

CLAUZ3: Static Side-Effect Contracts as the Unit of Consent for Agent-Authored Programs

Chris Mungall
Lawrence Berkeley National Laboratory

Preprint — June 9, 2026

Abstract

Large-language-model agents increasingly act on the world: they send email, move money, read and write files, and call arbitrary tools. The dominant permission models for such agents are either *capability grants* (“this agent may use email”), which are far too coarse to express the constraint a user actually cares about, or *runtime monitors*, which intercept individual tool calls and so can block the second effect of a transaction only after the first has already happened. We present CLAUZ3, a static contract layer for agent-authored Python. The agent writes a short program against an audited *trusted layer* and attaches a machine-checkable contract over its side effects; an SMT-backed prover discharges the contract *before any effect runs*, and the *contract*—not the code—becomes the artifact the user reviews and consents to. CLAUZ3 models a program’s trusted calls as a closed-world, path-conditioned, finite *trace of effect facts*—reasoning about *what the agent did*, not what the world looks like after—and gives a small relational algebra (universal predicates, emptiness, uniqueness, bounded counts and sums, filtered joins) that compiles to Z3. Because the trace is closed-world, allow-lists and emptiness contracts deliver a frame guarantee “for free” without the explicit frame conditions that world-state encodings require. A stateful extension—*fluents*, modeled on Reiter’s situation-calculus successor-state axioms—recovers post-state properties such as “every door is locked at the end” that the unordered model cannot express. The design inherits the proof-carrying-code shape (untrusted producer, small trusted checker, policy as the unit of trust) but inverts who writes the policy: where existing agent-verification proposals fix a policy on the operator side and check code against it, here the *agent* abduces the contract for one specific program and the user accepts, rejects, or asks for a stronger one. We describe the trust model, the contract language and its denotational meaning in Z3, the symbolic-execution engine (built by repurposing `deal`’s `deal-solver`), symbolic iteration over database query results, stateful fluents, the coverage- and entailment-based approval machinery, and worked examples. CLAUZ3’s nearest relatives are proof-carrying code and the recent agentic-verification systems FORGE and Guardians; what sets it apart is that the agent *proposes* the contract and that contract, once proved, is what the user signs off on. It is a research prototype, and we say plainly where trust still rests and what it cannot yet prove.

1 Introduction

Coding agents such as OpenAI’s Codex and Anthropic’s Claude Code have become everyday developer tools, autonomously editing files, running commands, and resolving real software tasks [18, 17]. They are also, increasingly, pointed beyond code—triaging issues, sending mail, querying databases, operating other services—and, increasingly, acting in the physical world, from robotics to home automation to self-driving laboratories [21, 22]. More and more of what they do is not computation but *action*. An autonomous coding or assistant agent that can call tools is, operationally, a program synthesizer with side effects. The user’s safety question is rarely “which capabilities does this agent have?” and almost always something sharper and more specific:

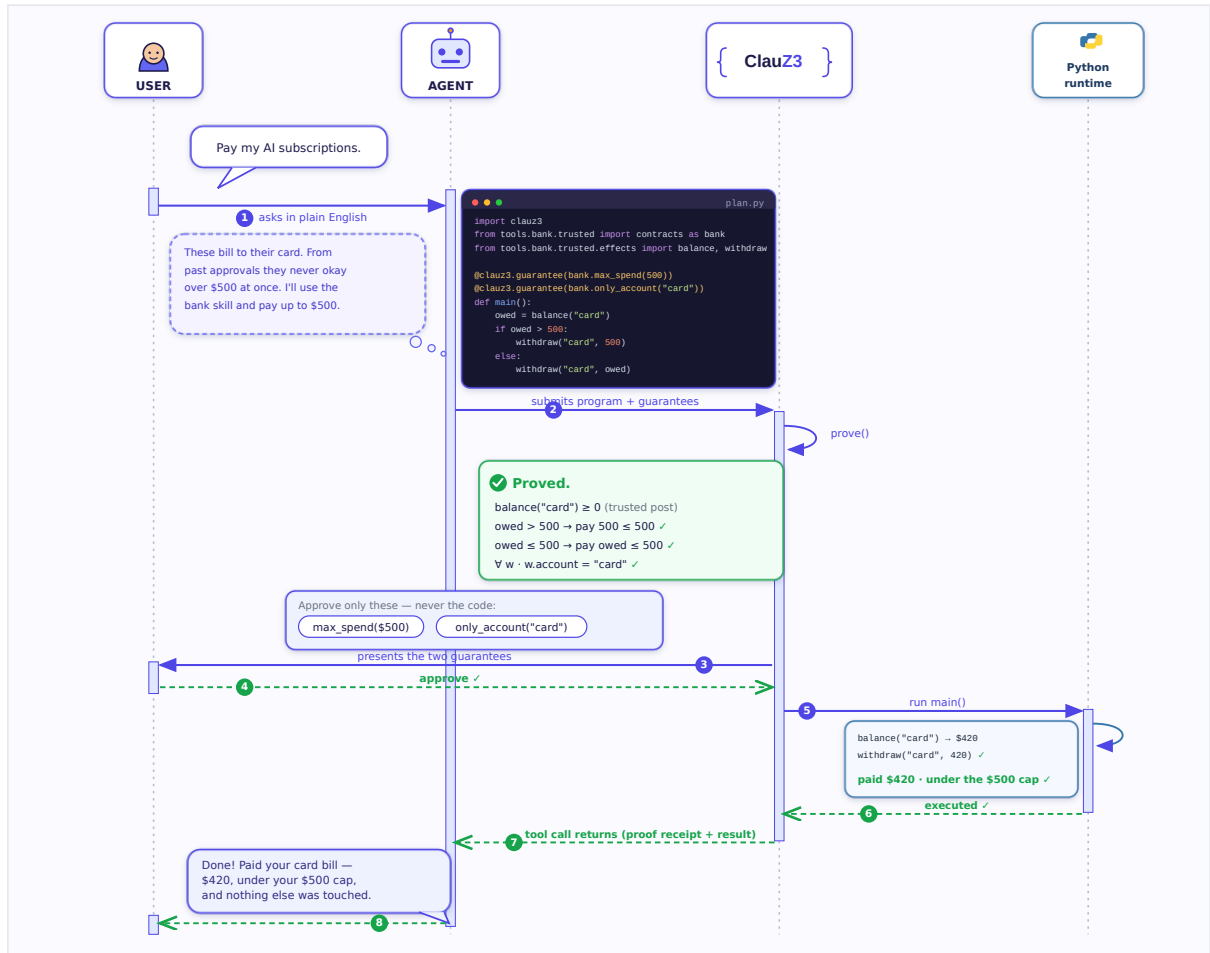


Figure 1: A CLAUI3 run as a sequence diagram. The agent writes an ordinary program and attaches *guarantees* about its side effects; the gate—a static prover over Z3—discharges them and presents *only the proved guarantees*, not the code, to the user; on approval the runtime executes the program and returns its result. The user consents to the proved contract, never to the program text. (Example: a bill-paying program proved to spend at most \$500 from one account.)

- Will this email anyone other than Bob?
- Will it email the same person twice?
- Will it withdraw more than \$5 in total?
- Will it write outside the sandbox directory?
- Will it leave a door unlocked when it is done (in a home-automation or robotic agent)?

A capability grant such as “*this agent may use email*” cannot distinguish “email only Bob,” “email Bob at most once,” “send no email at all,” and “make no promises about email.” These are different permission requests, and collapsing them is a security failure of expressiveness, not of enforcement. The last question is different in kind: it asks not *which* calls happen but about the *state* they leave behind—a distinction we return to with fluents (Section 9).

A different surface, familiar from today’s agentic coding tools, is the per-action approval prompt: the agent emits a script or a shell command and the environment asks the user to approve *that*. This anchors consent to the wrong artifact. The program is often hundreds of lines the user did not write and cannot be expected to audit branch by branch; the intent (“email Bob about the deploy”) is buried among incidental detail—whatever an environment variable resolves to, whatever a subprocess happens to print. Confronted with dozens of such prompts per session, users exhibit *security fatigue* [13], the documented tendency to disengage from security decisions and take the path of least resistance: approving everything, or disabling the permission checks wholesale—the response usable-security research predicts once the effort of compliance outstrips its perceived benefit and the *compliance budget* is spent [14]. The effect is well attested for permission dialogs and security warnings, where comprehension is low and click-through habitual [15, 16]. A prompt that demands real scrutiny on every action, yet is shown far too often to receive it, delivers the appearance of consent without its substance.

The standard answer to coarse capabilities is a runtime policy monitor: a reference monitor sits between the agent and its tools and decides each action as it is attempted. This is genuinely useful and provides defense in depth, but it has an awkward failure mode for programs whose author intends them to run as a single transaction. The agent may send the first of two emails, and only when the second is denied does the user discover that the plan was unsafe. The side effect is already committed; there is nothing left to monitor.

CLAUZ3 explores a different surface (Figure 1). The agent writes a short Python program against a small *trusted layer* of audited, side-effecting functions, and attaches a *contract*: a machine-checkable statement about the program’s trusted side effects, expressed in a reusable domain vocabulary. A prover backed by the Z3 SMT solver [3] either discharges the contract or rejects the program, *statically, before any trusted effect runs*. The contract—a short, checked sentence like “every email goes to an address in this list”—becomes the artifact the user reviews and consents to. The user never has to read every branch of the program; they read the proved contract—an artifact small enough to actually read, which is the point: it aims to restore the *informed* consent that program-by-program approval forfeits to fatigue. Because the decision is made over the whole program ahead of time, execution is all-or-nothing: there is no partial-transaction failure mode. This is the inversion of the PCC, FORGE, and Guardians convention: there, a fixed policy constrains untrusted code; here, the untrusted agent *proposes* the policy and the user grants or refines it.

This shape is borrowed almost directly from proof-carrying code [1, 2]: an untrusted code producer ships a program together with evidence that it meets a safety policy, and the consumer runs a small trusted checker rather than trusting the producer. CLAUZ3 moves trust from the agent to (a) a small audited trusted layer and (b) the prover. Two things distinguish it from classical PCC, and both are consequences of the agentic setting:

Contract abduction.

In classical PCC the consumer fixes the policy and checks code against it. In CLAUZ3 the *agent proposes* the contract for a specific program, and the user accepts it, rejects it, or asks

for a stronger one. The permission request and the policy are the same object. A weak contract is allowed but never silent: an agent that promises little is visibly asking for broad permission.

Side-effect policy, not memory safety.

PCC targeted type and memory safety of running code. CLAUZ3 targets *side-effect contracts*—who is emailed, how much is spent, which files are touched—over a closed-world trace of trusted calls.

Contributions.

1. A *contract-as-consent* architecture for agent permissions, with an explicit three-role trust model (User, Agent, Trusted Layer Engineer) plus an out-of-band approval service (Section 3).
2. A small, generic *relational contract language* over trusted effect facts—universal predicates, emptiness, uniqueness, bounded counts and sums, filters, and existential joins—with no domain-specific logic in the core (Section 4), and its denotational meaning in Z3 (Section 6).
3. A *closed-world, finite, path-conditioned effect-trace* semantics that reasons about *what the agent did* rather than what the world looks like after, yielding a frame guarantee implicitly and avoiding the explicit frame conditions that world-state postcondition encodings require (Section 6), together with a conditional soundness theorem: a discharged contract holds of every concrete execution, modulo an explicit trusted-layer assumption and the supported subset (Section 8).
4. A *stateful* extension—*fluents*—in the style of Reiter’s successor-state axioms [8]: the trusted layer declares how each effect mutates a keyed cell, and contracts query the cell’s *final* valuation (“every door is locked at the end”), recovering the post-state and order sensitivity the multiset model discards (Section 9).
5. A prover built by repurposing `deal`’s symbolic engine [25, 26]: trusted calls are intercepted at `@deal.has` boundaries and recorded as facts, guarantees become Z3 unsatisfiability checks, and bounded loops over typed query results are handled by *symbolic iteration* with universally quantified obligations (Section 7).
6. A consent pipeline: coverage policies that surface or enforce per-domain obligations, an approval service that decides requests by *entailment*, and a runtime backstop in which the same `deal` contracts that the prover discharges statically are still enforced at execution (Section 10).
7. Four worked examples and an explicit comparison with FORGE and Guardians (Sections 11–14), plus a candid account of limitations (Section 15).

CLAUZ3 is an experiment, not a finished product. We are explicit throughout about where trust still resides and what the prototype cannot yet prove.

2 Background

Runtime contracts and `deal`. `deal` [25] is a design-by-contract library for Python. A function is annotated with preconditions (`@deal.pre`), postconditions (`@deal.post`), and *effect markers* (`@deal.has`, e.g. `@deal.has("network")`), which `deal` checks at runtime. CLAUZ3 reuses this exact decorator vocabulary as its *trusted-layer interface language*: a trusted side-effecting function is one marked with `@deal.has(...)`, its `@deal.pre` markers state what must hold before it runs, and its signature defines the schema of the effect. The key move is that CLAUZ3 reads the same decorators and discharges them *statically* ahead of execution, whereas `deal` enforces them *at* execution. Same obligations, two enforcement points (Section 10).

Table 1: Roles and what each is trusted for. The split mirrors assume/guarantee reasoning.

Role	Writes	Trusted for
User	nothing	the final decision; consents to a contract, never reads or runs code
Agent	a program plus <code>@clauz3.guarantee(...)</code> claims about it	nothing; everything it writes must be proved
Trusted Layer Engineer (TLE)	the trusted layer: side-effecting functions <i>and</i> the <code>@contract</code> vocabulary	everything; this is the root of trust
Approval service	nothing	mediating the decision and issuing a receipt that attests the user’s consent; runs outside the agent’s control

SMT and symbolic execution. Z3 [3] is an SMT solver: given a first-order formula over theories (integers, reals, strings, arrays, algebraic datatypes), it decides satisfiability and returns a model or a proof of unsatisfiability. To prove a guarantee g holds under a set of assumptions G , one checks that $G \wedge \neg g$ is unsatisfiable. CLAUZ3’s prover obtains G and g by *symbolic execution* [4] of the agent program: program values become symbolic terms, branches accumulate path conditions, and trusted calls are recorded rather than executed. CLAUZ3 builds on `deal-solver` [26], the experimental Z3 backend for `deal`, which already maps a Python subset onto Z3 sorts and was originally designed to prove a single function’s own pre/post contracts self-consistent.

Proof-carrying code and assume/guarantee. The trust architecture is that of proof-carrying code [1, 2]: trust moves from an untrusted producer to a small trusted checker plus an explicit policy. The split of responsibility—the environment states assumptions, the program states guarantees, and the checker verifies that the guarantees follow from the assumptions—is classical assume/guarantee (rely/guarantee) reasoning [6, 5]. In CLAUZ3 the trusted layer supplies the assumptions and the agent supplies the guarantees.

Prompt injection. A recurring motivation in agentic security is indirect prompt injection [12]: attacker-controlled content in a tool result or document steers the agent into unintended actions. CLAUZ3 does not claim to prevent prompt injection: a manipulated agent can still author a program with a *weak but honest* contract. What CLAUZ3 ensures is that the weakness is visible before the user approves, and that whatever contract is approved provably holds of the program that runs.

3 Trust Model and Threat Boundaries

CLAUZ3 separates four parties by how much they are trusted (Table 1).

The Agent is untrusted. Nothing the agent writes—neither its code nor its guarantees—is believed until the prover discharges it. The agent may propose a weak contract, but the explicit null contract `no_guarantees()` is *semantically* equivalent to “no constraint” while being *syntactically* loud: it forces the agent to say out loud that it promises nothing, and that statement reaches the user and any saved policy.

The TLE is the root of trust. The Trusted Layer Engineer authors the small, audited modules under trusted roots such as `tools/email/trusted/`. These have two halves: (i) *trusted*

effects, the side-effecting functions and their `@deal` markers, which define what an effect *is* and what must hold before it runs; and (ii) *contract vocabulary*, reusable `@contract` predicates that turn raw effect facts into claims worth consenting to. If a trusted stub lies—claims an effect it does not have, or omits a precondition—the proof is only as good as that lie. Keeping this layer small and auditable is the entire security premise.

The approval service is out of band. The service shows the user the proved guarantees, *mediates* the approve/reject decision, and signals `clauz3 run` when the user has consented. It runs *outside* the agent’s control, and the agent cannot route around it: its only path to any side effect is `clauz3 run`, which will not execute the program until the service approves (Figure 1; Section 5). The agent therefore cannot fake the dialog, skip it, or reach the world some other way—not because a token is hard to forge, but because it has no channel that bypasses the gate. Nothing here depends on the service keeping an audit log; for a single `run` it may forget the decision as soon as the program proceeds. (Durable *recording* matters only for remembered, standing consent, which is closer to a limited power of attorney than a one-shot authorization.) The user is granting a scoped, one-program permission, not signing a standing contract.

What is *not* in scope. CLAUZ3 reasons only within the modelled trusted layer. It is not an OS sandbox and not a substitute for one; anything outside the declared trusted effects is neither modelled nor constrained. It has no opinion on how the agent was *talked into* writing a given program. And no language model sits anywhere in the trust path: the check is a deterministic Z3 proof, so “the AI graded its own homework” is not a failure mode.

4 The Contract Language

4.1 Trusted effects become facts

A trusted function is an audited side-effect boundary. The email layer’s sender, for instance, is:

```
import deal

@deal.pre(lambda addr, msg: "@" in addr, message='addr must contain "@')
@deal.has("trusted")
def send_email(addr: str, msg: str) -> None:
    ...
```

When the prover reaches a reachable call `send_email("bob@x.com", "hi")`, it does *not* execute the body. It discharges the precondition as a proof obligation and records a symbolic *effect fact*:

```
FactInfo(
    name="send_email",
    markers=("trusted",),
    args={"addr": <symbolic str>, "msg": <symbolic str>},
    cond=<reachability / path condition>,
    quantifiers=(), # loop frames, see 6.4
)
```

This is fully generic. The core does not know what a withdrawal, an email, or a file write *means*; it records the trusted function’s name, its `@deal.has` markers, the symbolic arguments bound from the call site, and the path condition under which the call is reachable. Every contract in the system is a relational constraint over this set of facts.

Table 2: The relation algebra over effect facts. F is the set of recorded facts of a relation, c_f the path condition of fact f , and predicates/keys are Z3 terms produced by compiling a one-argument lambda over the fact’s argument row.

Primitive	Reads as	Meaning (informal)
<code>R.all(p)</code>	$\forall$	every reachable row satisfies p
<code>R.empty()</code>	$\nexists$	no reachable row exists
<code>R.distinct(k)</code>	uniqueness	no two reachable rows share key k
<code>R.count() ≤ n</code>	cardinality	at most n reachable rows
<code>R.sum(s) ≤ n</code>	aggregate	$\sum$ of selector s over reachable rows is bounded
<code>R.where(p)</code>	filter	restrict R to rows satisfying p (composes with the above)
<code>a.shares_value(b,k)</code>	$\exists$ -join	some reachable a -row and b -row share key k
<code>no_guarantees()</code>	$\top$	the transparent null contract

4.2 Effect relations

`effect(name)` creates a relation view over the recorded facts. The `name` may be a trusted function name (`send_email`, `withdraw`) or a `@deal.has` marker (`trusted`, `network`); the function-name form gives a precise row schema drawn from the signature, whereas markers group cross-cutting effects more loosely. A relation exposes a small algebra (Table 2). Domain predicates are then authored once by the TLE as `@contract` helpers:

```
from clauz3.spec import ContractSpec, contract, effect
from clauz3.spec import no_guarantees as core_no_guarantees

Email = effect("send_email")

@contract
def only(addresses: list[str]) -> ContractSpec:
    return Email.all(lambda e: e.addr in addresses)

@contract
def unique_recipients() -> ContractSpec:
    return Email.distinct(lambda e: e.addr)

@contract
def recipient_at_most(addr: str, count: int) -> ContractSpec:
    return Email.where(lambda e: e.addr == addr).count() <= count

@contract
def no_guarantees() -> ContractSpec:
    return core_no_guarantees()
```

4.3 Agents state guarantees in that vocabulary

The agent program imports the vocabulary and attaches guarantees to the function under proof. `@clauz3.guarantee(...)` is a runtime no-op; it exists purely for the prover to read off the AST.

```
import clauz3
from tools.email.trusted import contracts as emails
from tools.email.trusted.effects import send_email

@clauz3.guarantee(emails.only(["bob@example.com"]))
@clauz3.guarantee(emails.unique_recipients())
def main() -> None:
```

```
send_email("bob@example.com", "Report is ready")
```

The two decorators *are* the permission request the user reviews. Note the clean separation of concerns: `only` and `unique_recipients` are written once, by the TLE, in terms of the generic `send_email` relation; the *email-specific* knowledge lives entirely in the trusted layer, and the core prover contains no email-, bank-, or database-specific logic.

4.4 The lambda subset

Predicate and key lambdas are not executed. CLAUZ3 parses their source to an AST and compiles a deliberately small expression subset into Z3 terms: field access (`e.addr`), constants, list literals, the comparisons `== != < <= > >=` `in not in`, boolean `and/or/not`, numeric `+/-`, `len(...)`, and the string methods `startswith/endswith` (which lower to Z3's `PrefixOf/SuffixOf`, enabling path-prefix policies such as `e.path.startswith(root)`). Anything outside the subset raises `UnsupportedError`—the language *fails closed* (Section 7).

5 Running It: Mediated Tool Access

Before the formal semantics, it is worth being concrete about what it takes to *deploy* CLAUZ3, because the whole argument rests on one operational precondition. The guarantees mean nothing if the agent has a side channel. As Figure 1 shows, every interaction with the environment must be *mediated by* CLAUZ3: the gate is the only path from the agent to any side-effecting tool. Concretely, the harness must **deny the agent every tool except clauz3**—no arbitrary shell, no file writes, no network, no ad-hoc subprocess—and the trusted effects (`send_email`, `withdraw`, `unlock_door`, ...) must be reachable *only* through `clauz3 run`. CLAUZ3 is a contract layer, not an OS sandbox; if the agent can run code through some other permitted tool, the mediation is bypassed and no guarantee survives.

Locking the agent to the gate. In a Claude Code deployment this is a permission policy. `clauz3 config` writes a `.claude/settings.json` that allows only read-only inspection plus the CLI:

```
{ "permissions": { "defaultMode": "default",  
  "allow": ["Read", "Glob", "Grep", "Bash(clauz3:*)"] } }
```

The agent may read the repository and invoke `clauz3`; anything else falls outside the allowlist and is refused (or escalated to the user). The trusted layer is dropped into the repository with `clauz3 install`, which copies a `tools/<domain>/trusted/` package; the agent then writes ordinary Python against those functions and attaches guarantees, exactly as in Section 4.

The run loop. `clauz3 run` is the single mediated entry point and performs the three steps of Figure 1 in order: it (i) proves the target's guarantees, refusing to proceed unless every obligation is OK; (ii) submits a JSON approval request—the proved guarantees and coverage, not the code—to an approval service and blocks; and (iii) executes the program only after the service returns an approval *receipt*. The agent never touches steps (ii) or (iii) directly; it calls `clauz3 run` and waits.

The approval service is separate and out of band. The service runs as its own process, started and configured by the user or the harness—*not* by the agent, which is forbidden from starting, replacing, or reconfiguring it. `clauz3 approval-service` brings up a local service with a browser UI; with `--policy` it can auto-decide requests by entailment and otherwise fall through to a human (Section 10). Since the agent's only route to execution runs through `clauz3`

`run`, which blocks on this separate service, it cannot fake the dialog, skip it, or run the program by another path. As defense in depth, the runner that executes the approved program also drops dangerous builtins and imports, but that hardening is secondary: the load-bearing boundary is the harness admitting nothing but `clauz3`.

6 Semantics: A Closed-World Effect Trace

6.1 Facts and path conditions

Symbolic execution of the target function produces a finite set F of effect facts. Each fact f carries a reachability condition c_f , a Z3 boolean term that is true exactly on the executions in which the call occurs. Branches fold into c_f : a fact recorded inside `if c: ...` has its condition rewritten to $\text{ite}(c, c_f, \perp)$, and the else-branch dually. Thus a fact is a record of (trusted-function name, schema-typed argument row, reachability condition, optional loop quantifiers); the name matters because the representation lemma of Section 8 identifies facts with named call occurrences. F is an *indexed family*, not a set: two reachable calls with identical name and arguments (say, an unrolled loop that sends the same email twice) are two facts.

Crucially, the trusted boundary is the *only* channel through which a modelled side effect can occur, and the prover enumerates all reachable trusted calls. The trace is therefore *closed-world*: an effect that is not in F provably does not happen.

6.2 Denotation of the primitives

Writing $\llbracket p \rrbracket_f$ for the Z3 term obtained by compiling lambda p against f 's argument row, the primitives denote:

$$\begin{aligned} \llbracket \text{R.all}(p) \rrbracket &= \bigwedge_{f \in F} (\neg c_f \vee \llbracket p \rrbracket_f), \\ \llbracket \text{R.empty}() \rrbracket &= \bigwedge_{f \in F} \neg c_f, \\ \llbracket \text{R.distinct}(k) \rrbracket &= \bigwedge_{\{f, f'\} \subseteq F} (\neg c_f \vee \neg c_{f'} \vee \llbracket k \rrbracket_f \neq \llbracket k \rrbracket_{f'}), \\ \llbracket \text{R.count}() \leq n \rrbracket &= \sum_{f \in F} \text{ite}(c_f, 1, 0) \leq n, \\ \llbracket \text{R.sum}(s) \leq n \rrbracket &= \sum_{f \in F} \text{ite}(c_f, \llbracket s \rrbracket_f, 0) \leq n, \\ \llbracket a.\text{shares_value}(b, k) \rrbracket &= \bigvee_{f \in A, f' \in B} (c_f \wedge c_{f'} \wedge \llbracket k \rrbracket_f = \llbracket k \rrbracket_{f'}). \end{aligned}$$

`where(q)` simply conjoins $\llbracket q \rrbracket_f$ into each c_f before the above. Because reachability is folded in by the primitive, two facts in mutually exclusive branches may legitimately share a `distinct` key (both cannot be reachable at once), and unreachable facts contribute 0 to a sum. Contract authors never write “if reachable then ...” by hand. The displays above are stated for quantifier-free facts; a fact carrying a loop frame stands for one effect *per iteration*, and each primitive lifts to that case differently (Section 7.4).

6.3 The frame guarantee, for free

A recurring difficulty in verifying effectful plans is the frame problem [7]: a postcondition that merely asserts the intended effect (“`foo.txt` and `bar.txt` are deleted”) is also satisfied by an over-broad implementation (`delete("*.txt")`), so one must add an explicit *frame condition* stating that everything else is unchanged. Encodings that reason over world state (a `fileSystem` variable, say) must carry such frame conditions or the solver will satisfy the obligation the cheapest way it can.

CLAUZ3 sidesteps this because it reasons over a closed-world finite trace of *effects* rather than over world state. A contract like `Email.only(...)` or `filesystem.only_write_under(root)` bounds the *entire* effect set directly: every fact must satisfy the predicate, and there are provably

no facts outside F . The frame guarantee for the effects the prover tracks is implicit in the universal quantification over a closed set. This is a structural difference from world-state encodings such as the Z3/Dafny obligations in Guardians (Section 14), and it is one of the design’s more pleasant properties.

7 From Python to Z3: The Prover

CLAUZ3’s prover is built by vendoring and extending `deal-solver`. Its original job—prove one function’s own `@deal.pre/@deal.post` self-consistent—is repurposed to prove side-effect contracts over a trace of trusted calls. We describe the pipeline and the additions.

7.1 Proxy sorts

Every symbolic value is a *proxy* wrapping a Z3 expression and exposing Python-operator methods (`m_eq`, `m_add`, `m_len`, ...). Python types map to Z3 sorts: `int`→`Int`, `str`→`String`, `bool`→`Bool`, with `list`, `set`, `dict`, and tuple proxies layered on top. Function-argument annotations are turned into fresh symbolic constants of the corresponding sort; an unannotated or unsupported argument makes the theorem unprovable.

7.2 Recording facts at `@deal.has` boundaries

When symbolic execution reaches a call, the prover inspects the callee’s contracts. If it carries one or more `@deal.has` markers, the call is a trusted boundary and is handled specially: (i) the callee’s `@deal.pre` preconditions are evaluated at the (symbolic) call arguments and added to the set of proof obligations; (ii) the body is *not* executed; (iii) a `FactInfo` is appended to the trace, with `cond` set to the current not-interrupted-and-reachable condition; (iv) a dummy value is returned to the evaluator. Trusted functions are thus modelled today as `None`-returning side-effect stubs (the one exception, query results, is Section 7.4).

7.3 Guarantees as unsatisfiability checks

After executing the body, the prover assembles the *givens* G : the target’s own preconditions, the trusted postconditions assumed at each boundary—each guarded by that call’s reachability condition, $c_f \Rightarrow post_f$ —and background axioms (e.g. lengths are non-negative). The guard on assumed postconditions is load-bearing: assumed *unguarded*, a declared postcondition that happens to be unsatisfiable (a TLE typo, say) on a call that is *unreachable* would make G inconsistent and every contract vacuously provable, without the trusted layer ever violating assumption (A1) of Section 8, which constrains only calls actually invoked. For each guarantee g (obtained by evaluating the decorator’s argument, which dispatches into the `@contract` helper and ultimately the relation denotations of Section 6), it forms one proof obligation and asks Z3:

$$\text{is } G \wedge \neg g \text{ unsatisfiable?}$$

If unsat, g is proved (OK). If sat, Z3’s model is a concrete counterexample and the obligation FAILs. If Z3 returns `unknown`, the obligation is SKIPPed. A theorem’s overall conclusion aggregates per-obligation results: any FAIL is fatal; all-OK is OK; a mixture of OK and SKIP is PARTIAL. Any runner must treat SKIP/PARTIAL as “not approved”—the system *fails closed*.

7.4 Branches, bounded loops, and symbolic iteration

Branches. `if/else` is symbolic: both arms run in child contexts and their variables, returns, exceptions, and *facts* are merged into the parent gated on the branch test, as described in Section 6. There is no path explosion: paths are merged with `ite`, not enumerated.

Concrete unrolling. A `for` loop over a literal list/tuple or `range(N)` with literal N is unrolled: the body is re-evaluated once per element, emitting ordinary concrete facts. This is what lets the bank example prove a total-withdrawal bound over `range(5)` via the `sum` aggregate (Section 11).

Symbolic iteration. The harder case is iterating over data the prover has not seen. A trusted query returns a typed list of rows:

```
class UserRow(clauz3.Row):
    name: str
    email: str
    consented: bool
    role: str

@deal.has("db_read")
@deal.post(lambda result: len(result) <= 100)
def db_query(table: str, where: dict) -> list[UserRow]: ...
```

A `list[UserRow]` return is modelled as a Z3 array $\text{Int} \rightarrow \text{UserRow}$ (with `UserRow` compiled to an algebraic datatype with one accessor per field) plus a symbolic length. When the agent writes `for row in db_query(...)`, the prover binds `row` to `Select(array, i)` for a fresh index i , pushes a *quantifier frame* $0 \leq i < \text{len}$, executes the body *once* symbolically, and pops the frame. Facts emitted inside the loop carry that frame, and each relation primitive lifts to framed facts in its own way. `all` and `empty` wrap their per-fact clause in $\forall i. (0 \leq i < \text{len}) \Rightarrow \dots$. `distinct` needs *two* kinds of clause: cross-fact pairs are quantified over both facts’ frames, and a framed fact additionally contributes a *same-fact* clause $\forall i \neq j. \dots \Rightarrow \llbracket k \rrbracket_{f[i]} \neq \llbracket k \rrbracket_{f[j]}$, catching duplicate keys *within* one loop. A summand cannot be wrapped in a universal, so `count` and `sum` instead multiply a framed fact’s contribution by the loop length, $\text{ite}(c_f, \llbracket s \rrbracket_f \cdot \text{len}, 0)$; this is exact only when neither the selector nor the fact’s condition depends on the index, so a selector that does is rejected (`UnsupportedError`), and a condition that does (an effect under a branch inside the loop) leaves the multiplied form sound for the offered bound comparisons but conservative. `shares_value` over any framed fact is rejected outright. Preconditions are discharged for *every* iteration: each `@deal.pre` obligation is wrapped in the same universal. Finally, `v1` restricts the loop body syntactically—`break`, `continue`, and `return` are rejected—while the two remaining side conditions of Lemma 1 (no state carried across iterations, no exceptions) are *assumed, not yet detected* (Section 15). “Email every consenting user” is thus proved without ever knowing how many users there are or who they are. A *column-binding* contract such as `addresses_from(UserRow, "email")` is checked structurally: the prover verifies that each `send_email` fact’s `addr` term has the syntactic shape `email(Select(query, .))` with the expected schema—catching “emailed the `name` column instead of `email`” as a proof failure.

8 Soundness

The denotations of Section 6 live in Z3; this section says what it *means* for a real execution to obey a contract, and under which assumptions a discharged proof guarantees it. We give a deliberately lightweight account—a concrete-trace semantics, a representation lemma, and a conditional soundness theorem with proof sketches—rather than a mechanized development over a full operational semantics of the Python subset, which would be a separate effort.

8.1 What the proof is sound *against*

A contract constrains only trusted side effects, and the relation algebra is insensitive to their order, so the object an execution exposes to a contract is a finite *multiset* of effect occurrences.

Definition 1 (Concrete trace). Fix a *valuation* σ : concrete values for the target’s symbolic arguments together with the concrete results of every trusted *read* (e.g. a query’s returned rows). Under σ the program follows one concrete path; its *concrete trace* M_σ is the finite multiset $\llbracket (n_j, v_j) \rrbracket$ of trusted calls it makes, each a function name n_j with a concrete argument row v_j .

That M_σ is well defined and finite is a property of the supported subset, not an extra assumption: the subset has no **while**, no recursion, and only loops bounded by a literal or by a read result’s length, so execution under σ is deterministic and terminating.

Definition 2 (Concrete satisfaction). For a relation R write $M_\sigma \upharpoonright R$ for the sub-multiset of calls whose name or marker is R ’s. The relation $M_\sigma \models C$ is defined structurally, reading each primitive as its plain mathematical meaning over that multiset:

$$\begin{aligned} M_\sigma \models \mathbf{R.all}(p) &\Leftrightarrow \forall a \in M_\sigma \upharpoonright R. p(a), \\ M_\sigma \models \mathbf{R.empty}() &\Leftrightarrow M_\sigma \upharpoonright R = \emptyset, \\ M_\sigma \models \mathbf{R.distinct}(k) &\Leftrightarrow |M_\sigma \upharpoonright R| = |\{k(a) : a \in M_\sigma \upharpoonright R\}|, \\ M_\sigma \models \mathbf{R.count}() \leq n &\Leftrightarrow |M_\sigma \upharpoonright R| \leq n, \\ M_\sigma \models \mathbf{R.sum}(s) \leq n &\Leftrightarrow \sum_{a \in M_\sigma \upharpoonright R} s(a) \leq n, \end{aligned}$$

with **where** restricting the sub-multiset, **shares_value** the evident existential, and **no_guarantees** always true. In the **distinct** clause the left side counts *occurrences* (multiset cardinality) and the right side is the *set* of keys, so two occurrences with identical argument rows—the same call made twice—violate **distinct**; a value-level reading ($\forall a \neq a'. k(a) \neq k(a')$) would let exact duplicates slip through vacuously. This is exactly the sentence a user reads off the approval dialog.

8.2 Representation: the fact set captures the trace

Symbolic execution of the target produces the fact set F (Section 6); each $f \in F$ carries symbolic arguments $args_f$ over the inputs and read-result arrays, and a reachability term c_f . A valuation σ assigns each a concrete truth value $\sigma(c_f)$ and concrete arguments $\sigma(args_f)$.

Lemma 1 (Single-shot loop adequacy). *Let for row in $q()$: B be a loop whose iterand q is modelled as a read array A of length ℓ , where (i) B carries no state across iterations, (ii) B contains no **break**, **continue**, or **return**, and (iii) no statement of B can raise—an exception would short-circuit the remaining iterations exactly as **break** does. (Section 7.4 records which of these v1 enforces.) Then the loop’s concrete effects are $\biguplus_{0 \leq i < \ell} \mathbf{effects}(B[\text{row} := A[i]])$, and the single quantifier-framed template fact emitted by executing $\bar{B}$ once at a fresh index, expanded over $0 \leq i < \text{len}$, equals this multiset.*

Proof sketch. By (i), iteration i ’s symbolic state and emitted facts are a function of i and A alone, with no dependence on iterations $< i$; so the body’s effect at concrete index i is obtained by instantiating the single symbolic body at i . By (ii)–(iii) every index $0.. \ell - 1$ is reached and none is skipped or short-circuited. Hence executing the body once under a fresh symbolic i and universally quantifying over $0 \leq i < \text{len}$ is equivalent to the disjoint union over concrete i . Conditions (i)–(iii) are therefore not incidental limitations but the side conditions that make “execute once, then quantify” sound. $\square$

Lemma 2 (Representation). *For every valuation σ within the supported subset,*

$$M_\sigma = \llbracket (\text{name}_f, \sigma(args_f)) : f \in F, \sigma(c_f) = \top \rrbracket,$$

reading F as the indexed family of Section 6, so that two facts with equal name and arguments contribute two occurrences to the multiset.

Proof sketch. The trusted boundary is the only channel through which a modelled effect occurs (*closed world*), and symbolic execution records a fact at every reachable trusted call. Straight-line code and trusted calls add their occurrence with $c_f = \top$ on the taken path; **if/else** gate each recorded c_f by the branch test, so under σ exactly the facts on σ 's path have $\sigma(c_f) = \top$; literal-bounded loops are unrolled into concrete facts; and symbolic loops are handled by Lemma 1. Induction on the program's statement structure then identifies the σ -image of the reachable facts with the calls actually made. $\square$

8.3 Conditional soundness

We make the trusted computing base explicit as three assumptions—one per source of trust in the system (the trusted layer, the prover's coverage of the program, and the solver). Note none of them is a claim about the *agent*: the agent is untrusted, so soundness requires no hypothesis about its behaviour.

(A1) Trusted-layer faithfulness.

Each trusted function's real runtime effect is exactly one occurrence of $(name, args)$, it is invoked only when its `@deal.pre` holds, and its declared `@deal.post` markers hold of its results. (This is the TLE's obligation; if a stub lies, soundness is void.)

(A2) Coverage.

Every construct the target uses lies in the supported subset, so the proof yields **OK** rather than **SKIP**/**PARTIAL**.

(A3) Solver soundness.

An **unsat** verdict from Z3 is correct.

Theorem 1 (Soundness modulo the trusted layer). *Assume (A1)–(A3). If the prover reports **OK** for a contract C on the target—i.e. $G \wedge \neg \llbracket C \rrbracket$ is unsatisfiable, where G collects the givens of Section 7—then for every valuation σ that satisfies the target's own declared preconditions, the concrete trace satisfies the contract: $M_\sigma \models C$. (The restriction is forced because G assumes those preconditions; the runtime backstop of Section 10 enforces exactly this hypothesis at the call boundary.)*

Proof sketch. **OK** means $\llbracket C \rrbracket$ holds in every model of G . Any valuation σ admitted by (A1) that satisfies the target's preconditions is such a model: the givens are those preconditions (σ satisfies them by hypothesis), the *guarded* trusted postconditions $c_f \Rightarrow post_f$ (true under (A1) when the call occurs, vacuous when it does not), and the background axioms; and σ respects the path conditions. By the denotations of Section 6, $\llbracket C \rrbracket$ evaluated at σ is the same Boolean as concrete satisfaction of the same shape over the multiset $\{ (name_f, \sigma(args_f)) : \sigma(c_f) = \top \}$ (each primitive's symbolic clause is the pointwise image of its concrete clause: universal becomes $\bigwedge$, cardinality becomes $\sum$ ite, etc.). By Lemma 2 that multiset is M_σ . Hence $M_\sigma \models C$. By (A3) the **unsat** verdict establishing **OK** is not spurious. $\square$

Corollary 1 (Frame, for free). *For $C = R.all(p)$ or $R.empty()$, Lemma 2 makes $M_\sigma \upharpoonright R$ exactly the σ -image of R 's facts, so the contract bounds the entire concrete R -effect set; there are provably no R -effects outside it. The frame guarantee of Section 6.3 is thus a corollary of closed-world representation, requiring no separate frame condition.*

Corollary 2 (Sound, not complete). *Theorem 1 gives no guarantee when (A2) fails: a **SKIP** (Z3 returns **unknown**; **PARTIAL** in aggregate) or an out-of-subset construct (**UnsupportedError**) voids the hypotheses, and a runner must treat any of these as rejection (fail-closed). Dually, a **FAIL** exhibits a model of $G \wedge \neg \llbracket C \rrbracket$ —a valuation σ whose concrete trace M_σ violates C . Two caveats make that counterexample genuine relative to Definition 1, not unconditionally: it presumes the encoding is exact on the supported subset (an over-approximating step would admit*

spurious models), and its read results are ones the declared *postconditions* permit—a world the trusted layer allows, not necessarily one the actual environment can produce.

9 Fluents: Stateful Contracts

The relation algebra of Section 4 reads the trace as an *unordered* multiset—which is exactly why `distinct` may ignore order and why two facts in mutually exclusive branches are harmless. But some contracts are irreducibly about *post-state* or *order*. Consider a home-automation agent that unlocks the front door, fetches a delivery, and locks up again, under the contract “*no door is left unlocked at the end.*” This is not a query over the bag of calls: `unlock_door`; `lock_door` and `lock_door`; `unlock_door` record the *same* fact set yet leave the door in opposite states, and `Unlock.count() == Lock.count()` is neither necessary nor sufficient (an equal count can still leave the *wrong* door open). *Fluents* add this missing dimension. The name and the idea have a long lineage: time-varying *fluents* and the situation calculus are due to McCarthy and Hayes [7]; Reiter’s *successor-state axioms* [8, 9] are the standard formulation that dissolves the very frame problem Section 6.3 sidesteps for stateless contracts; and the event calculus [10] is a related logic-programming treatment of how fluents change over a trace of events. CLAUZ3 borrows the successor-state-axiom shape in a deliberately small, finite, last-write-wins form rather than adopting the full calculus.

9.1 Fluents and successor-state axioms

A fluent is a named, keyed, mutable cell with a key type, a value type, and an initial value that every key holds before any call runs. A trusted function declares *how* it mutates fluents with an `@effect` axiom whose parameters mirror the function’s:

```
from clauz3.fluent import effect, fluent

DoorLocked = fluent("door_locked", key=str, value=bool, initial=True)

@deal.pre(lambda door: len(door) > 0)
@deal.has("trusted")
@effect(lambda door: DoorLocked.set(door, False))
def unlock_door(door: str) -> None: ...

@deal.has("trusted")
@effect(lambda door: DoorLocked.set(door, True))
def lock_door(door: str) -> None: ...
```

Like `@clauz3.guarantee`, `@effect` is inert at runtime; it merely records, on each fluent, how the call updates it. The decisive point is that the axiom lives in the *trusted* layer, so the trusted layer enforces it. An honor-system precondition such as `end_session(front_locked=True)` would let a manipulated agent simply *assert* the door was re-locked; a successor-state axiom instead *derives* the final state from the calls actually made, which the agent cannot forge. `@effect` is therefore a trusted-layer construct, rejected in agent-authored entry files exactly like `@deal.has` and `@contract`.

9.2 Contracts over the final state

Contracts query a fluent’s valuation *after the whole program runs*, via `.final`:

```
@contract
def all_doors_locked_at_end() -> ContractSpec:
    return DoorLocked.final.all(lambda d: d.value == True)
```

```
@contract
def door_locked_at_end(door: str) -> ContractSpec:
    return DoorLocked.final[door] == True
```

`final.all(p)` quantifies over keys—over the *entire* key sort, not just keys the program touched, so it can hold only when the initial value itself satisfies p (untouched keys retain it; for `DoorLocked` with `initial=True` that is exactly the intended reading: doors never touched remain locked). `final[k]` compares one key’s final value against a literal. These are *state* contracts, not *event* contracts: they permit unlocking mid-errand provided a later call re-locks, which the event-level `never_unlock_door/no_unlocks` cannot express—and vice versa. The two coexist because they answer different questions: *did this ever happen?* versus *how is the house left?*

9.3 Encoding: a guarded-store fold

Each fluent is a Z3 array over its key sort, folded through the recorded fact trace *in program order*:

$$a_0 = K(\text{initial}), \quad a_i = \text{ite}(c_i, \text{Store}(a_{i-1}, k_i, v_i), a_{i-1}),$$

so only reachable calls mutate the array and the last write along any path wins. Then `final[k]` is `Select(a|F|, k)` and `final.all(p)` is $\forall k. p(\text{Select}(a_{|F|}, k))$. This is the same array-threading the program subset already uses for local-variable assignment (Section 7), lifted onto the trusted layer’s effect declarations; it runs at contract-solve time over the fact trace that symbolic execution has already flattened into one guarded, program-ordered layer, so the executor is unchanged.

9.4 Semantics and soundness

Fluents are the one place the model leaves the unordered multiset of Section 6. The concrete object must become a *sequence*: let $\vec{M}_\sigma = \langle (n_1, v_1), \dots, (n_m, v_m) \rangle$ be the trusted calls in execution order under valuation σ , and define the fluent’s final valuation by the obvious fold—start from *initial* and apply each call’s successor-state assignments in turn. A final-state contract holds iff that fold satisfies the predicate. The representation lemma (Lemma 2) extends because symbolic execution emits facts in program order and merges branches into a single guarded layer: facts from the two arms of a branch may be interleaved arbitrarily in that layer, but their guards are mutually exclusive, so restricted to any one concrete path the flattened order agrees with execution order, and the σ -image of the guarded-store fold is exactly the concrete fold; Theorem 1 then carries over to final-state contracts under (A1)–(A3) *plus* one side condition: **(A4)** no fluent-mutating effect occurs under a loop quantifier. A quantified fact stands for N calls with no per-iteration order, which is incompatible with last-write-wins, so such programs fail closed with an `UnsupportedError` rather than guess an order.

9.5 Worked example

The canonical fluent case is the door errand (`examples/home-automation`). The passing program unlocks the front door, orders a delivery, and re-locks; `all_doors_locked_at_end()` holds even though an unlock *event* occurred. The instructive failing program is:

```
@clauz3.guarantee(home.all_doors_locked_at_end())
def main() -> None:
    unlock_door("back")
    lock_door("back")
    unlock_door("front")    # never re-locked
```

The fold leaves `DoorLocked["front"] = False`, so the contract fails—even though a balanced unlock/lock *count* would not flag it, and even an equal count could leave the wrong door open. Only the final-state fluent rules it out.

Fluents are a deliberately small step into stateful reasoning, not a full trace logic: they recover *post-state*, but order policies of the form “a safety check must precede a transfer” still need the security automata of Section 14, which remain future work.

10 From Proof to Consent

A proved contract is only useful if it reaches the user as a decision and is honored at runtime. CLAUZ3 provides four mechanisms.

10.1 Coverage policies: making silence loud

A program can import several trusted domains and attach a strong contract to one while saying *nothing* about another. To a user skimming a dialog, an absent guarantee and an explicit `no_guarantees()` look identical. A coverage policy lets the TLE state, per domain, which contracts *matter*:

```
# tools/email/trusted/policy.py
from clauz3.policy import domain_policy
POLICY = domain_policy(
    when_used="send_email",
    recommended=["only", "unique_recipients"],
    required=[],
    label="email",
)
```

`recommended` is pure meta-reasoning over the *set* of guarantees: if a recommended contract is unstated the approval request is flagged (`recommended_gap`), but the proof is never blocked. `required` is assume/guarantee: a required (nullary) invariant is *conjoined into the proof* whenever its triggering effect is reachable, by rewriting only the proved copy of the source to inject a `@clauz3.guarantee` decorator—so a TLE obligation travels the same path as an agent guarantee, and a violation rejects the program before approval. The coverage report (`covered`, `recommended_gap`, `silent_gap`, `required_gap`) is attached to the approval request so a reviewer sees a flag at a glance.

10.2 The approval service and entailment-based policy

`clauz3 run` proves the program, and *only if all obligations are OK* submits a JSON approval request to an externally configured service. The request carries the proved guarantees, the coverage report, and the exact program; the service presents them, *mediates* the decision, and tells `clauz3 run` to proceed once the user consents. The service runs outside the agent’s control, and the agent cannot get around it: with no channel to the world except `clauz3 run` (Figure 1), waiting on the service is unavoidable. Its trust comes from that mediation and the service’s independence, not from persisting a log of past decisions.

A *policy admin*—the user in a configuration role—may supply rules that auto-decide requests by *entailment* rather than by matching contract text. A rule’s clauses are conjoined as extra guarantees and the target is re-proved; the rule fires only if the program logically entails them. The two decisions are duals:

- `auto_approved` fires when the program entails *all* `when_proven` clauses. (An allow-list alone is unsafe—one address emailed ten thousand times—so the admin makes the conjunction complete by also requiring the bounding count/uniqueness contracts.)

Table 3: Worked examples and the proof shapes they exercise.

Example	Headline contracts
email	only, none, unique_recipients, recipient_at_most, content_length_at_most, same_content
bank	max_spend, only_account
email-from-db	addresses_from, only_table, only_where
text	length_between, must_not_contain, no_regex_metacharacters, only_edit_under
home-automation	all_doors_locked_at_end, never_unlock_door, grocery_budget

- `auto_rejected` fires when the program does *not* entail every `unless_proven` clause: “reject unless you have proven you avoid this,” e.g. `recipient_at_most("ceo@example.com", 0)`.

Deny wins: every reject rule is evaluated before any approve rule, and a clause that cannot be evaluated falls through to a human rather than being decided on incomplete information. Because entailment is logical implication, a rule holds even if the program never mentions that contract by name—a strictly stronger notion than the mention-level coverage check.

10.3 The runtime backstop: same obligations, two times

Static proof is the default, but not the only way to run a program. `CLAUZ3` layers on `deal` precisely so that the obligations the prover discharges *ahead* of execution are the obligations `deal` enforces *at* execution. The `send_email` precondition `"@"` in `addr` is discharged statically for a proved program; if a program is run in a weaker, runtime-only mode, the same precondition fires at the call:

```
>>> send_email("not-an-email", "hi")
deal.PreContractError: addr must contain "@" (where addr='not-an-email', ...)
```

This is what makes “falling back to runtime” a principled, user-visible choice rather than a hole: if the agent declares `no_guarantees()` for a domain and the user accepts that, the system records that weaker mode, and `deal`’s preconditions, postconditions, and effect markers are still on guard at the trusted boundary. The one piece genuinely not yet implemented is binding the approval *receipt* into that boundary so a trusted effect can refuse to run without a valid receipt.

10.4 The runner

`clauz3 run` executes the proved-and-approved program with a reduced builtin set: a deny-list of dangerous imports (`os`, `subprocess`, `socket`, ...) and builtins (`eval`, `exec`, `open`, `getattr`, ...) and dunder-attribute access, with the receipt exposed as `CLAUZ3_APPROVAL_RECEIPT`. This is a source-level guard, *not* a hardened sandbox; the agent harness must still restrict execution to the `clauz3` command path.

11 Worked Examples

The repository ships several end-to-end examples, each with passing and failing cases. We summarize a representative subset and the proof shapes they exercise (Table 3); further domains (a LIMS lab layer, an HTTP layer) reuse the same primitives.

Allow-list (email). The program of Section 4 proves `only(["bob@example.com"])` and `unique_recipients()`. A variant that emails `alice@example.com` fails the first; a variant that emails Bob twice on the same path fails the second (but two Bob emails in mutually exclusive branches *pass*, because both facts cannot be reachable at once).

Budget over a loop (bank).

```
@clauz3.guarantee(bank.max_spend(100))
def main() -> None:
    for _ in range(5):
        withdraw("checking", 10)
```

The loop unrolls to five concrete `withdraw` facts; `max_spend(100)` denotes $\sum_i \text{ite}(c_i, 10, 0) \leq 100$, which Z3 discharges. Raising the per-iteration amount or the iteration count past the bound produces a counterexample.

Column binding (email-from-db). Iterating over `db_query("users", where={"consented": True})` and emailing `row.email` proves `addresses_from(UserRow, "email")` by symbolic iteration. Emailing `row.name` instead fails the structural column check; mixing a literal address into the loop fails because that fact’s `addr` term does not have the query-column shape.

The exfiltration scenario. A natural adversarial case (after Guardians) is an agent steered by a malicious inbox message into forwarding data to `it@othercorp.com`. Under `emails_only([...internal...])` such a program *cannot be proved*: the destination is outside the allow-list, and the allow-list bounds the entire effect set. This catches the *destination* of the exfiltration with no new machinery. Catching the *source→sink* version (an allowed destination but a body carrying data read from the inbox) is beyond today’s relational model and motivates the provenance work in Section 16.

The frame-problem “delete *.txt” scenario. Meijer’s running illustration of why Guardians needs a *frame condition* is an agent that is asked to delete `foo.txt` and `bar.txt` and instead issues `delete_file("*.txt")`: the naive postcondition (*foo.txt and bar.txt no longer exist*) is trivially satisfied by the over-broad call, so Guardians’ verifier must conjoin an explicit frame condition (“files not matching the intended set are unchanged”) before the over-broad plan fails to verify. In CLAUZ3 the contract sits one level lower—over the trace of calls, not the state of the filesystem—and no frame condition is needed:

```
@clauz3.guarantee(fs.delete_only(["foo.txt", "bar.txt"]))
def main() -> None:
    delete_file("*.txt")    # FAILS: "*.txt" not in the allow-list
```

The trusted call records a single fact with `path = "*.txt"`; the allow-list contract `Delete.all(lambda d: d.path in ["foo.txt", "bar.txt"])` denotes a universal over the entire effect set, and the literal `"*.txt"` is provably not in it. The fluent variant works just as well and is instructive: declare `FileExists` as a fluent with successor-state axiom `@effect(lambda path: FileExists.set(path, False))` on `delete_file`, then state `FileExists.final["foo.txt"] == False`. The over-broad call updates only `FileExists["*.txt"]`, so the proof obligation on `"foo.txt"` fails—there is no implicit deduction that the glob *also* hit that key. Closed-world semantics extends from effects to state: only the successor-state axioms the trusted layer literally declares count, and the classical frame condition is unnecessary in both formulations.

12 Application Domains

The worked examples are deliberately small, but the pattern they instantiate is not. CLAUZ3 applies wherever an agent’s side effects matter to a user and the permission worth granting is finer than a capability grant. The only thing a new domain needs is a trusted layer that exposes its effects as `@deal.has` functions and a contract vocabulary over them (Section 4); the prover, the approval flow, and the runtime are domain-agnostic. We sketch the space below. The email, bank, database, text, filesystem, and home-automation layers exist today; the rest is where the approach is headed, and we flag the gaps honestly.

Software-engineering agents. This is the use case the coding agents of Section 1 were built for, and the one where a contract surface most directly buys *differentiated* policy: free rein over some effects, a strict contract on others. Editing source is usually fine—but “touch no file under `tests/`” is a one-line path-prefix contract (the filesystem layer’s `never_write_under("tests/")`, via the same `startswith` primitive as `only_write_under`), and it closes the familiar failure mode where an agent makes a red test pass by quietly rewriting the test. The effects with real blast radius—database mutations, `git` history rewrites, GitHub operations (force-push, branch or issue deletion), package publishes, deployments—are exactly the ones to gate behind a proved contract: “reads only the reporting schema,” “writes at most one row and never DROPs a table,” “pushes only to a feature branch, never `main`.” The cautionary tale is by now a genre—an agent given broad database access wipes a production database, during a change freeze, against explicit instructions [20]. A contract bounding the database effect set (read-only, or writes confined to a development schema) makes that particular catastrophe *unprovable*, so `clauz3 run` refuses the program before its first statement executes, instead of a runtime monitor catching the `DROP` only after earlier statements have already committed.

Personal assistants. The canonical agentic-assistant tasks—email, calendar, messaging, payments, shopping—are exactly the quantitative/allowlist contracts the email and bank layers prototype: recipient allowlists, “no one contacted twice,” spend caps, “do not share this thread outside the named group.” The assistant proposes a program and a contract; the user consents to “emails only the three people I named, at most once each, and spends nothing.”

Home and building automation. Locks, thermostats, and appliances are a natural fit for the stateful fluents of Section 9: “every door is locked at the end,” bounded thermostat setpoints, device allowlists. The home-automation layer is exactly this in miniature. Transactionality matters here in a tangible way—you do not want a program that unlocks a door and then fails before re-locking it—which is precisely the all-or-nothing property static proof provides.

Robotics and embodied agents. This is the sharpest fit, because the generate-then-execute loop is becoming physical. Frameworks such as OpenClaw [19] already have an LLM author and run Python that drives manipulators and humanoid robots (e.g. Unitree G1, or ROS2 arms through a bridge)—the same untrusted-code-with-side-effects shape CLAUZ3 targets, now ending in motors rather than an SMTP socket. A static contract over the generated control program—a workspace or geofence bound (“the end-effector never leaves this region”), a force or speed cap, an object allowlist, or a fluent like “the gripper is released and the arm is home at the end”—is checkable *before the robot moves*, and the all-or-nothing property avoids a half-completed motion that leaves the workcell in an unsafe state. The honest caveat is that genuinely physical contracts—continuous force, geometry, trajectories—largely exceed today’s discrete relational and fluent language; but the architecture carries over directly, with the trusted actuator drivers playing the role of the trusted effect layer.

Table 4: ClauZ3 in the landscape of agent-permission and verification systems. Two cells (bold) are where ClauZ3 is alone in the field.

System	Verified when	Policy by	Consent artifact	Substrate
Capability grants	—	admin per tool	a capability list	—
Per-call approval prompts	runtime	—	each action	source code
Runtime monitor (FORGE)	runtime	operator (Datalog)	each action	execution context
Workflow verification (Guardians)	before data	operator	workflow AST	world state
Proof-carrying code (classical)	before exec	consumer	safety policy	code-level invariants
CLAUZ3	before exec	agent abduces; user accepts	proved contract	closed-world effect trace

Self-driving and autonomous laboratories. An autonomous experiment is a protocol of instrument actions—pipetting, heating, moving plates—and the safety questions are contractual: reagent and cost budgets, instrument and plate allowlists, “never combine these two reagents,” bounded run times. The LIMS example is a toy of this. Autonomous labs [21, 22] are an especially compelling target: protocols are transactional and physically irreversible (you cannot un-mix a reaction), the cost and safety stakes reward verifying before executing, and the trusted layer maps cleanly onto existing instrument drivers. Some of the most useful contracts here are about *order* (“a safety check precedes any heating step”), which today’s fluents only partially reach (Section 9).

Across all four, the same three properties recur: the meaningful permission is quantitative or stateful and domain-specific; the effects bottom out in a small audited driver layer authored once per domain—by a trusted-layer engineer, an instrument vendor, or a robot integrator—and reused across programs; and all-or-nothing consent matters because physical and financial effects are not cleanly reversible. The gaps that recur—continuous and geometric contracts, provenance, and effect ordering—are collected in Section 16.

13 Implementation and Status

CLAUZ3 is implemented in Python (≥ 3.11) and depends on Z3 [3], `deal` [25], and `astroid`; the approval service uses FastAPI. The prover vendors and extends `deal-solver` [26]. The CLI (`clauz3`) exposes `prove`, `run`, `tools`, `install` (copy a trusted layer, optionally emitting agent skill stubs), `config` (write read-only-plus-clauz3 agent permissions), `approval-service`, `policy-check`, and `mock-approval-service`. Two “real” standard-library tools ship under `stdlib/`: a `filesystemlayer` (`read_only`, `only_write_under`, `never_read_under`, `writes_at_most`) and a `grep` layer that imports it. Implemented contract primitives are `no_guarantees`, `all`, `empty`, `where`, `count`, `distinct`, `sum`, and `shares_value`, plus column-binding via `Row` schemas.

14 Related Work

Table 4 places CLAUZ3 in the broader landscape. Two cells stand out: CLAUZ3 is the only system in this list where the *agent* authors the policy and the user grants it, and the only one whose reasoning substrate is a closed-world effect *trace* rather than world state or a flat policy. The rest of this section unpacks the comparison with the closest agent-verification neighbours.

Runtime policy monitors and FORGE. Reference-monitor architectures intercept policy-relevant actions during execution. FORGE [24] is a recent instance for agentic systems: policies are Datalog rules over abstract predicates describing execution context, a reference monitor decides each policy-relevant action, and an observability service maintains causal/provenance facts across tool calls; enforcement is governed by an *environment contract* stating what facts

the environment promises to publish. CLAUZ3 shares the starting premise—prompt instructions are not enforcement, and the agent/world boundary should be explicit—and even borrows the “environment contract” framing for its trusted layer. The mechanisms are complementary rather than competing: a static contract proof gives a concise permission artifact and avoids partial side effects; a runtime monitor gives defense in depth when execution drifts from the proved model. The strongest eventual system likely does both—prove statically, then enforce a matching receipt at runtime.

Guardians. Guardians [23] proposes a generate→verify→execute discipline: the model emits a complete structured *workflow AST* up front, whose arguments are *symbolic references* to prior results rather than concrete (possibly attacker-controlled) values; the plan is verified against a security policy and only then executed. Because the code is fixed before the data arrives, malicious content in tool results or even tool descriptions cannot introduce a new side-effecting step. Verification draws on source/sink taint analysis, security automata [11] over the tool-call sequence, and Z3/Dafny over pre/post/*frame* conditions—with explicit attention to the frame problem [7]. The overlap with CLAUZ3 is substantial: both verify before any side effect runs, both make the effect boundary explicit, both compile to Z3, and both replace natural-language promises with a checkable artifact (and both cite eliminating rollbacks—the transactional argument—as a reason to verify first).

The differences sharpen each system’s identity. Guardians verifies a bespoke workflow AST against an *operator-defined* policy; CLAUZ3 proves an *ordinary Python program* against an *agent-abduced* contract and treats that contract as the consent artifact. Guardians is ahead on three axes CLAUZ3 lacks: a first-class symbolic workflow representation (and the clean code/-data separation that gives the prompt-injection story), built-in taint/provenance, and security automata over tool-call *sequences* (CLAUZ3’s facts are an unordered set carrying only path conditions—though fluents (Section 9) recover post-state via successor-state axioms, they stop short of automata over arbitrary call *orderings*). One difference cuts the other way: because CLAUZ3 reasons over a closed-world finite trace rather than world state, it gets the frame guarantee implicitly (Section 6.3) where Guardians must write explicit frame conditions. The two look more like complementary halves—a workflow front-end lowering into the same effect layer, taint in the fact layer, automata over effect traces—than alternatives.

Proof-carrying code and operator vs. abduced policy. CLAUZ3 inherits the PCC architecture [1, 2] (untrusted producer, trusted checker, policy as the unit of consent) but, like the agentic setting demands, has the agent *propose* the policy rather than fixing it on the consumer side. Classic PCC ships and cheaply re-checks a proof; CLAUZ3 currently re-proves with Z3. Having the agent carry a checkable proof is a natural future direction.

15 Limitations

We are deliberate about what the prototype does *not* do.

- **Trusted-layer trust.** A wrong or lying trusted stub makes the proof wrong. The design moves trust to a small audited layer; it does not abolish trust.
- **Contract-helper bodies are executed.** `@contract` builder functions run as Python; only their *lambda* bodies are AST-validated. Fully static validation of helper bodies is open.
- **A small language.** The supported Python subset and the relation algebra are intentionally narrow. Unbounded loops, rich string/regex predicates, grouped aggregates, index-dependent `sum` selectors, and `shares_value` over loop-framed facts are unsupported and

fail closed (`count` and index-independent `sum` over framed facts *are* supported by length-multiplication, Section 7.4). Two side conditions of Lemma 1 are *assumed rather than detected*: cross-iteration accumulators and exception-raising loop bodies are not yet rejected by the v1 checker, so a program violating them is analyzed with a wrong (conservatively wrong, in the cases we have constructed) symbolic value instead of an `UnsupportedError`. Closing this is implementation work, not a design change.

- **Prover gaps against the stated semantics.** The vendored prover currently assumes boundary postconditions *unguarded* rather than as $c_f \Rightarrow post_f$ (Section 7), and drops postconditions assumed inside branches instead of merging them gated on the branch test. The first is the vacuity hazard noted in Section 7; the second is an incompleteness (spurious FAILs), consistent with fail-closed but worth fixing.
- **Limited provenance and ordering.** Facts carry path conditions but no source labels, so source→sink taint cannot yet be expressed. Order enters only through fluents’ final state (Section 9); arbitrary “X before Y” automata cannot. Fluents themselves are v1-limited: no effects inside loops (incompatible with last-write-wins, fail-closed), only literal or parameter keys/values, and key/value types in `{str,int,bool}`.
- **Not a sandbox; receipts not bound.** The runner is a source-level guard, not OS isolation, and runtime receipt enforcement at the trusted boundary is not implemented.
- **Not a prompt-injection defense.** CLAUZ3 reasons about what a proved program can do, not about how the agent was induced to write it.

16 Future Work

The effect-fact layer is the natural extension point. Promising directions: (1) *taint/provenance* in `FactInfo.args`, generalizing the existing column-binding anchor into a source label that propagates through assignments, to express “a value from `fetch_mail` must not reach `send_email.body`”; (2) *security automata* over an ordered effect trace, for sequence policies such as “a check must precede a transfer”; (3) a *Datalog/SQL or OWL/DL frontend* that compiles to the same relation primitives and supports static contradiction, redundancy, and subsumption checks over policies, plus deterministic natural-language rendering for the approval dialog; (4) a *workflow front-end* that lowers a Guardians-style plan into the effect IR so both can reuse one prover and executor; (5) *agent-carried proofs* in the PCC sense, cheaply re-checked rather than re-proved; and (6) an *adversarial benchmark*, beginning with the email-exfiltration scenario.

17 Conclusion

CLAUZ3 reframes the agent-permission question. Instead of granting a coarse capability or refereeing each action at runtime, it asks the agent to write a short program and *abduce* a contract over its trusted side effects, proves that contract statically before anything runs, and makes the proved contract—not the code—the thing the user reads and consents to. The technical core is modest and, we think, clarifying: trusted calls become a closed-world finite trace of effect facts, a small relational algebra over that trace compiles to Z3, and the closed-world reading gives a frame guarantee without explicit frame conditions. The result is binary, transactional consent over the whole program, a permission artifact small enough to actually read, and a clean *assume/guarantee* split between an audited trusted layer and an untrusted agent. Much remains—provenance, ordering, a richer frontend, receipt enforcement—but the unit of consent being a *proved contract* rather than a capability or a per-call veto is, we argue, the piece worth keeping.

References

- [1] G. C. Necula. Proof-Carrying Code. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 106–119, 1997.
- [2] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 229–243, 1996.
- [3] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 4963, pp. 337–340, 2008.
- [4] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [5] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [6] C. B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.
- [7] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.
- [8] R. Reiter. The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pp. 359–380. Academic Press, 1991.
- [9] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.
- [10] R. Kowalski and M. Sergot. A Logic-Based Calculus of Events. *New Generation Computing*, 4(1):67–95, 1986.
- [11] F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
- [12] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. Not What You’ve Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *Proc. 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023.
- [13] B. Stanton, M. F. Theofanos, S. S. Prettyman, and S. Furman. Security Fatigue. *IT Professional*, 18(5):26–32, 2016.
- [14] A. Beaument, M. A. Sasse, and M. Wonham. The Compliance Budget: Managing Security Behaviour in Organisations. In *Proc. 2008 New Security Paradigms Workshop (NSPW)*, pp. 47–58, 2008.
- [15] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proc. 8th Symposium on Usable Privacy and Security (SOUPS)*, 2012.
- [16] D. Akhawe and A. P. Felt. Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness. In *Proc. 22nd USENIX Security Symposium*, pp. 257–272, 2013.

- [17] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proc. 11th International Conference on Learning Representations (ICLR)*, 2023.
- [18] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *Proc. 12th International Conference on Learning Representations (ICLR)*, 2024.
- [19] OpenClaw. An Open-Source Agentic Framework for Computer and Robot Control. <https://openclaw.ai/>, accessed 2026.
- [20] AI Incident Database. Incident 1152: LLM-Driven Replit Agent Reportedly Executed Unauthorized Destructive Commands During a Code Freeze, Leading to Loss of Production Data. <https://incidentdatabase.ai/cite/1152/>, 2025.
- [21] R. Ferreira Da Silva, R. Moore II, B. Mintz, et al. Shaping the Future of Self-Driving Autonomous Laboratories Workshop. Technical Report ORNL/TM-2024/3714, Oak Ridge National Laboratory, 2024. doi:10.2172/2481197.
- [22] L. M. Helleckes, S. Putz, K. Gupta, M. Franzreb, and H. Garcia Martin. Perspectives for Artificial Intelligence in Bioprocess Automation. *Current Opinion in Biotechnology*, 97:103392, 2026. doi:10.1016/j.copbio.2025.103392.
- [23] E. Meijer. Guardians of the Agents: Formal Verification of AI Workflows. *ACM Queue*, 23(4), 2025; *Communications of the ACM*, January 2026. doi:10.1145/3777544.
- [24] M. Palumbo, A. Choudhary, S. Choi, G. Amir, S. Chalasani, and S. Jha. Formal Policy Enforcement for Real-World Agentic Systems (FORGE). *arXiv preprint* arXiv:2602.16708, 2026.
- [25] Orsinium Labs. *deal: Design-by-Contract for Python*. <https://deal.readthedocs.io/>.
- [26] Orsinium Labs. *deal-solver: A Z3-backed formal verifier for deal contracts*. <https://github.com/life4/deal-solver>.
- [27] CLAUZ3 contributors. *CLAUZ3: Static Contracts for Agent-Authored Python*. <https://github.com/normalform-ai/clauz3>.