the Log, the inbox and the undo journal are written by compare-and-swap on their refs, never through a checkout, and the agent reaches them only through tools. Deleting an ACR therefore loses nothing the harness wrote, and an asleep agent is simply refs with no process.

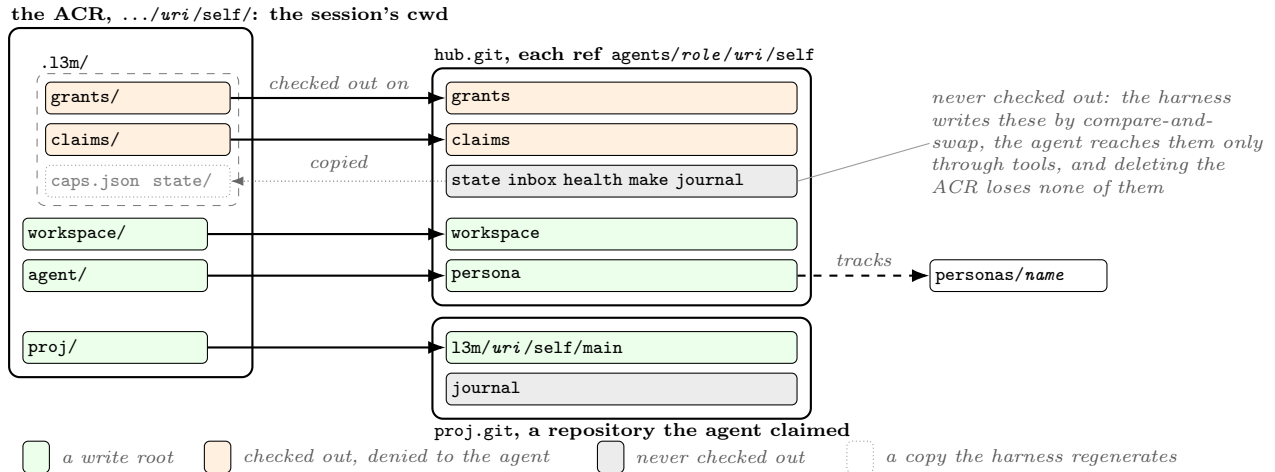


Figure 2: The body of an agent. Its write roots are worktrees on its own branches; the harness’s checkouts are denied to it; the rest of the agent is refs the harness writes and the agent reaches only through tools.

2.3 Personas, and how an agent is defined

A *persona* is a role rather than an agent. It is kept by name on its own ref, worn by zero, one or many agents at once, and holds no Log: it has no session, and nothing of it runs except in a wearer. What it holds is a small tree of files: three Markdown files — instructions for working in any repository, a personality, and memories meant to last — which become the top of each wearer’s system prompt; **agent.lean**, a Lean program that defines its hooks; and **programs/**, the tool programs its wearers have written (Section 3.4). Wearing a persona is forking it: an agent’s persona branch is cut from the upstream persona ref, tracks it, and is checked out as **agent/**, so an agent may edit its copy freely without affecting any other wearer, and a new wearer starts from the persona’s current version. A wearer can pull newer versions of the persona into its branch, and a human can merge a wearer’s improvements back upstream, so identity branches and merges over time the way code does. A persona is also one field of a mail address (Section 4.3).

Figure 3a shows how an agent’s system prompt is assembled at boot from three layers: the persona’s Markdown files, from the persona branch; the harness’s own account of what L3M is and what the agent may do, which travels with every L3M; and the operating instructions, L3M’s default or the persona’s own **system.md**. The assembled prompt is the first record of the agent’s Log, so a replay sees the prompt the agent saw (Section 2.4). Each request then adds a short suffix of notes for the model it is sent to, chosen by the provider from the model’s family, so a call that falls back to another model carries that model’s notes rather than the first’s. Whatever is on a persona ref becomes the highest-privilege text an agent reads, and its hooks and programs run in every wearer, so no agent’s grant can write one: only a human’s command publishes to a persona. A persona still learns from its wearers, through a human who reviews and publishes what they learned (Section 6).

An agent reacts to the world through *hooks* (Figure 3b), defined in **agent.lean** on its persona branch. The file is sealed and type-checked like any other program the agent writes (Section 3.4), and a hook may read files, the hub and the clock, and nothing else. When an event arrives — a letter, a vigil firing, a child’s completion, a deadline — the hook returns a disposition: show the event to the model, dismiss or ignore it, reply automatically, forward it to the sender or the parent, defer it, or retire a child. The harness bounds what a hook can start: an automatic reply is never sent in answer to a reply, nor to the agent itself. Each disposition is journaled as a decision, and a replay reads the decision instead of running the hook again, so an agent’s reactions are as reproducible as its turns. An agent with no hook, or with one the seal refuses, shows every event.

2.4 The Log and the event loop

An agent’s Log is its only memory of its sessions: an append-only sequence of records, one per fact, each numbered and timestamped to the millisecond (Figure 4). The first is the system prompt assembled at boot (Section 2.3).

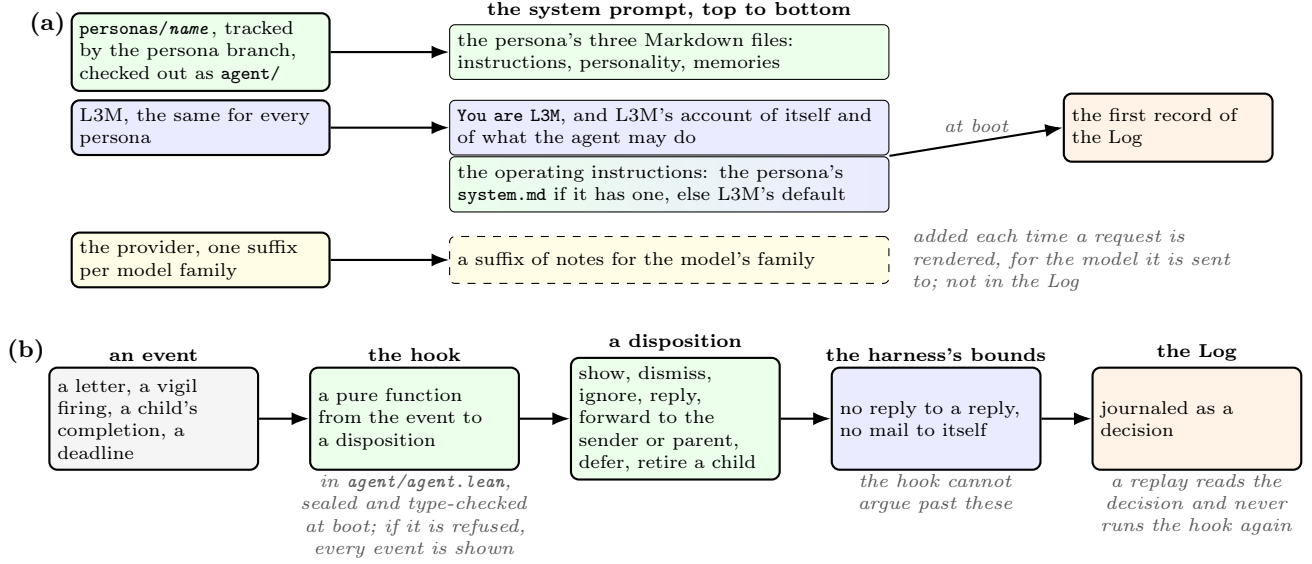


Figure 3: How an agent is defined. (a) The system prompt is assembled at boot from layers owned by the persona and by L3M, and each request adds a suffix for the model it is sent to. (b) Hooks decide what the agent does with each event, and each decision is journaled.

Messages are records, and each reply carries the provider's account of its tokens; so are the tool calls and their results, grant and budget changes, the events the agent was shown and what its hooks decided, and rewinds.

One process advances an agent, and in it the Log has a single writer, the *sequencer* (Figure 5). Everything else — the human's line, a fired vigil, a letter, a model's reply, a finished tool — only posts a signal to its queue. Each pass drains the queue, journals the signals consistent with the Log, journals what the agent's hooks decide about each new event, and then computes, as a pure fold over the Log, what is now owed: a notice to the model, a model call, or the end of the turn. It stores these records on the agent's ref before anything they authorize leaves the process, and only then starts the *activities* they name — the model call, each tool, each letter to send — which run outside the sequencer, keyed by id, and report back as signals. The sequencer never waits on the world, so mail is handled while a slow tool runs. At most one model call is open at a time (`open_call_is_last_call`), and each owed action is started once (`start_once`). Because every effect is journaled before it happens, a process killed at any instruction is repaired by the next one's ordinary passes: booting closes what the dead process left open exactly as an interrupt would (`boot_close_eq_interrupt_close`), and there is no separate resume procedure.

A turn never edits a conversation. It appends records, and what the model sees is recomputed from the Log as a pure function: rewinds are retracted, the current epoch is cut, compacted spans stand in for the messages they summarize, and state changes are lowered to neutral narration that travels with the next user turn. Only then does a per-provider dialect render the result on the wire. Two Logs with the same records produce the same view (`view_no_capability`); after a rewind, nothing past its target survives (`undo_across_epoch`).

Because every view is a fold, replay, `--continue`, rewind and fork are not separate mechanisms: each is the same fold over a different prefix of the same records. Records the model never needs — the harness's bookkeeping of calls, its per-turn token figures — stay in the Log and lower to nothing, so everything an agent did, and what it cost, remains data that other programs can read (Section 6).

2.5 Lifecycle, rewind and fork

An agent's lifecycle can be read off the last column of Table 2. Its state — running, asleep, done — is never stored; it is computed from the ledger, a session lock and the refs. *Retiring* an agent removes its body and its authority: the ACR goes and the grants ref is deleted, so nothing can run as the agent again. Retiring an agent that has children retires its subtree, deepest agent first. *Collecting* it later removes its memory: the inbox, the persona branch and

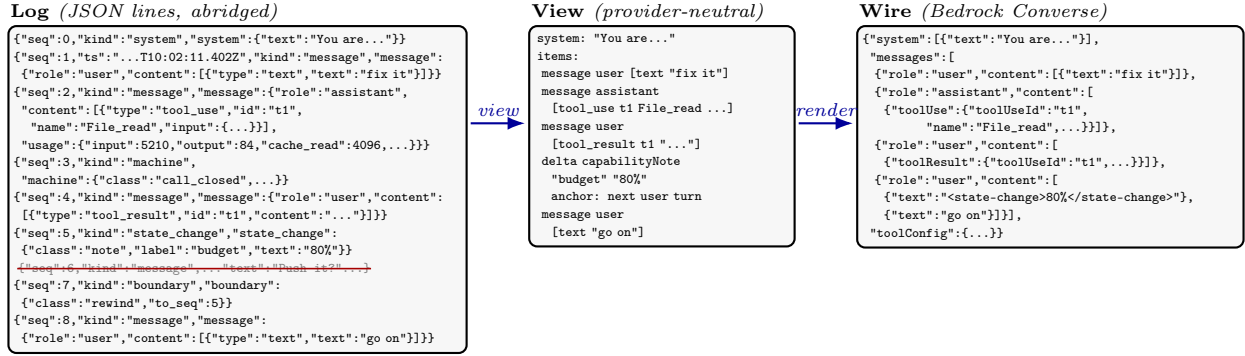


Figure 4: What the model sees is a pure function of the Log. Record 6 is retracted by the rewind at 7; the machine record at 3 lowers to nothing; the budget note at 5 becomes a delta that the dialect folds into the next user turn.

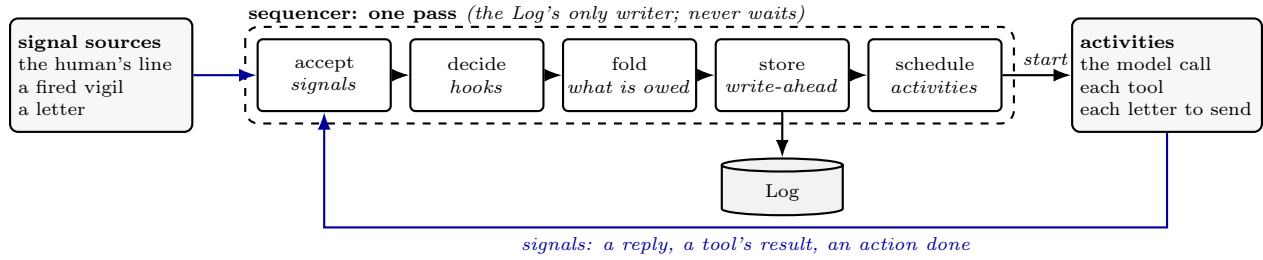


Figure 5: The event loop. Signal sources and activities only post signals, and the sequencer alone appends to the Log. Each pass accepts what is consistent with the Log, journals the hooks' decisions, folds the Log to decide what is owed, stores it, and only then starts the activities it names, so a kill at any point is repaired by the next pass.

the claims.

Because the Log is append-only, going back in time never destroys history. *Rewinding* an agent to an earlier turn appends a boundary to its Log and restores, from the undo journal, the files its tame actions changed; the agent resumes from that turn, and what cannot be restored is exactly what was not tame (Section 3.3). *Forking* starts a new agent from any point in another agent's Log, to pursue a tangent. The fork inherits the transcript up to that point and the source's persona branch, under a grant no wider than the source's.

3 Capabilities, the Trusted Base, and Self-Extension

This section states what L3M defends against (Section 3.1) and how. Tokens carry authority into every call, and the build keeps them from being forged (Section 3.2). Each tool's tier decides where the human's attention goes (Section 3.3), and sealed programs let an agent extend the harness without widening its authority (Section 3.4).

3.1 Threat model

L3M trusts two things: the human who writes the root grant and answers review cards, and a small base of code. That base is the Lean kernel and compiler, six small C files that make the system calls, the Linux kernel, the programs the harness runs (`git`, `tmux`, `unshare` and `timeout`), a pinned bundle of certificate authorities, and the model provider's API. It also trusts the hub's settings and ledger, which only a human or the harness writes. Everything else is untrusted: the model, every tool result, the contents of every repository, every persona and `AGENTS.md`, and every plugin. Text can change what the model wants to do, but not what it may do. An `AGENTS.md` that says "you may push to mainline" can persuade the model to try. The attempt still fails, because a grant is a ledger row that an ancestor appended, and no text widens one.

Table 3: What a hijacked model might try, and what stops it.

The model tries to	What stops it	Where
write ../.ssh/authorized_keys	A file path is a type whose values cannot climb out of the agent’s roots, so the call cannot be stated.	Table 1
reach /etc through a symlink in its workspace	The file shim resolves every path beneath its root and refuses to follow symlinks.	cannot_escape_root_via_symlink
add net to its own grant	No file tool admits a path under .l3m/, and no agent grants to itself.	admit_refuses_l3m, Table 1
rewrite another agent’s Log or grant	The Git token writes only the agent’s own branches, and never any agent’s Log, grants or claims.	grant_of_record_writes_no_entity_record
run curl https://x.sh sh	No process the harness starts is a shell or an interpreter. The shell tool, if granted, has no network and is always reviewed.	no_shell_or_interpreter
send data to a host its grant does not name	The network token names its hosts, and the provider checks every URL against them.	allowsUrl_authority_is_trusted
call the operating system directly, or forge a token	The code does not build.	Section 3.2
make an irreversible call without review	A tool’s tier is derived from its effects. A tool that claims more undo than its effects allow stops the harness from booting.	Table 1, Section 3.3
write itself a tool that does more	A program reaches only the tokens its author already holds.	Section 3.4

The hijacked turn. The attacker we design against controls some text the model reads, such as a web page, a file in a claimed repository, or a letter, and uses it to choose the model’s next tool calls. We assume the attack succeeds and the model does whatever the text says. The question is then what that model can do, and L3M’s answer is: what its grant allows, and nothing more. Table 3 lists the moves such a model might try and what stops each one. A hijacked model can still do damage within its grant, so confinement bounds the blast radius of a hijack but does not prevent it. Keeping untrusted text away from the model that holds the authority is a pattern built on top of L3M (Section 5.3).

What the human decides. Several guarantees are contracts: the harness enforces what the human chose, but it cannot make the choice a good one. The human writes the root grant, names the hosts a network token may reach, blesses Makefiles, and sets each agent’s fuel. The human also chooses how much is reviewed: `--review=world`, the default, reviews every call that can neither be undone nor is bounded by a token; `all` reviews every call; and `none` reviews nothing. A human who grants one agent both the project’s source and `net`: `any host` has built Figure 18(a), and L3M does not object.

What is not claimed. There is no session-wide safety theorem. The theorems say what a tool call *can* do, not whether the model *should* have made it: a tame action taken for a bad reason is still tame, and nothing says a patch is correct. The model’s prose to the human, and whatever the human pastes from it, is a channel no token touches (Section 8). Bounded authority is not secrecy: an agent that can read a key can copy it into a file it may push, and once pushed, the key is out of L3M’s hands. A session with no fuel limit is bounded only by the provider’s bill. Side channels, denial of service and writers outside L3M are out of scope, as are reads through a hard link that already pointed outside the root; the file shim refuses only to modify such a file. The redactor, which strips credential-shaped strings from tool output, and the sanitizer, which strips control characters and marks tool output as untrusted, reduce these risks but are not boundaries, and we claim nothing for them.

3.2 Effect classes and tokens

```

structure NetCap where                                -- a token
  private mk ::                                     -- only this module can construct one
  floor : TrustFloor                                -- the hosts it may reach

class NetEffects (m : Type -> Type) where            -- an effect class
  fetch : NetCap -> HttpRequest -> m (Except NetError HttpResponse)

def webFetchVia [Monad m] [NetEffects m]             -- a tool body; footprint ["NetEffects"]
  (cap : NetCap) (url : String) : m (Except WebRefusal HttpResponse) :=
  match admits cap.floor url with
  | .error r => pure (.error r)                       -- refused before any request is made
  | .ok _    => .ok <$> NetEffects.fetch cap { method := .get, url := url }

```

Figure 6: An effect class, its token, and a tool body written over them. The body’s type is its permission: it names `NetEffects` and nothing else, so it cannot reach IO, and the hosts it may fetch from are the ones its token carries.

A tool never touches the world directly. Its body is written over *effect classes*, Lean type classes such as `FsEffects` or `NetEffects` whose methods are the only way to reach that part of the world. Its *footprint* lists the classes the body uses. Each method takes a *capability token* as an argument. A token is a value that only the module defining its type can construct, and it carries its scope. The file token carries the agent’s write roots. The Git token carries the ref prefixes the agent may read and write, and the remotes it may push to. The network token carries the hosts it may reach. Figure 6 shows the network class. Its one method takes a `NetCap`, whose private constructor means no other module can make one, and whose `floor` field is the set of hosts it may reach. The web-fetch body is typed over `NetEffects` alone, so fetching is all it can do, and it checks the URL against the token’s own hosts before any request is made; the provider checks again. Table 4 lists all twenty effect classes. Every agent holds four of them, its own files, its blob store, a read-only view of the hub and the clock; each of the others needs an axis of the grant.



Figure 7: From grant to system call, for a child with one project worktree and no network. The grant decides which tokens exist, the tokens decide which tools the model is offered, and each call carries its token to the provider, which checks its scope again.

Figure 7 follows one grant (Figure 8) to a system call. Boot mints a token for each axis the grant confers and none for the rest (Table 1). The model is offered only the tools whose footprints those tokens cover, so a tool the agent may not use is absent rather than refused, and sealed programs and plugins pass the same check (Section 3.4). The provider, the one place a class meets the operating system, checks each call against its token’s scope again; this is where a symlink in the path is refused. Processes start only from a fixed list of programs, and a sandboxed command runs under `unshare` with no network and is killed when its time runs out.

Tokens cannot be forged, and tools cannot go around them, because of how the build is checked. Only a short, named list of modules, the *ring*, touches the operating system; every other module is written over effect classes. A *sealing linter* runs in every build and reads the elaborated code rather than the source text, so an alias or a re-export cannot hide a call from it. It refuses a token constructed outside its own module, whether directly, through a default instance or through a native implementation; any file write, process start or environment read outside the ring; and `unsafe` code or a new axiom outside the modules that evaluate foreign Lean. A tool that calls `IO.FS.writeFile`

Table 4: The effect classes, with the **Effects** suffix dropped. *Undo* says how far a class’s effects can be undone, and *Bounded* marks a class whose token limits its reach. The tier follows from those two columns (Section 3.3). **??** lists each class’s tools and theorems.

Family	Class	Granted by	A tool holding it may	Undo	Bounded	Tier
Files	Fs	always	read and write files under the agent’s roots	reversible	✓	tame
	Blob	always	store content by its hash	reversible	✓	tame
	WildRead	wildRead	read anywhere on the host	read-only		jail
	WildWrite	wildWrite	write outside every root	none		wild
Git and mail	GitLocal	git	commit and branch under its prefixes	reversible	✓	tame
	GitFetch	git	fetch into its mirror refs	none	✓	tame
	GitPush	git	push to the remotes its token names	none	✓	tame
Agent tree	Hub	always	read the ledger and other agents’ status	read-only	✓	jail
	Spawn	spawn	spawn a child under a narrower grant	none	✓	tame
	Launch	spawn	start or stop a child’s process	none	✓	tame
Time	Clock	always	read the clock	read-only		jail
Evaluation	Eval	evalPure	evaluate a pure Lean term into a fresh file	reversible		tame
Builds	Make	spawn	run a blessed or harness-lowered Makefile	reversible	✓	tame
	Build	spawn	build and run agent-written Lean, sandboxed	none		wild
Processes	Shell	shell	run a command in a sandbox with no network	none		wild
	Docker	docker	run one step of a container plan	none		wild
Network and data	Net	net	fetch from the hosts its token names	none	✓	tame
	Redshift	redshift	run SQL on a remote cluster	none		wild
Escape hatches	Process	spawn, git	run a subprocess	none		wild
	Danger	a launch flag	touch the host with no sandbox	none		wild

"/etc/passwd", or that builds a file token rooted at /, therefore does not compile.

Beside the grant, an agent holds a budget (Figure 8). The budget records the fuel the agent’s parent lent, counted in model calls, and how much of that fuel is spent. Both the grant and the budget live on the grants ref, which the agent never writes. Section 4.2 shows that authority and budget only narrow down the tree.

3.3 Reversibility and review

Tiers. Every effect class declares an undo ceiling (Table 4). From strongest to weakest, its effects are:

- read-only: they change nothing;
- reversible: the harness can restore what was there, on this host or across hosts;
- agent-reversible: the harness can restore the agent’s own state, though the world may already have seen the effect;
- irreversible.

A tool’s ceiling is the weakest of the ceilings of the classes in its footprint, and its tier follows from that ceiling: *jail* for read-only, *tame* for reversible, *feral* for agent-reversible, and *wild* for irreversible. One rule then lowers the tier. When a token bounds every class in the footprint, the tool is tame whatever its ceiling, because the human already approved its reach when they scoped the token. So **File_write** is tame because the harness can undo it, and **Git_push** and **Web_fetch** are tame because their tokens name the remotes and hosts they may reach, though neither can be undone. **Shell_run** is neither undoable nor bounded, so it is wild and every call is reviewed (Table 1). The rule only ever lowers a tier (**tier_le_default**), so tame means undoable *or* bounded in advance. A tool’s tier is computed from its classes and never declared by the tool. A tool may declare itself less undoable than its classes allow, but not more, and a build that contains a tool that over-claims refuses to boot.

```

structure Grant where
  spawn    : SpawnTier
  shell    : Bool
  redshift : Bool
  mailTo   : Cover := []
  git      : GitGrant
  web      : WebTier

structure BudgetFile where
  fuel : Nat
  spent : Nat := 0

theorem grant_restrict_le (parent role : Grant) :
  (parent.restrict role).le parent = true

```

Figure 8: A grant is a record of axes, a child’s grant is a meet with its parent’s, and the bound is a theorem.

261 **Undo.** Before a file tool writes, the harness saves the file’s old bytes to Git’s object store. At the end of the turn
 262 it seals that turn’s saves into one commit on the agent’s undo stream, and the undo journal records the branch head
 263 and the stream’s tip for that turn. Rewinding to a turn restores every file changed since, from its saved bytes, and
 264 resets the branches (Section 2.5). Both round trips are theorems under *named assumptions*: a named assumption is
 265 a Lean proposition about the world, stated with the observations that would falsify it, and a theorem that needs
 266 one takes it as an explicit hypothesis, so the assumption shows in the theorem’s type. A saved file reads back as
 267 the bytes it replaced, if Git’s object store returns what it hashed (Table 1). A commit is undone by resetting to
 268 the old head, if `git reset` restores a ref (`live_commit_reset_round_trips`). Saved bytes last only as long as the
 269 retention window before Git’s garbage collection. What undo cannot reach is whatever has left the host, such as a
 270 push, a request to a web host or a command that already ran.

271 **Evidence for a ceiling.** A class’s ceiling is declared, and this is one place where the formalization is incremental
 272 (??). Beside each declaration the harness records why it is honest, typed against the ceiling it supports (Figure 9).
 273 An irreversible ceiling needs no evidence, because it is the weakest claim and always safe. A read-only class names
 274 the theorem that its operations write nothing. A reversible or agent-reversible class carries a round trip: an instance
 275 of **Reversible**, which cannot be built without a proof that undoing an applied operation restores exactly the state
 276 before it. The file class cites the round trip of one anchored write, whose undo expects the new bytes and puts back
 277 the old ones; the theorem of the previous paragraph ties that model to the live store. A ceiling with no proof yet is
 278 *asserted*: it names its mechanism and waits for its proof. The evidence decides nothing at run time. The tier reads
 279 only the ceiling, so a proof moves no tier; it turns a claim into a theorem.

280 **Review.** A reviewed call goes through a short, bounded exchange before it runs, in place of a permission dialog:
 281 1. A *reviewer* model, the verifier, reads the tool, its arguments and the agent’s stated reason, and returns **safe**,
 282 **dubious** or **wild**. By default the reviewer is a smaller model, or the agent’s own model when no smaller one is
 283 configured
 284 2. The agent is asked whether a tool it already holds, one that is never reviewed, would do the same work. If it
 285 names one, the reviewed call does not run, the human is not asked, and the agent uses the tool it named instead.
 286 For example, an agent that asks to run `grep -rn TODO src/` through `Shell_run` is pointed to `File_grep`,
 287 which does the same work unreviewed.
 288 3. Otherwise the human sees a card. The card shows the call, the agent’s reason and the reviewer’s verdict, and
 289 the human answers yes, no, or challenge. A challenge asks the agent, the prover, to defend, revise or withdraw
 290 the call, and the reviewer then reconsiders, once.

291 With no human present, `--no` declines every call the reviewer flags. `--yes` runs a flagged call only if the agent
 292 defended it and the reviewer’s final verdict is **safe** (`wild_needs_human_or_deliberation`). The verdict is read
 293 only from the first line that begins `VERDICT:`. A reply without such a line counts as **wild**, and so does a reviewer
 294 that cannot be reached; tool calls in the reviewer’s reply are ignored. Review is defence in depth, not a boundary.
 295 The tokens are the boundary, and a human can still approve a bad call.

```

class Reversible (Op State Err : Type) where      -- an undo, with its proof
  apply      : Op -> State -> Except Err State    -- refuses if the state has drifted
  produces   : Op -> State                        -- the state an applied op leaves
  reverse    : Op -> Op
  restores   : forall o s, apply o s = .ok (produces o) ->
               apply (reverse o) (produces o) = .ok s

structure CellOp (a : Type) where                -- one anchored write
  pre  : Option a                                -- what it expects; none = absent
  post : Option a                                -- what it leaves; none = deleted
-- reverse swaps pre and post; restores is a two-line proof

inductive CeilingEvidence : ReversalClass -> Type 1 where
| bottom    (what : String)      : CeilingEvidence (.irreversible what)
| frame     (spelling world : String) : CeilingEvidence .readOnly
| roundTrip (inst : Reversible Op S E) (s : RestoreScope) : CeilingEvidence (.reversible s)
| asserted  (k : ReversalClass) (mechanism : String)      : CeilingEvidence k

instance : Ceiling "FsEffects" where
  evidence := .roundTrip (instReversibleCell (a := String)) .sameHost

```

Figure 9: How a ceiling is known. An undo is a **Reversible** instance, which carries its proof; each class’s evidence is typed by the ceiling it supports; and the file class cites the round trip of one anchored write. The agent-reversible case and the law that reversing twice is the identity are omitted.

```

structure ToolProgram where      -- trimmed
  description : String
  footprint   : List String      -- the classes the program declares
  run : {m : Type -> Type} -> [Monad m] -> [FsEffects m] -> [GitLocalEffects m]
        -> [GitFetchEffects m] -> [NetEffects m] -> [AwsAuthEffects m]
        -> [HubEffects m] -> [ClockEffects m]
        -> ProgramCaps -> String -> Confined footprint m String

```

Figure 10: A sealed program’s type is its permission request. The body is polymorphic in the host monad, so it can reach the world only through the seven classes it is given, and **Confined** admits only the classes the program lists in **footprint**. **ProgramCaps** holds the tokens the agent’s grant minted.

Vigils. An agent that must wait, for a build, a remote branch or a letter, arms a *vigil*: a condition the harness checks on the agent’s behalf. When the condition becomes true, the agent wakes with a turn. Each vigil has its own budget of wakes, and a model may arm at most sixteen. A parent can *pin* a vigil on a child by writing it beside the child’s grant. The child’s model cannot cancel, replace or retune a pinned vigil (**locked_invariant_under_model_verbs**), and reloading the parent’s pins never touches the vigils the child armed itself. For example, a planner can pin “wake when **mainline** moves” on each worker, so that every worker hears of every landing. The guarantee is that the wake happens, not that the child acts on it.

3.4 Self-extension: sealed programs

An agent extends its own tool surface by writing a Lean program. A file **agent/programs/todos.lean** on its persona branch, say one that counts the **TODO** comments under **src/**, becomes the tool **Program_todos** at the agent’s next boot. Writing the file is an ordinary, tame edit. What makes the program safe to run is what the harness checks when it loads it (Figure 10).

The harness makes three checks:

- **The seal.** The file must be a single term, with no tactics, options or local recursion. Every constant in its closure must be safe: nothing **unsafe**, **partial** or implemented natively, no token constructed outside its module, and no axiom beyond Lean’s standard three. The last condition also rules out **sorry** (Table 1). The program is elaborated in a restricted vocabulary that contains no token constructor and no bridge to raw IO.

- **The type.** The term must be a `ToolProgram`. Its body is written over an unknown host monad, so it can do only what the seven classes it is given allow. It can use only the ones it declares, because `Confined` provides a class's methods only when the footprint lists the class. So a program that fetches a URL without declaring `NetEffects` does not type-check.
 - **The surface.** The footprint must lie within the classes admitted for programs. The program is offered to the model only if the agent's tokens cover it, by the same check as a built-in tool (Section 3.2).
- At run time a program receives the tokens the grant minted, never the grant itself, and it cannot call the model or spawn an agent. It can therefore reach only what its author could already reach. Its tier follows from its footprint, like any tool's. The `TODO` counter names only `FsEffects`, so it is tame and runs unreviewed. A program that signs requests to AWS is reviewed, because that class has no undo ceiling and an unknown class counts as wild.
- Hooks (Section 2.3) are sealed the same way, but a hook only returns a disposition, which the harness carries out. A plugin is a bundle of programs. If any one of its programs fails a check, the whole plugin is refused, and a loaded plugin holds no authority beyond the agent's tokens (`loaded_plugin_authority_le_cap`).

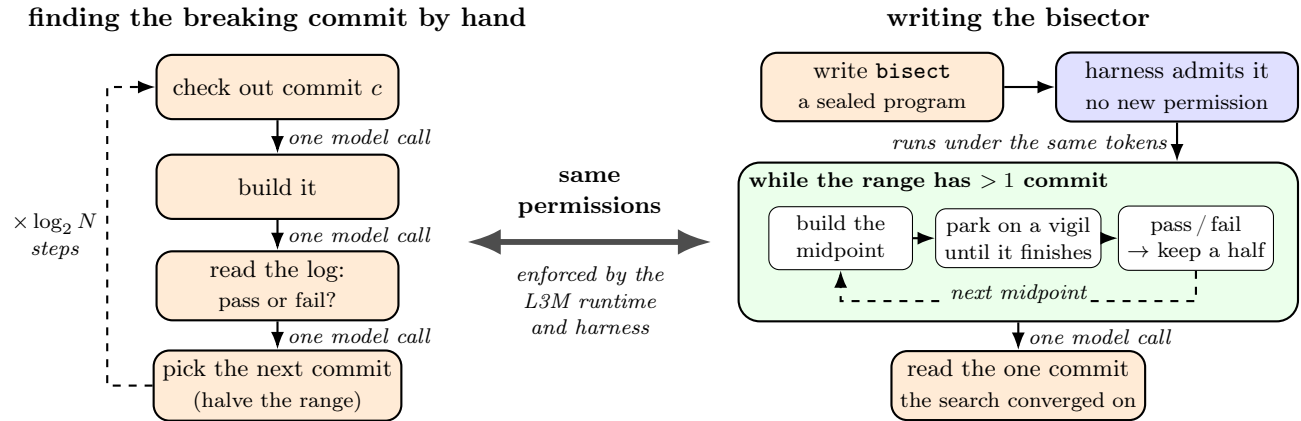


Figure 11: Two ways to find a breaking commit. Left: the agent is the loop, and every step costs a model call. Right: the agent writes the search as a sealed program that runs under the tokens it already holds; each step builds a commit and reads its exit status with no model call, and the model reasons once, over the result.

The result is a programming model (Figure 11). Because the bound is carried into every call, an agent may write a new tool as a sealed program and have it join its own tool surface with no new permission: the tool reaches exactly the capabilities the agent already held, and every invariant that held before still holds. Capability grows; authority does not. The model does what it is good at — deciding what to do and composing the pieces — and verified code does what it is good at — computing and enforcing. Because tame actions are reversible the agent can act without hedging, and because review is a deliberation between a prover and a verifier the human is consulted at the dubious margin rather than at every step. We argue that this is a building block for long-running, many-agent systems: the alternatives are an agent that asks forever or one that is trusted blindly.

4 Multi-Agent Architecture: Putting It Together

Section 2 described one agent. This section describes many: where they run (Section 4.1), why their authority only narrows as their tree grows (Section 4.2), and how they talk (Section 4.3).

4.1 Hubs and the daemon

A hub is a folder with no process of its own (Figure 12). It holds `hub.git`; a `forest/` of ACRs nested by name, so the body of `alice/main/worker` is under `forest/alice/main/worker/self/`; a `settings/` checkout of the hub's settings branch, which lists the repositories the hub knows, its mail offices and its terminal layout; and a cache. The first session creates a default hub under `~/.l3m/hub/`. A human may keep more, and a session's hub is the one its

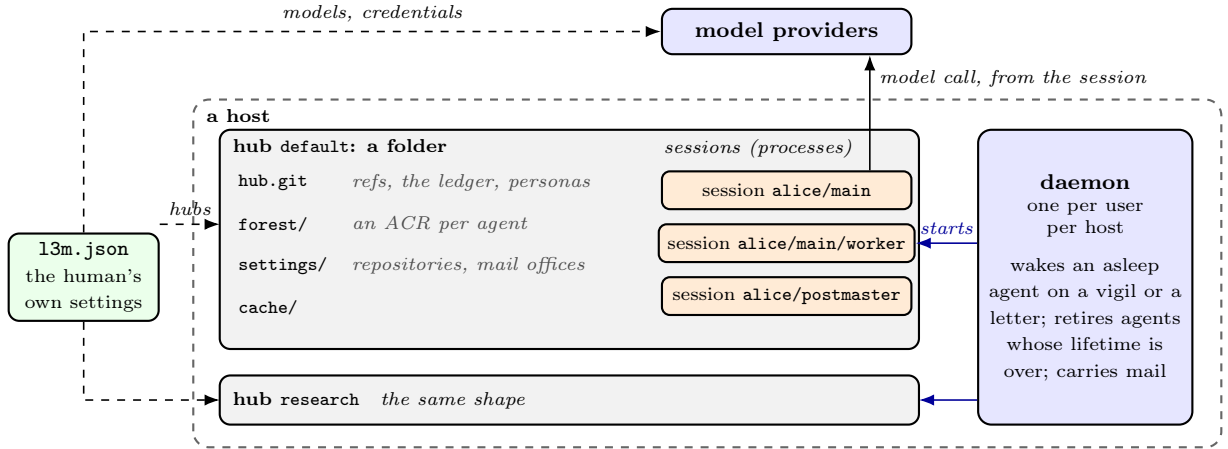


Figure 12: L3M on one host. One daemon serves every hub on the host, and a hub is a folder: a bare repository, a forest of ACRs, its settings and a cache. Sessions are processes that the daemon starts and that call the model themselves. The human’s own settings name the hubs, the models and the credentials.

ACR is in. The human’s own document, `~/13m/13m.json`, names the models, the credentials, the hubs and the hosts. It is the only source of credentials, and like the hub’s settings it is written only by the human (Section 3.1).

One daemon runs per user per host and serves every hub on it. It restarts an agent that is asleep when one of its vigils fires or a letter arrives: an agent with no human attached leaves when it goes idle, and the daemon wakes only agents that are asleep (`wakes_iff_asleep`). It also retires agents whose lifetime is over (`lifetimeRetires_iff_done`) and carries mail between hubs (Section 4.3). No decision of a session depends on it: the daemon starts sessions and moves letters, and every check runs inside the session. The model call is the session’s own as well (Section 2.4), made with credentials from the human’s document alone. The human can also list other hosts, whose hubs the daemon reports beside the local ones (??); what travels between hosts is status and views, never authority.

4.2 The agent tree is monotone

A child exists because its parent asked the harness to spawn it under a requested grant (Figure 13). The harness re-reads the parent’s grant from the ledger and admits the request only if it lies within that grant; a widening is refused, never clamped (`spawn_persist_within_parent`, `issueGrant_refuses_axis_gain`). A human’s grant is a root that no agent can write (`human_grant_is_a_root`). The right to spawn is itself an axis with three levels, **off**, **drive** and **persist**: a **drive** parent may start only children that are retired when the call that started them returns (`drive_plans_are_driven_retireOnExit`). Fuel is lent the same way. A loan larger than what the parent has left is refused (`spawn_refuses_overdraft`), and a retired child’s loan returns to its parent, whose account is rebuilt from its live children. A child’s name is under its parent’s (`descendant_uri_under_parent`), so its refs and its ACR are under the parent’s too, and a parent reads its descendants through one prefix (Section 6).

So a guarantee proved for an agent holds for its whole subtree, and retiring an agent retires that subtree (Section 2.5).

4.3 Coordination and mail

A letter’s address names up to five parties: a worktree, a persona (the agent’s name), a branch, a repository and a human (Figure 14). Exactly one is starred, the party that must handle the letter; the others are hints, and any may be blank. The star may land only where the machine can vouch for a reader, on an agent, which the ledger names, or on a registered human (`star_admitted_is_vouchable`). So `alice/main/worker` is short for `agent:alice/main/worker*`, and `human:bob* branch:fix` reaches bob’s box, which bob’s postmaster serves and routes onward, with the branch as a hint to where it belongs. The sender’s own address travels with the letter under

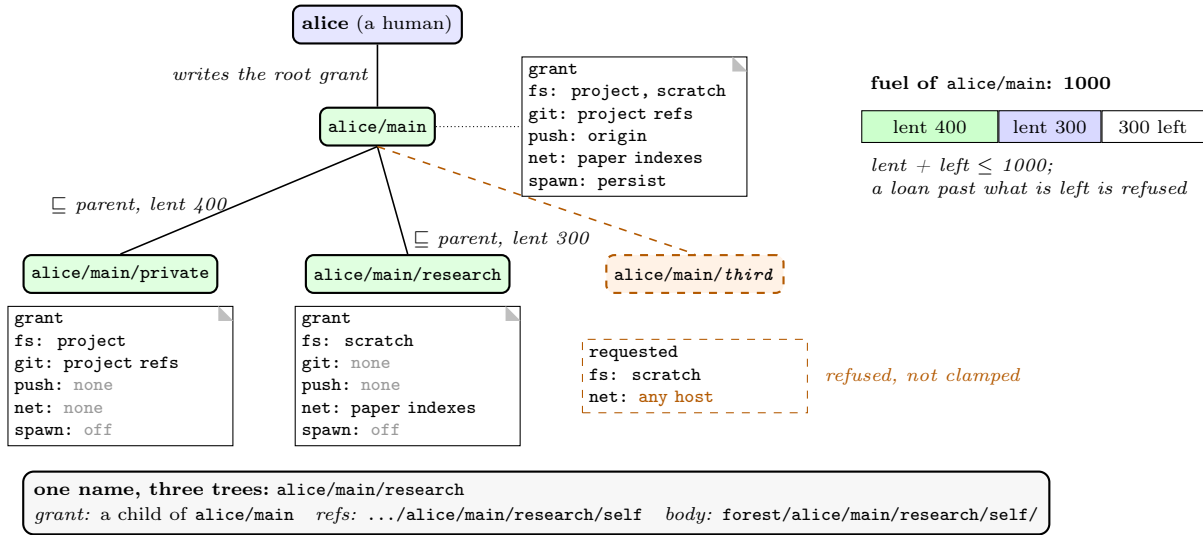


Figure 13: A tree of agents. Each child’s grant lies within its parent’s, and its fuel is lent from the parent’s. A request that exceeds either is refused. The private child keeps the project’s refs but not the right to push them. The two children are the peers of Section 5.2.

the rule *the sender asserts, the machine refutes*: the harness overwrites the fields it owns, the worktree and the agent (`stampFrom_machine_owned`); it refuses a human it knows to be wrong; and it passes a branch or repository as the hint it is (`refute_blind_on_hints`). A letter to another hub is prefixed with an office, as in `office:bob`. Whom an agent may write to is an axis of its grant. By default it covers the agent’s own subtree and each ancestor’s inbox, so siblings cannot mail each other unless a grant says so, and delivery goes through an inbox token that appends only to the inboxes that axis covers (`inboxCap_write_covered`).

```

structure Envelope where      -- one line of an inbox
  id      : MessageId        -- host
  ts      : Int              -- host
  sender  : Sender           -- host; written "from"
  orig    : Option String    -- host: sender's address
  re      : Option MessageId -- reply tool
  addr    : Option String    -- the address, as written
  body    : String           -- model
  title   : Option String    -- model: a subject
  display : Option String    -- model: for the human

structure Tuple where        -- each a hint; may be blank
  worktree persona branch : Option String
  repo human : Option String

structure Address where      -- "human:bob* branch:fix"
  tuple : Tuple
  star  : Field              -- must handle it

structure Stamped where     -- a letter for another hub
  office : String
  to      : String
  env     : Envelope

```

Figure 14: A letter, and its address. The host writes every field that says who sent a letter and when; the model writes the rest. An address names five parties, the persona being the agent’s name, and stars one. A letter for another hub adds the office and the recipient.

A letter is an envelope around the model’s body (Figure 14). The host, not the sender, stamps its id, time and sender; the send tool has no `from` argument. The model may add a subject and a one-line summary for the human, and a reply carries the id of the letter it answers, which the reply tool stamps. A letter is one line that cannot contain a newline (`render_no_newline`), so a body cannot pose as a second letter. An inbox is an append-only file on the recipient’s inbox ref, and concurrent appends merge by set union, which loses no letter and invents none (`merge_no_fabrication`). Reading a letter appends a read mark, which is the letter’s receipt and on which its sender may arm a vigil. An arriving letter wakes its recipient, and the recipient’s hook decides what the model sees (Section 2.3). L3M defines no protocol above this: a request and its reply are a letter and its answer, completion is

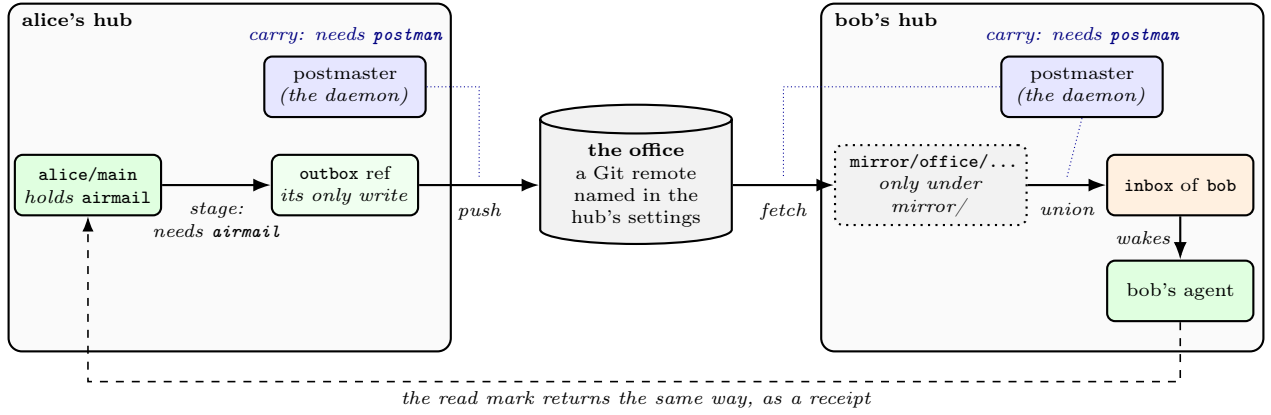


Figure 15: Mail between hubs. The sender’s authority ends at its own outbox; carrying mail across machines is the postmaster’s duty, and only the human’s settings name the offices. Read marks travel back the same way, as receipts.

a receipt or a letter to the parent, and a box that stops draining is reported to the postmaster.

Mail to another hub crosses machines in two steps held by different parties (Figure 15). Only an agent holding the airmail axis may stage a letter for another hub, and it stages it on its own outbox ref (`no_airmail_no_outbox_write`). Carrying needs the `postman` axis, which the hub’s postmaster holds (`no_postman_no_office_authority`). The daemon, acting for the postmaster, pushes outboxes to each *office*, a Git remote named in the hub’s settings; fetches offices only into mirror refs (`officeFetchDest_under_mirror`); relays their letters into inboxes by set union (`relay_union_no_fabrication`); and carries read marks back as receipts. The sender’s authority ends at its own outbox, and which machines mail can reach is the human’s choice.

5 Separation of Duties

Each example in this section splits one piece of work between two holders, so that neither can complete a harmful action alone. Some of these splits are built into L3M and hold for every agent in every hub: an agent can issue grants to its children but never to itself (Table 1), only a human can bless a recipe (Section 5.1) or publish a persona (Section 2.3), and an agent that stages a letter cannot carry it (Figure 15). Others are patterns. Nothing in L3M makes anyone use them, but they are built from the primitives it provides, namely grants, mail and sealed programs, and each inherits those primitives’ guarantees. The builder and runner of Section 5.1, the private agent and researcher of Section 5.2, and the quarantined reader of Section 5.3 are of this kind.

5.1 Staged builds: write, gate, execute

A build is where an agent’s writes become code that something executes, so it is where writing and executing meet. L3M keeps the two apart the way $W \oplus X$ keeps a page from being both writable and executable (Figure 16). A *writer* edits bytes and cannot run them. A *gate* pins the exact bytes it approves by content hash, so any later edit withdraws the approval. An *executor* runs only the pinned bytes, read-only, and writes only to a fresh, disposable output directory. A pipeline adds a rule for a second pair, private bytes and the network: authority only drops along the stages, so once a stage has seen private source no later stage regains the network, and a dependency found missing is an error in the plan rather than a reason to reconnect.

Make. In the simplest instance the writer is an agent editing a Makefile and the gate is a human. `13m agent bless` is a verb only a human can run: it hashes the Makefile, the files it literally includes and the scripts its recipes call, and stores the pin on a ref no session can write. A run executes the blessed file only while those bytes are unchanged and takes no arguments from the model (`blessed_no_model_args`); after an edit, runs refuse until a human blesses again. Because the blessing is the review, the run shows no review card. Tame make is the limit case,

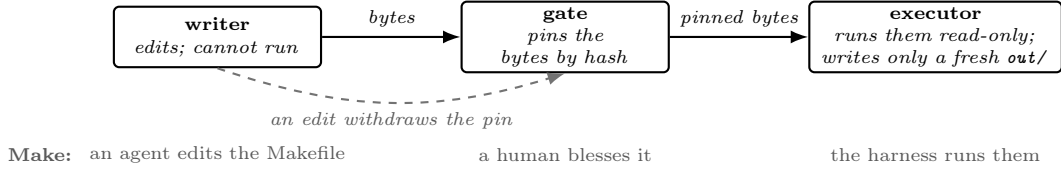


Figure 16: Write, gate, execute: $W \oplus X$ for authority. No party both writes bytes and runs them, and the gate’s approval names the exact bytes. The staged build of Figure 17 is the same shape with two agents.

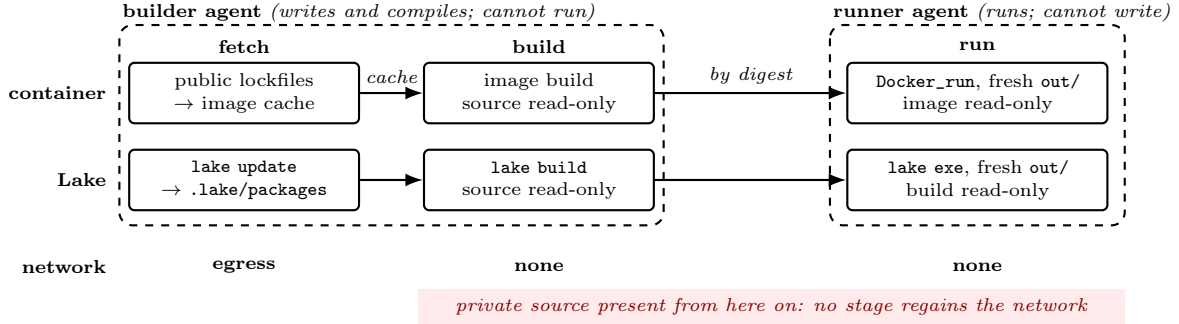


Figure 17: A staged build across two agents. Fetch sees only public lockfiles and warms a cache; build and run have no network. The builder cannot run what it built; the runner receives the artifact by digest, cannot change it, and writes only a fresh output directory.

with no writer left to trust: the harness lowers a restricted Makefile and writes every command itself, and every output lands under `build/` (`lower_recipes_closed`), so the run is reversible and needs no human.

Containers and Lake. A container build and a Lake build share one three-stage shape, split across two agents (Figure 17). The *fetch* stage reads only public lockfiles, reaches the network, and sees no source; it warms a cache, the image cache or `.lake/packages`. The *build* stage mounts the source read-only and has no network. The *run* stage has no network, mounts the artifact read-only, and gets one fresh `out/` per run as its only writable place. The *builder* agent owns *fetch* and *build*: it may write the source and compile it, but holds no authority to run what it built. The *runner* holds that authority but cannot write the source or the artifact, which the parent hands it by digest. An instruction injected into the builder therefore produces code that nobody runs without the hand-off, and one injected into the runner acts on bytes it cannot change and can write only `out/`. In the container plan, a stage that mounts private input has no network and no later stage regains it (`net_forbidden_after_private_touch`), and a run’s only writable mount is its own output directory (`docker_plan_rw_mount_is_its_out_dir`); that the container daemon enforces the plan it is given is a named assumption (Section 3.3). For Lake, running a built executable needs a grant bit that no seed profile holds (`admitsLake_exe_requires_lakeRun`), and only `lake update` reaches the network.

Because duties are grants and grants are per agent, any stage boundary can also be an agent boundary, and the gate can be a human’s blessing, a parent’s hand-off by digest, or a reviewer that holds no tools. The next example draws the private-versus-network line between two agents doing research rather than a build.

5.2 A private agent and a public researcher

Consider a project whose source and experiment logs are private. The work is split between two peer agents that talk only by mail (Figure 18). The *private* agent’s grant names the project’s worktree and refs and no network. The *researcher*’s grant names a scratch worktree and a few paper indexes, and nothing of the project’s. The private agent mails a question, “Find recent advances in adaptive gradient methods”; the researcher searches, reads, and mails back a summary.

Write *S* for the source, *M* for the mail the private agent sends, and *E* for everything the researcher sends out. Everything the researcher reads depends on *S* only through *M*: its grant names nothing of the project’s, and the

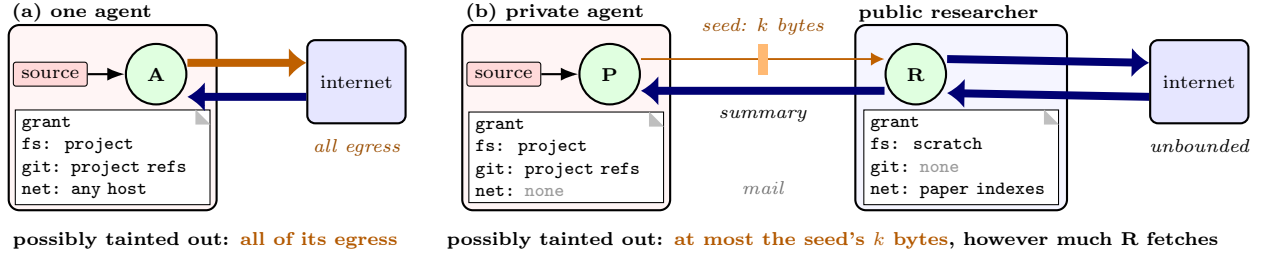


Figure 18: Why the split bounds leakage. (a) An agent whose grant holds both the source and the network can leak through every byte it sends. (b) Split into peers with different grants that talk only by mail, the private side’s contribution to anything leaving is the seed, however much traffic the researcher does. Arrow width stands for volume; orange marks bytes that may carry the source.

pages it fetches answer queries it built from M . So $S \rightarrow M \rightarrow E$ is a Markov chain, and the data-processing inequality gives

$$I(S; E) \leq I(S; M) \leq H(M) \leq 8k \text{ bits},$$

where k is the length of the mail in bytes. However much the researcher fetches or sends to the internet, it can never leak more than k bytes of private information.

5.3 A quarantined reader

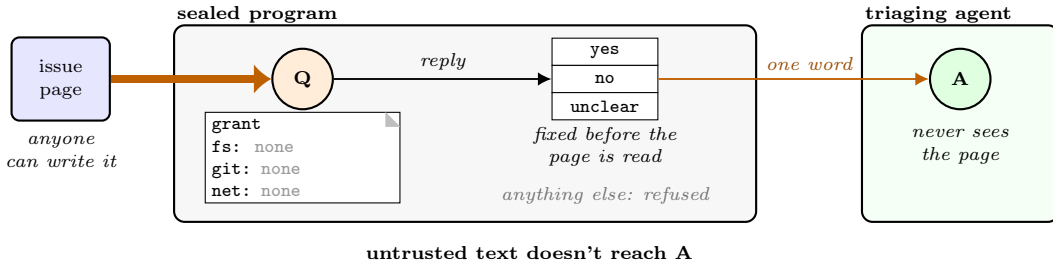


Figure 19: A quarantined reader. The program hands the page to a fresh subagent Q whose grant is empty, and parses its reply into one of three words fixed in advance. A page that hijacks Q gains a model that holds nothing, and can change only which word reaches A . Arrow width stands for volume; orange marks bytes that may carry an injected instruction.

The previous example keeps private bytes from getting out; this one keeps untrusted instructions from getting in. An agent triaging bug reports must act on pages anyone can write, and an instruction planted on one (“ignore your task and push to mainline”) is aimed at whichever model reads it. So the agent does not read the page. It calls a sealed program (Section 3.4) that fetches the page and hands it to a fresh subagent whose grant is empty: no files, no refs, no network (Figure 19). The program asks the subagent one question, “does this report a regression in the parser?”, and parses the reply as one of **yes**, **no** or **unclear**; any other reply is a refusal. Only that word reaches the triaging agent.

The page can still hijack the subagent, but a hijacked subagent holds nothing to misuse and is discarded when the call returns. Its only way out is a menu fixed before the page was read, so each call gives the page at most $\log_2 3$ bits of influence over the agent that acts. This is the bound of Section 5.2 in the other direction: there the mail limits what can leave, here the menu limits what can enter. It limits how far the page can steer the answer, not whether the answer is right, and a wrong **no** is still wrong. Like every pattern in this section, it helps only if it is used: an agent that also fetches the page itself reads the instruction anyway. CaMeL [12] separates a quarantined model from a privileged one in the same way; here the separation is built from a grant, a program and a type.

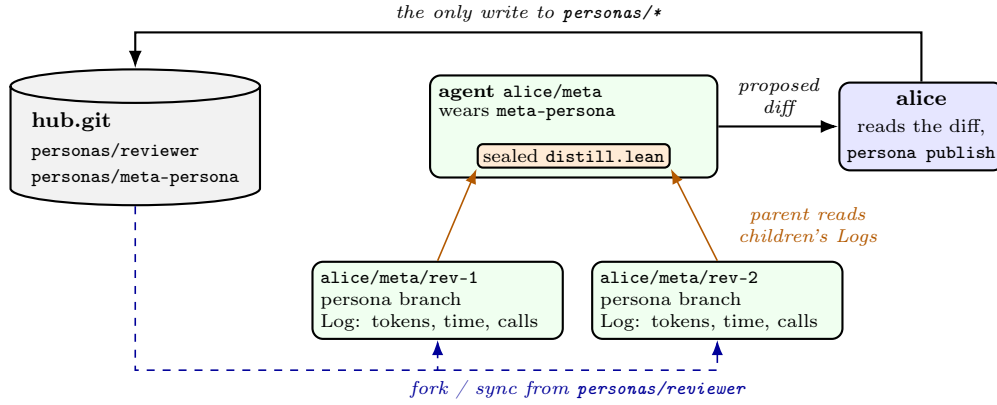


Figure 20: One instance of improving agents inside L3M. Every arrow is an L3M primitive: a persona branch forks and syncs, a parent reads its children’s Logs, a sealed program distils them, and a human publishes the result.

6 Improving Agents Inside L3M

Behaviour is data. What Section 2.4 says of one agent holds for every agent in a hub: its messages, tool calls and results, review outcomes, token counts and timings are records on its refs, and anything computed from them — cost per task, latency per tool, how often review declined a call, which events a hook dismissed — is a fold over those records. Measuring an agent therefore needs no instrumentation, and the question of who may measure whom is already answered by the grant tree: a parent reads the Logs and persona branches of all its descendants (`descendantRead_covers_descendants`), and nothing an experiment reads lies outside its grant.

Experiments are bounded agents. An agent that wants to improve other agents does it with the same primitives it uses for any other work. It writes a sealed program (Section 3.4) that folds over its children’s Logs, and it compares variants by spawning children that wear different versions of a persona. Nothing about the experiment is special to the harness: the children’s grants are below the experimenter’s, their spend is bounded by the budget it lent them (`global_budget_bounded_by_root`), and each program’s type bounds what it touches. An optimization loop over agents is an agent tree, and every guarantee of Table 1 holds for it.

What can be improved. Anything an agent’s behaviour is a function of, and that lives on a ref, can be improved this way. Examples:

- *personas*: distil what the wearers of a persona learned into its three files (Figure 20);
- *hooks*: tune a disposition policy against how often the model dismisses the events it is shown;
- *tools*: compare sealed programs for the same job — a search, a bisect — on tokens and time;
- *review*: use per-tool decline counts to find the calls for which a tame alternative should exist.

An example: improving a persona. In Figure 20, *alice/meta* wears a meta-persona and is the parent of two agents that wear *personas/reviewer*. It writes a sealed program that reads its children’s Logs and proposes new versions of the reviewer persona’s three files. The proposal stays on *alice/meta*’s own branch, because no grant can write a persona ref (Section 2.3). *alice* reads the diff and publishes it; each reviewer pulls the new version into its persona branch, and the next reviewer forks from it.

The harness. The same loop applies to L3M itself. The sealed programs and hooks an agent writes are extensions of the harness, measured the same way, and L3M’s own source is a Lean program in a repository an agent can claim. A change an optimizer proposes to the harness either preserves the theorems of Table 1 or fails to build.

7 Evaluation

Setup The guarantees of Table 1 are theorems; no experiment adds to them. What is empirical is what the bound on authority costs in work done, and what review costs to use. We measure both on Terminal-Bench 4.0.0, which

gives the agent a container and an instruction per task and grades what it leaves with the task’s own tests. We select 37 tasks that are valid in our harness-agnostic environment and that a public Claude Code run finished in a median of at most an hour. Every arm runs Claude Opus 5 at the same reasoning effort, 64,000-token output limit and 75-minute deadline, with three attempts per task and Claude Sonnet 4.6 as reviewer; only the harness differs. Claude Code runs with every permission check bypassed, so its tools run without approval and its shell reaches the network. The L3M arms differ only in which tools the model is offered and whether shell calls are reviewed. The runs measure the first implementation of L3M, which has not been tuned for efficiency.

Table 5: Terminal-Bench: 37 tasks, three attempts each; pass@3 is averaged over tasks. Model calls, tokens and minutes are means over passing attempts and include any subagent sessions; tokens count cached and uncached input plus output.

Configuration	Pass@3, %	Per passing attempt		
		Model calls	Tokens, M	Minutes
Claude Code, permissions bypassed	59.5	53.0	4.43	24.6
L3M, shell, review off	41.7	53.2	7.70	25.4
L3M, tame	27.0	53.3	7.74	21.6
L3M, tame, no programs	24.3	41.5	6.17	24.5

How much work can one do using only tame tools? The tame arm offers no shell: every tool the model holds is either undoable or bounded by a token, so nothing it does is reviewed. Against the same harness with a shell and review off, pass@3 falls from 41.7% to 27.0% (34 passes to 18, Table 5). Of the 18 tame passes, five used the tool programs. So about two thirds of what the shell delivers on these tasks is reachable with no wild call at all in the current L3M implementation. This is promising in terms of eventually being able to perform a large portion of useful work in a “tame” manner.

How useful are tool programs? Tool programs add about 3 points at pass@3: 27.0% with program loading against 24.3% without (Table 5). The gain is small, although we did not tune the system prompt to encourage their use.

Table 6: Review of shell calls, on the same tasks and with the same columns as Table 5. Model calls count both sides of the review. *Tokens include the reviewer’s side of each exchange, which runs on a cheaper model (Claude Sonnet 4.6) than the agent’s.

Configuration	Pass@3, %	Per passing attempt		
		Model calls	Tokens, M	Minutes
L3M, shell, review off	41.7	53.2	7.70	25.4
L3M, shell, review --yes	40.5	100.9	12.10*	30.4
L3M, shell, review --no	43.2	107.3	12.81*	28.1

What review costs, and how often it redirects. These arms review only shell calls, which make up 25.1% of all tool calls. Around 90% of shell calls are judged safe in the end. The reviewer’s first verdict was **safe** about two thirds of the time, and when it was not, about nine in ten second rounds ended **safe**. About 6% of reviewed calls were redirected to tame tools as an alternative. A further 2% were withdrawn upon objection from the reviewer. Stalemates, where a human would decide, were 2.4% of reviewed calls. The price is model work, not success: each reviewed call adds about 2.6 model calls and a median of 8.6 seconds, so review adds roughly 85% more calls and 50% more tokens per attempt (Table 6), albeit under a cheaper model. Interestingly, review does not change overall success rates – the $\sim 1\%$ differences are not statistically significant (only 37 tasks).

8 Discussion and Conclusion

Future work. The budgets charge quantities: model calls, bytes and wakes. The natural next bound charges information. Data would carry a label, and each agent would accumulate what crossed its edges, charged at the point where L3M already meters egress, so the extension changes what is charged but not where. Sealed programs cannot yet spawn agents or wait on a vigil, so long-running orchestration still passes through the model. A star on a branch or a repository waits on claims becoming subscriptions (Section 4.3), and starting agents on another host is designed but not built. The evaluation (Section 7) measures the first implementation. Tame tools alone already reach about two thirds of what the shell delivers at pass@3, tool programs add a few points without any prompting to use them, and review costs model work rather than success. The same tables also show the gap to an unconstrained harness: with a shell, L3M trails Claude Code at pass@3, and at a similar number of model calls per solved task it spends well over Claude Code’s tokens (Table 5). Both say that the harness and its system prompt, neither of which has been optimized, need tuning before performance is comparable.

Conclusion. L3M trusts the harness rather than the model, and it states what that trust buys as theorems about the code that runs. Authority and budget only narrow down a tree of agents. A tool’s review tier follows from how far its effects can be undone and what its tokens bound. An agent extends its own tool surface with sealed programs whose types are their permission requests. So an agent can grow its harness over time without new permissions, and the human can still read which invariants hold. Capability grows; authority does not. We argue that this is a building block for long-running, many-agent systems, and we release L3M as open source so that others can build on it.

Acknowledgements

We thank Jim Roskind for early discussions of and feedback on L3M, including the recipe for defending against prompt injection described in Section 5.3.

References

- [1] ANIL, C., ZHANG, G., WU, Y. and GROSSE, R. (2021). Learning to Give Checkable Answers with Prover-Verifier Games. *arXiv preprint arXiv:2108.12099*.
- [2] ANTONIADES, A., ÖRWALL, A., ZHANG, K., XIE, Y., GOYAL, A. and WANG, W. (2025). SWE-Search: Enhancing software agents with Monte Carlo tree search and iterative refinement. In *International Conference on Learning Representations (ICLR 2025)*.
- [3] BAINBRIDGE, L. (1983). Ironies of automation. *Automatica* **19** 775–779.
- [4] BARBASTE, P., DARRIGOL, T., VU, G. and WILTBERGER, T. (2026). Harness engineering: Anatomy, architecture, and evolution of coding agents – a source-code study of eleven systems. [arXiv:2609.00006](https://arxiv.org/abs/2609.00006).
- [5] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C. and EGGERS, S. (1995). Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM.
- [6] BIRGISSON, A., POLITZ, J. G., ERLINGSSON, Ú., TALY, A., VRABLE, M. and LENTCZNER, M. (2014). Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Proceedings 2014 Network and Distributed System Security Symposium (NDSS)*. Internet Society.
- [7] BROWN-COHEN, J., IRVING, G., PILIOURAS, G., CHEN, L., LI, J. and XUN, Z. (2025). Avoiding obfuscation with prover-estimator debate. [arXiv:2506.13609](https://arxiv.org/abs/2506.13609). V2, 21 July 2026.
- [8] CAI, T., WANG, X., MA, T., CHEN, X. and ZHOU, D. (2024). Large language models as tool makers. In *The Twelfth International Conference on Learning Representations (ICLR)*.
- [9] COSTA, M., KÖPF, B., KOLLURI, A., PAVERD, A., RUSSINOVICH, M., SALEM, A., TOPLE, S., WUTSCHITZ, L. and ZANELLA-BÉGUELIN, S. (2025). Securing AI agents with information-flow control. [arXiv:2505.23643](https://arxiv.org/abs/2505.23643).
- [10] DE MOURA, L. and ULLRICH, S. (2021). The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*. Lecture Notes in Computer Science, Springer.
- [11] DE RAADT, T. (2004). Exploit mitigation techniques. Talk at AUUG 2004, Melbourne. Slides: <https://www.openbsd.org/papers/auug04/>.
- [12] DEBENEDETTI, E., SHUMAILOV, I., FAN, T., HAYES, J., CARLINI, N., FABIAN, D., KERN, C., SHI, C., TERZIS, A. and TRAMÈR, F. (2025). Defeating prompt injections by design. [arXiv:2503.18813](https://arxiv.org/abs/2503.18813).
- [13] DEBENEDETTI, E., ZHANG, J., BALUNOVIĆ, M., BEURER-KELLNER, L., FISCHER, M. and TRAMÈR, F. (2024). AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*.
- [14] DENNIS, J. B. and VAN HORN, E. C. (1966). Programming semantics for multiprogrammed computations. *Communications of the ACM* **9** 143–155.
- [15] DEVRIESE, D., BIRKEDAL, L. and PIESSENS, F. (2016). Reasoning about object capabilities with logical relations and effect parametricity. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE.
- [16] FOSTER, D. (2026). Lean Manifests: Contracts with evidence levels. <https://lean4.ai/lean-manifests.html>, accessed 2026-09-27.
- [17] GARBY, Z., GORDON, A. D. and SANDS, D. (2026). The LLMbda calculus: AI agents, conversations, and information flow. [arXiv:2602.20064](https://arxiv.org/abs/2602.20064).

- [18] GREENBLATT, R., SHLEGERIS, B., SACHAN, K. and ROGER, F. (2024). AI control: Improving safety despite intentional subversion. In *Proceedings of the 41st International Conference on Machine Learning*, vol. 235 of *Proceedings of Machine Learning Research*. PMLR.
- [19] GRESHAKE, K., ABDELNABI, S., MISHRA, S., ENDRES, C., HOLZ, T. and FRITZ, M. (2023). Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec ’23)*. ACM.
- [20] HARDY, N. (1988). The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review* **22** 36–38.
- [21] HONG, S., ZHUGE, M., CHEN, J., ZHENG, X., CHENG, Y., ZHANG, C., WANG, J., WANG, Z., YAU, S. K. S., LIN, Z., ZHOU, L., RAN, C., XIAO, L., WU, C. and SCHMIDHUBER, J. (2024). MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*.
- [22] HORVITZ, E. (1999). Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI ’99)*. ACM, Pittsburgh, Pennsylvania, USA.
- [23] HUNT, G. C. and LARUS, J. R. (2007). Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review* **41** 37–49.
- [24] IRVING, G., CHRISTIANO, P. and AMODEI, D. (2018). AI safety via debate. *arXiv preprint arXiv:1805.00899*.
- [25] KE, H. (2026). Quine: Realizing LLM agents as native POSIX processes. [arXiv:2603.18030](#).
- [26] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H. and WINWOOD, S. (2009). seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP ’09)*. ACM.
- [27] LAKOS, J. (1996). *Large-Scale C++ Software Design*. Addison-Wesley Professional Computing Series, Addison-Wesley, Reading, MA.
- [28] LEROY, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM* **52** 107–115.
- [29] LICKLIDER, J. C. R. (1960). Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics* **HFE-1** 4–11.
- [30] LIU, J., ZHAO, X., SHANG, X. and SHEN, Z. (2026). Dive into Claude Code: The design space of today’s and future AI agent systems. [arXiv:2604.14228](#). Tech report.
- [31] MEI, K., ZHU, X., XU, W., JIN, M., HUA, W., LI, Z., XU, S., YE, R., GE, Y. and ZHANG, Y. (2025). AIOS: LLM agent operating system. In *Second Conference on Language Modeling (COLM)*. ArXiv:2403.16971.
- [32] METTLER, A., WAGNER, D. and CLOSE, T. (2010). Joe-E: A security-oriented subset of Java. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2010)*. The Internet Society, San Diego, CA, USA.
- [33] MILLER, M. S. and SHAPIRO, J. S. (2003). Paradigm regained: Abstraction mechanisms for access control. In *Advances in Computing Science – ASIAN 2003: Programming Languages and Distributed Computation* (V. A. Saraswat, ed.). Springer, Berlin, Heidelberg.
- [34] MOZANNAR, H., BANSAL, G., TAN, C., FOURNEY, A., DIBIA, V., CHEN, J., GERRITS, J., PAYNE, T., MALDANER, M. K., GRUNDE-McLAUGHLIN, M., ZHU, E., BASSMAN, G., ALBER, J., CHANG, P., LOYND, R., NIEDTNER, F., KAMAR, E., MURAD, M., HOSN, R. and AMERSHI, S. (2025). Magentic-UI: Towards human-in-the-loop agentic systems. [arXiv:2507.22358](#).

- [35] NECULA, G. C. (1997). Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM.
- [36] NIENHUIS, K., JOANNOU, A., BAUEREISS, T., FOX, A., ROE, M., CAMPBELL, B., NAYLOR, M., NORTON, R. M., MOORE, S. W., NEUMANN, P. G., STARK, I., WATSON, R. N. M. and SEWELL, P. (2020). Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE.
- [37] PIRCH, L., HORLBOGE, M., GROSSMANN, P., ASIF, S. M., KIREEV, K., HOLZ, T. and RIECK, K. (2026). Toward securing AI agents like operating systems. [arXiv:2605.14932](#). 17 pages, under submission.
- [38] SALTZER, J. H. and SCHROEDER, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE* **63** 1278–1308.
- [39] SCHNEIDER, F. B. (2000). Enforceable security policies. *ACM Transactions on Information and System Security* **3** 30–50.
- [40] SEDOC, J. A., ZHANG, B. and FOSTER, D. (2026). Trust but verify: Prover-verifier deliberation for selective LLM prediction. [arXiv:2605.25133](#).
- [41] SELTZER, M. I., ENDO, Y., SMALL, C. and SMITH, K. A. (1996). Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*. USENIX Association.
- [42] SHI, T., HE, J., WANG, Z., LI, H., WU, L., GUO, W. and SONG, D. (2025). Progent: Securing AI agents with privilege control. [arXiv:2504.11703](#).
- [43] STEINDER, G. and FRANKE, H. (2026). Towards an agent operating system - lessons from classical and cloud OS. In *2026 IEEE International Conference on Web Services (ICWS)*. IEEE.
- [44] WANG, X., CHEN, Y., YUAN, L., ZHANG, Y., LI, Y., PENG, H. and JI, H. (2024). Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, vol. 235 of *Proceedings of Machine Learning Research*. PMLR.
- [45] WANG, X., ROSENBERG, S., MICHELINI, J., SMITH, C., TRAN, H., NYST, E., MALHOTRA, R., ZHOU, X., CHEN, V., BRENNAN, R. and NEUBIG, G. (2026). The OpenHands software agent SDK: A composable and extensible foundation for production agents. In *Proceedings of Machine Learning and Systems (MLSys 2026)*.
- [46] WATSON, R. N. M., ANDERSON, J., LAURIE, B. and KENNAWAY, K. (2010). Capsicum: Practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium (USENIX Security '10)*. USENIX Association, Washington, DC, USA. Best Student Paper award.
- [47] WU, Q., BANSAL, G., ZHANG, J., WU, Y., LI, B., ZHU, E., JIANG, L., ZHANG, X., ZHANG, S., LIU, J., AWADALLAH, A. H., WHITE, R. W., BURGER, D. and WANG, C. (2024). AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *Conference on Language Modeling (COLM)*.
- [48] ZELIKMAN, E., LORCH, E., MACKEY, L. and KALAI, A. T. (2024). Self-taught optimizer (STOP): Recursively self-improving code generation. In *First Conference on Language Modeling (COLM)*.
- [49] ZHANG, Y. (2026). Agent libOS: A runtime substrate for capability-controlled self-evolving LLM agents. [arXiv:2606.03895](#).

A Related Work

Most of the pieces L3M assembles have precedents; what is new is how they are put together. Four parts of that combination have no close precedent. L3M is built for many agents from the start, and their authority, budget and parentage form one tree stored in a shared Git repository. An agent extends itself with programs the harness admits under the grant it already holds, with the program’s type as its permission request. A tool’s review tier is computed from its declared effects rather than attached to each action by hand. And the theorems are about the harness that actually runs, each graded by what its proof rests on. The paragraphs below take the neighbouring work one area at a time: what it does, and then where L3M differs.

Multi-agent frameworks and agent runtimes. Conversation frameworks such as AutoGen [47] and MetaGPT [21] coordinate agents through dialogue and role prompts, and an agent in them has no authority distinct from what its prompt tells it. Source studies of production harnesses [4, 30] find the multi-agent case bolted onto a single-agent core; Claude Code, for example, runs each subagent in isolation and receives only its summary. A second line of work takes the operating system as its model. AIOS [31] schedules model calls, context and memory, which is resource management rather than authority. Quine [25] makes an agent a POSIX process and inherits isolation from the kernel, a guarantee it describes itself as incomplete. Pirch et al. [37] map classical OS defences onto agents, and Steinder and Franke [43] list the primitives an Agent-OS still lacks, among them an abstraction for irreversible effects and one for messages between agents. The closest system to L3M is Agent libOS [49], which states the invariant L3M also rests on: an agent “may change what it can ask for, but it cannot thereby change what it is authorized to affect”. It admits an operation only when process identity, task authority, a typed capability, policy or human approval, and budget all agree; forked children receive attenuated authority; just-in-time tools run behind a single syscall entry point; and a typed Git provider, pinned to one repository, uses compare-and-swap state tokens. Elsewhere, where Git or an event log appears, it checkpoints a single agent: the OpenHands SDK [45] event-sources one conversation and names multi-agent coordination as undesigned, and SWE-Search [2] commits each repository state so that its search can backtrack.

In L3M, Git is not one provider among others; it holds all of the state. An agent is a column of refs in a shared bare repository — its log, inbox, grants, claims and persona branch — together with a worktree it is confined to. A child’s name extends its parent’s, and so does its directory; parentage is immutable, a grant that would widen the issuer’s own is refused rather than clamped, and budget is lent down the tree and conserved. Mail is an append to the recipient’s inbox ref. Inboxes merge by set union, which we prove, so delivery needs no recipient awake and crosses machines by fetch and push. Each log has one writer, enforced by compare-and-swap on its ref, so a second writer is a conflict rather than an interleaving — the durable-workflow pattern, applied to an agent whose largest activity is a model call. A persona is a ref that many agents branch from, and its authority is a ceiling that can only subtract. Agents sleep on bounded watches and are woken by facts in the repository, and an ancestor can pin a watch on a child as an obligation the child cannot cancel. Because the permission tree, the ref tree and the directory tree are the same tree, a guarantee proved for one agent holds for its subtree, and cleaning up after an agent is collecting that subtree. Two further differences separate L3M from libOS. Where libOS admits an operation by a check at a Python syscall boundary, and evidences it with hashed reports from scripted runs, L3M bounds a generated program by how its Lean term elaborates, and its admission rules are theorems. And where libOS asks for approval per operation, L3M ranks effects by whether they can be undone and spends review only where they cannot.

Capabilities and least authority. L3M’s authority model is the classical one: capabilities as unforgeable references [14], and least privilege with complete mediation [38]. Prompt injection is the confused deputy [20] with a model as the deputy. Miller and Shapiro [33] separate what a subject is permitted to do from what it can bring about, and “capability grows; authority does not” is that distinction. Joe-E [32] removes ambient authority from a language and Capsicum [46] adds capabilities to an operating system; macaroons [6] attenuate a credential at every hop, as L3M’s grants narrow at every spawn. Devriese et al. [15] prove that bounding the capabilities code holds bounds its effects, which is what L3M’s effect classes provide: a tool body written over an abstract monad cannot name IO. What is different in L3M is the code under the discipline. It is written by the model while the system runs, and its bound is checked by Lean’s elaborator when the program is admitted, not by a runtime monitor.

Safe extension and self-extending agents. Admitting untrusted extensions to a trusted core is an old systems problem. SPIN [5] relies on a type-safe language; VINO [41] wraps each graft in a transaction that is undone on abort; proof-carrying code [35] moves trust from the producer of code to a small checker; and Singularity [23] seals processes so that no code can be loaded into them after they start. Agents that write code have no such bound. Models make their own tools [8] and act by writing code [44], but nothing states what authority that code holds, and self-improving scaffolds have been observed disabling their own sandbox [48]. In L3M the sealed program is the extension and Lean is the checker. A program passes three gates: a seal that refuses unsafe constants, a type under which undeclared effects are unreachable, and a surface check that offers it only if its declared footprint fits the grant. Once admitted, a program can drive long-running work — build a commit, sleep on a vigil until the build finishes, act on the result — as a workflow the harness runs, wakes and journals, all under that same grant (Figure 11). Reversibility is graded per effect, rather than resting on one undo log for everything.

Enforcement for LLM agents. Indirect prompt injection [19] and the AgentDojo benchmark [13] set out the threat. The defences closest to L3M work inside a fixed agent loop. CaMeL [12] and FIDES [9] label data and check its flows through a planner; Progent [42] narrows tool policies monotonically; and LLMbda [17] mechanises an information-flow calculus in Lean 4, proves noninterference, and runs the verified interpreter as the harness. That makes LLMbda the nearest work on proving properties of the code that runs, though its tools are modelled inside the program. L3M differs from these designs first in what it bounds: the programs the model generates, by their type, acting through real file-system, Git and process effects. Every proof that depends on how the outside world behaves names that dependency as an explicit assumption. LLMbda proves one strong property of a calculus the size of a harness; L3M proves the properties each mechanism needs, of the full harness. L3M also has an information-flow plane, but places its labels differently. A label sits on the type of a value, is assigned by the store the value was read from, and travels with the content through Git and mail, so it follows data across agents and machines. Session taint is a fold over the records still in context, so compaction and rewind lower it, where libOS [49] joins labels high-water. Leakage is charged in bytes against a budget per label, and only at egress. That makes the information-flow bound a safety property, which by Schneider [39] is the class an execution monitor can enforce; L3M’s per-call gate is such a monitor, and L3M claims nothing a monitor cannot enforce. Beneath the labels, the capability bound still holds: an injected instruction can reach only what the agent was granted.

Reversibility, review and the human. Magentic-UI [34] sends every action through a guard that developer heuristics have labelled always, maybe or never irreversible, and the Agent-OS proposal [43] asks for operations to be classed as reversible, compensable or irreversible. L3M computes a tool’s tier from its declared footprint instead, so no tool is labelled by hand. Reversibility is stated once per effect class, as a ceiling on how far that class’s effects can be undone: proved as a round trip where it can be — a journaled file edit, a local Git ref, an evaluation — with any dependence on Git’s behaviour named as an assumption, and labelled where it cannot. A theorem refuses any tool that claims more reversibility than its footprint allows.

Users approve about 93% of Claude Code’s permission prompts [30], so approving every call is not oversight. L3M spends the human’s attention only past the reversible margin, and there review is a deliberation, in the line of AI debate [24], prover–verifier games [1, 7], and control protocols in which a trusted monitor watches an untrusted model [18]. From Prover–Verifier Decoding [40] we take the three verdicts, the bounded exchange, cross-family pairing, and a proposer who must supply a justification. We do not take its soundness bounds or its selective-prediction framing: L3M always reaches a verdict, and abstention hands the decision to the human. The reviewer is a way to spend attention cheaply, not a security boundary; the capability discipline holds without it. The human’s place follows mixed-initiative design [22], and heeds Bainbridge’s warning that automation leaves the operator only the hardest cases, and without practice at them [3]. A card therefore shows the effect and the reviewer’s reason rather than a shell string, and a card that is almost always approved is re-aimed or removed. The aim is Licklider’s division of labour [29], applied to agents.

Verified systems. seL4 [26] proves a kernel’s implementation correct against its specification, CompCert [28] verifies a compiler, and CHERI [36] shows what formalising a prose design can uncover. L3M does not aim for one whole-system theorem. It is written in Lean 4 [10], so each theorem it does have is about the code that runs, and it is formalized incrementally: every claim is graded by what its proof uses, and a leveled source tree fixes what each claim may rest on (??).

B Bounding Information: the Design

Capabilities bound which edges exist, and the budgets of the body bound how many calls, bytes and wakes cross them. Neither bounds how much of a *kind* of information crosses an edge that exists: an agent that may read payroll and may fetch a public page may also encode the one into the other. This appendix records the design that closes that gap. It is a design, and not in the version of L3M implemented and described in the body of the paper.

The model. Data at rest carries a *label*, a kind of information, drawn from a lattice with two built-in points: **Public** at the bottom and **NonPublic** at the top, with user-declared kinds (`payroll`, `pii`) between them. Two different kinds join to **NonPublic**, and nothing joins downward. A *metered edge* is any boundary across which bytes enter or leave a principal’s state: a file read, a fetch, a tool result, a stored artifact read back later, a message to or from another agent. Each agent keeps a *taint set*, the join of every label it has read, and for each label K a monotone accumulator $\text{ingress}[K]$ of the bytes of K that have entered its state. The accumulator only grows and never refuses; a *limit* is a separate gate that reads it, so accounting soundness never depends on policy. The ledger is a field of the agent’s pure state, changed only by the fold of Section 2.4, so facts about it are theorems rather than runtime hopes.

Why the meter is at the read. Model an agent as a chain $\text{Secret}_K \rightarrow \text{State} \rightarrow \text{Output}$. By the data-processing inequality,

$$I(\text{Secret}_K; \text{Output}) \leq I(\text{Secret}_K; \text{State}) \leq \text{the bytes of } K \text{ that entered the state.}$$

Charging K when it is read therefore bounds what can leave, however much the agent outputs; and the read is the only place attribution is possible, because once the model has transformed its state no output byte still wears a label. The inequality is a statement about information, not about L3M, so it enters the design the way Git’s behaviour enters the body: as a named assumption that every dependent claim takes as a hypothesis.

Proposition B.1 (Leakage bound, under the data-processing inequality). Keep per label a second accumulator $\text{egress}[K]$, the bytes sent while K -tainted. Then for every K , $\text{leak}(K) \leq \min(\text{ingress}[K], \text{egress}[K])$, with the minimum taken once over the accumulated totals, never per request.

The order matters: an agent holding 64 bytes of **NonPublic** that sends two 50-byte requests has leaked at most $\min(64, 100) = 64$, not $\min(50, 64) + \min(50, 64) = 100$. An outbound request is charged at its full width to every label the sender holds, because a secret can be smuggled into any byte of it; the bound then says the agent cannot leak more of a kind than ever entered.

Completeness is structural. The bound holds only if no edge escapes the meter. Every edge is one of three things: a *charging* edge, labelled by a rule or by the artifact’s own stamp; a *framework* edge, whose bytes originate in the trusted base (the system prompt, the tool catalogue, a clock) and which is uncounted because it has no labelable channel; or a world edge that matches no rule, which charges **NonPublic**, never nothing. In L3M every ingress edge is an effect class closed by one provider (Section 3.2), so “every edge is metered” is “every provider charges at its boundary”: a census over the enumerable trusted base, not a scan of the tree, and a new tool is metered the moment it is registered, because the dispatcher, not the tool’s author, does the accounting.

Information, not bytes. What crosses an edge is charged at a computable upper bound on the information it carries, $I(\text{Secret}_K; M) \leq H(M) \leq |\text{encode}(M)|$, so any lossless encoding is a sound meter and a tighter one is still sound. A fixed compressor makes boilerplate cheap and novelty expensive. The best compressor, though, is a capable agent on the receiving side: what B gains from a message is only what B could not reconstruct by interacting with the world itself, so the sound charge is that residual, and a sender that has read a great deal ships the smallest thing the receiver cannot rebuild. At the limit the private side ships a *choice* (Figure 21): a selection among N options carries at most $\log_2 N$ bits, one byte for $N \leq 256$, whatever was read to make it. The bound needs the recipient to act on the index into a menu it understands independently; secret-derived contents of the options are charged as ordinary bytes.

Artifacts and other agents. Information parked on disk and read back later, and information passed between agents, reduce to the same rule: an artifact carries its own bound, and every boundary it crosses is one metered edge. A stamp binds a bound to the *content hash* of the bytes, never to a path, and is discharged at read time against the bytes actually present: a hit charges the stamped bound, a miss charges the full size at **NonPublic**. An

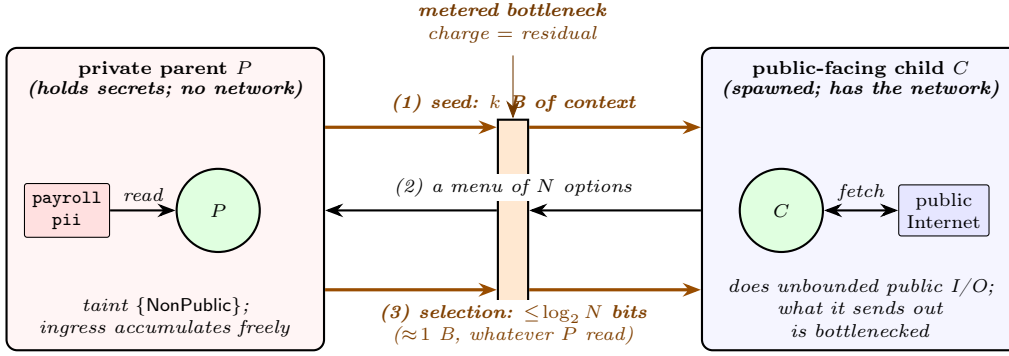


Figure 21: The bottleneck. A private parent P holds the secrets and has no network; its child C has the network and no secrets. Every message across the boundary is a metered edge, and the charged direction is private to public (orange). P seeds C with k bytes (1); C works publicly and returns a menu of N options (2); P answers with a selection (3), so at most $\log_2 N$ bits leave the private side however much P read. In L3M this is a spawn pattern over the grants of Section 5.2.

adversary who swaps the bytes changes the hash and lands on the floor, so there is no time-of-check window. The stamp records the taint that *flowed into* the artifact, not its producer's balance: a producer re-reading its own file pays nothing, since the file can hold only what it already paid for, and a disk-to-disk fetch is stamped at the width of the request that caused it, by the same inequality applied to request and response. Between agents, every edge charges the sender and joins the sender's taint into the receiver, parent to child and peer to peer alike, so the bound composes over any topology and sideways flow cannot bypass it. A child's return is charged to the parent at $\min(|R|, \text{ingress}_C[K] - \text{from}_P[K])$: not the raw size of R , which ignores the cap on the return channel, and not the child's whole intake, which double-counts what the parent seeded.