The Anatomy of a Safe Autonomous Claude Code Agent

>_ a reference architecture for agents that work while you sleep

PrimeLine · companion to the /autonom + /desk series · reconstructed reference implementation

Why most "autonomous agents" are confidently wrong

An agent that writes code, writes its own test, and judges its own test will convince itself that broken work is done. At interactive speed you catch it. Overnight, unattended, across a queue of tasks, you don't. The agent ships green garbage and you find out the next morning.

Autonomy is not the hard part. A `while` loop is autonomy. The hard part is the layer that makes unattended action *safe to trust*: a system that can act on everything reversible by itself, prove each action against real state before it counts as done, and surface only the irreducibly human decisions to you.

This document is the full architecture of that layer, in nine parts, with working reference code. Two slash commands sit on top of it:

- `/autonom` - the engine. Drains the work queue, acts on everything safely automatable, loops until dry.
- `/desk` - the dashboard. Surfaces only the decisions that genuinely need a human, ranked.

They are two halves of one rule: *act on everything safe, escalate only what isn't.*

The nine layers

#	Layer	What it does	What it prevents
1	Human trigger	Activates only when you type the word. No cron, no self-spawn	A runaway loop you never started
2	Loop contract	Decide → execute → loop until dry. Never ends with a question	"Drained one item, now what?" fake-autonomy
3	Single-session lease	One run at a time, auto-expires after 4h	Two sessions clobbering each other
4	Quota governor	Refuses below 20% headroom, throttles below 35%	Burning your interactive budget
5	Reversible-only + worktree	Every change git-revertible, lands in an isolated worktree	An unrecoverable mistake
6	Forced verification gate	3-leg proof (it fired, it changed real state, a consumer can use it) before any "done"	Self-certified broken work
7	Safety spine outside self-modification	The agent cannot edit its own verifier, lease, or governor	A system that heals its own brakes
8	Verify-before-change	Consult the dependency map first; "zero references" lies	A deletion that looked safe
9	Decision desk	Ranks and surfaces only human-required decisions	Important calls rotting in a log

Each layer below: what it is, the exact logic, and reference code you can run.

Layer 1 - The human trigger

The agent activates only when you type a single trigger word in a fresh session. It never spawns itself, never runs on a cron, never wakes on a timer. A SessionStart notice can *inform* you that work is waiting, but informing is not spawning.

This is the cheapest safety property and the most important: the blast radius of any bug is bounded by the fact that nothing runs unless you started it.

```
TRIGGER = "autonom"  # one word, case-insensitive, typed by a human

def should_activate(user_message: str) → bool:
    return user_message.strip().lower() == TRIGGER
```

The notice that *informs* (separate from activation) just reads the producer signal (Layer 4 logic) and prints a one-liner. It has no power to act.

Layer 2 – The loop contract

A run that drains one batch and then asks "want me to commit, or build X?" is not autonomous. Autonomous means: decide, execute, and keep going until there is genuinely nothing safe left to do.

The contract has four rules:

1. **Never end with a question or an option menu.** You have full authority for reversible work. The only things that surface are the hard stops in rule 4, and even those are a flagged statement, not a menu.
2. **Decide-and-execute defaults (do not ask):**
   - Reversible drained work → commit it to the session branch after each batch.
   - Excluded / unsafe item → skip it, log it, move on.
   - Item needing a design decision, over ~30 min, or destructive-without-backup → defer it with a reason code, log it, keep draining.
3. **Loop until dry.** After each batch, re-read the producer signal. If it still fires, drain again. Stop only when the queue is empty, the governor refuses, the verifier hits a kill-criterion, or every remaining item is excluded or deferred.
4. **Closeout is a report, never a question.** State what was committed, what was deferred (with reason codes), what was skipped.

```
def autonomous_loop(ctx):
    while True:
        signal = compute_signal(ctx.repo_root)      # Layer 4
        if not signal.fires:
            break                                    # queue dry
        gov = check_governor(ctx.repo_root)          # Layer 4
        if gov.decision == "REFUSE":
            break
        for item in drain_batch(signal):
            decision = decide(item)                  # act / skip / defer
            if decision == "act":
                apply_in_worktree(item)              # Layer 5
                if stop_gate(item).passed:           # Layer 6
                    commit(item)                     # default, never ask
                else:
                    discard_worktree(item); log_failure(item)
            elif decision == "defer":
                log_deferred(item, reason_code)
            else:
                log_skipped(item)
        # loop back, re-read the producer
    return write_report()                            # never a question
```

Layer 3 - The single-session lease

Two autonomous sessions running at once will read-modify-write the same state files and silently corrupt each other. A file-backed lease makes the run mutually exclusive: a second activation in another session is refused.

Exact semantics:

- **TTL = 4 hours.** A lease older than that is treated as stale and can be reclaimed (the original session likely died).
- **Idempotent re-claim:** the same session re-claiming its own live lease is a no-op, not a refusal.
- **Locked read-modify-write** via `flock`, with a bounded retry loop and a graceful fallback on filesystems that do not support locking.

```
import fcntl, json, os, time
from dataclasses import dataclass
from pathlib import Path

LEASE_TTL_SECONDS = 4 * 3600
_LOCK_TIMEOUT_S = 5.0
_POLL_S = 0.02

@dataclass
class LeaseState:
    session_id: str
    claimed_at: float
    released: bool
    def is_stale(self, ttl=LEASE_TTL_SECONDS) -> bool:
        return (time.time() - self.claimed_at) > ttl

class LeaseRefused(RuntimeError): ...

def claim_lease(session_id: str, lease_path: Path) -> LeaseState:
    fd = os.open(lease_path, os.O_RDWR | os.O_CREAT, 0o644)
    deadline = time.time() + _LOCK_TIMEOUT_S
    while True:
        try:
            fcntl.flock(fd, fcntl.LOCK_EX | fcntl.LOCK_NB)
            break
        except OSError:
            if time.time() > deadline:
                raise TimeoutError("lease lock contended")
            time.sleep(_POLL_S)
    try:
        raw = os.read(fd, 65536).decode() or "{}"
        cur = json.loads(raw) if raw.strip() else {}
        existing = LeaseState(cur.get("session_id", ""), cur.get("claimed_at", 0.0),
                              cur.get("released", True))
        if existing.session_id == session_id and not existing.released:
            return existing                                  # idempotent
        if existing.session_id and not existing.released and not existing.is_stale():
            raise LeaseRefused(f"held by {existing.session_id}")
        new = LeaseState(session_id, time.time(), False)
        os.lseek(fd, 0, 0); os.ftruncate(fd, 0)
        os.write(fd, json.dumps(new.__dict__).encode()); os.fsync(fd)
        return new
    finally:
        fcntl.flock(fd, fcntl.LOCK_UN); os.close(fd)
```

> Note: never write JSON state with an inode-swapping `mv` / `os.replace` from a concurrent writer. Use a locked in-place read-modify-write, or a second writer can clobber the first's append. (The blog series covers this lost-append race in its own post.)

Layer 4 – The quota governor

Before each run, and again between work items, the agent checks how much of the model budget is left and decides whether it is allowed to proceed. This protects your interactive budget: the agent yields the moment headroom gets tight.

Three thresholds, two of them tunable:

Headroom	Decision
below 20%	**REFUSE** – do not run
20% to 35%	**THROTTLE** – run, but yield between items
above 35%	**GO**

The boundary at exactly 20% falls to THROTTLE (strict less-than for REFUSE). When no budget data is available, default to a conservative 60% headroom – safe to run, not blindly optimistic.

```
from enum import Enum
from dataclasses import dataclass

class GovernorDecision(str, Enum):
    GO = "GO"; THROTTLE = "THROTTLE"; REFUSE = "REFUSE"

HARD_RESERVE_PCT = 20.0
SOFT_THROTTLE_PCT = 35.0
CONSERVATIVE_DEFAULT_HEADROOM_PCT = 60.0

@dataclass
class GovernorResult:
    decision: GovernorDecision
    headroom_pct: float
    reason: str
    @property
    def allows_run(self) → bool:
        return self.decision ≠ GovernorDecision.REFUSE

def make_governor_decision(headroom_pct: float,
                           hard=HARD_RESERVE_PCT, soft=SOFT_THROTTLE_PCT) → GovernorResult:
    if headroom_pct < hard:
        d = GovernorDecision.REFUSE
    elif headroom_pct < soft:
        d = GovernorDecision.THROTTLE
    else:
        d = GovernorDecision.GO
    return GovernorResult(d, headroom_pct, f"headroom {headroom_pct:.1f}% → {d.value}")
```

`headroom = 100 - used_percentage`, clamped to [0, 100]. Read the used-percentage from whatever budget telemetry you have; fall back to the conservative default when the source is missing.

Layer 5 - Reversible-only and worktree isolation

The reason you can let this run while you sleep: every action is undoable, and nothing touches your working tree until it has passed verification.

Two hard rules:

- **Reversible-only.** Every change must be revertible via `git revert` or a file restore. No hard deletes (archive instead). No destructive operations without an explicit human sign-off.
- **Worktree isolation.** All writes land in a throwaway git worktree on a session branch. The work merges back only after the verification gate (Layer 6) passes. A failed item discards its worktree and leaves your tree untouched.

```
# one isolated worktree per session
git worktree add /tmp/agent-work-$(date +%s) -b agent/session-$(date +%Y%m%d)
# ... agent applies changes there, runs the stop-gate ...
# pass → merge the branch back; fail → git worktree remove (no trace)
```

The agent also never commits to `main` directly and never runs `git add -A` (it stages explicit paths only), so a stray file can't ride along into a commit.

Layer 6 - The forced verification gate

This is the core. Before any iteration is allowed to claim "done," it must supply a three-leg proof, and the gate blocks the claim if any leg is missing.

The three legs (Empirical Completion Proof):

1. **Trigger** - the thing fired/ran under realistic conditions, with a timestamp.
2. **Effect** - it produced the expected effect on real system state (a test output slice, a file diff, a log line).
3. **Consumer** - a downstream consumer can use that effect (CI badge green, next phase unblocked, a human can act on the doc).

The gate is a cheap, always-on syntactic check. It cannot verify truth - it verifies that the producer *articulated* all three legs. That alone kills the most common failure mode: a confident "done" backed by nothing.

```
from dataclasses import dataclass, field

_MIN_LEG_LENGTH = 10
_TRIGGER_WORDS = frozenset({
    "done", "works", "working", "tested", "shipped", "verified", "ready",
    "complete", "passing", "live", "deployed", "integrated", "fixed",
    "resolved", "ok", "green", "validated", "confirmed", "in place",
    "hooked up", "wired", "registered", "active", "operational",
})

@dataclass
class EPTEvidence:
    trigger: str = ""
    effect: str = ""
    consumer: str = ""
    def legs_present(self) → dict:
        return {k: len(v.strip()) ⩾ _MIN_LEG_LENGTH
                for k, v in {"trigger": self.trigger,
                             "effect": self.effect,
                             "consumer": self.consumer}.items()}

@dataclass
class StopGateResult:
    passed: bool
    block_reason: str = ""
    missing_legs: list = field(default_factory=list)

def check_stop_gate(claim: str, evidence: EPTEvidence | None = None,
                    *, require_trigger_word: bool = True) → StopGateResult:
    triggered = any(w in claim.lower() for w in _TRIGGER_WORDS)
    if not triggered and require_trigger_word:
        return StopGateResult(passed=True)          # not a completion claim, allow
    legs = (evidence or EPTEvidence()).legs_present()
    missing = [leg for leg, ok in legs.items() if not ok]
    if missing:
        return StopGateResult(
            passed=False,
            block_reason=f"BLOCKED: completion claim missing EPT legs {missing}. "
                         "Work is deferred-and-untested until all 3 are provided.",
            missing_legs=missing)
    return StopGateResult(passed=True)
```

Inside the autonomous loop the gate runs with `require_trigger_word=False`, so every single iteration must prove all three legs regardless of how the claim is phrased. There is no bypass via wording.

The adversarial judge (for high-risk changes)

The syntactic gate is tier 0. High-risk changes get a tier-1 check: a separate-model judge prompted to *refute*, not approve. Two invariants make it trustworthy:

- **Separation of duties.** The producer model may not judge its own work. A change produced by one model family must be judged by a different one. (A judge that shares the producer's blind spots is theater.)

- **Conservative bias.** The judge defaults to "not verified" when uncertain, and any verdict with confidence below 0.6 is forced back to not-verified.

```
JUDGE_SYSTEM = (
    "You are an ADVERSARIAL VERIFIER. Your job is to REFUTE the claim. "
    "Default to 'not verified' when uncertain. If real_state is empty, set verified=false. "
    "Never set verified=true with confidence < 0.6. "
    "Give at least one refutation reason even when you verify."
)

def parse_verdict(v: dict) → dict:
    if v.get("verified") and v.get("confidence", 0.0) < 0.6:
        v["verified"] = False
        v.setdefault("reasons", []).insert(0, "confidence < 0.6, overriding to not-verified")
    return v   # any parse failure → verified=False (fail-safe, never fail-open)
```

What counts as high-risk is decided by a router: more than ~5 changed files, or any path touching state stores, hooks, settings, orchestration, security-adjacent keywords (`auth`, `credential`, `secret`, `token`, `password`), or the verification spine itself. You cannot force-downgrade the risk level for a spine path.

Layer 7 - The safety spine lives outside self-modification

A self-improving system that can edit its own hooks, rules, or verifier can "heal" a guardrail or quietly degrade its own brakes. So the system may auto-evolve everything *except* the parts that judge and constrain it. Those stay human-sign-off.

The spine - the verifier code and its tests, the lease, the governor, the producer, the trigger skill itself, and the autonomy classifier - is a protected set. Any autonomous change touching a spine path is forced to INTERACTIVE: the agent must stop and ask you.

```
SPINE_PATTERNS = (
    "lib/verifier/",        # the verifier and its tests
    "agent/lease",          # the lease
    "agent/governor",       # the governor
    "agent/producer",       # the work signal
    "hooks/forced-verify",  # the live verification hook
    "skills/autonom",       # the trigger skill itself
    "autonomy-classifier",  # the rules that decide what is autonomous
)

def is_spine_path(path: str) → bool:
    return any(p in path for p in SPINE_PATTERNS)
```

This is the layer behind the build-in-public story "my agent refused to let me make it more autonomous." When you try to widen the autonomous mode's own permissions, the change lands on a spine path and the system blocks itself until you sign off by hand. The brake lives outside the engine it brakes.

Layer 8 - Verify before you change anything

Before retiring, editing, or moving any component, the agent consults a dependency map first and never blind-changes. "Zero references" is a lie: things connect in more ways than a grep shows.

The six consumer pathways to check before calling anything safe to remove:

1. Graph edges
2. Routing / dispatch config
3. Detection / keyword index
4. Knowledge-store references
5. Plain-text mentions (`@name` , imports, docs)
6. Symlinks

A component is only safe to retire if **all six** return nothing. If the picture is still uncertain after the check, the agent defers rather than acts. Reversible-only still binds: archive, never hard-delete; every change `git revert` -able.

The map itself is a per-subsystem dependency atlas - consumers, lateral connections, reverse blast radius - that the agent reads instead of re-deriving. Treat it as a point-in-time snapshot: re-confirm a flagged gap is still live before acting on it.

Layer 9 - The decision desk

Everything the agent could *not* safely decide on its own lands here, ranked, so the important calls never rot in a log. This is the human-facing half of the system: `/autonom` acts on everything reversible, `/desk` surfaces only what genuinely needs you.

The ranking formula

```
score = severity x age_factor x irreversibility x blast_radius x ack_penalty
```

Factor	Values
severity	P0=4.0, P1=3.0, P2=2.0, P3=1.0 (default 1.0)
age_factor	`1.0 + age_days / 7.0` - climbs one point per un-actioned week
irreversibility	time-bound decisions 2.0, manual review 1.5, routine 1.0
blast_radius	2.0 if title/detail mentions security, data-loss, production, deploy, delete, schema; else 1.0
ack_penalty	0.5 if acknowledged-but-unresolved, else 1.0

Two patterns make this cheap and self-maintaining:

Free escalation without a new writer. A decision's first-seen timestamp is the *minimum* timestamp across all its emissions in the append-only history. So `age_factor` climbs

automatically - an un-actioned item rises in rank every week with no separate tracker. The append-only log *is* the first-seen source.

Current, not ever-seen. Take each producing module's *latest run only*. A finding the module stopped emitting (because it got resolved, or no longer detects) drops off on its own. No pruning job needed.

```
SEVERITY_WEIGHT = {"P0": 4.0, "P1": 3.0, "P2": 2.0, "P3": 1.0}
EPOCH_DAY = 86400.0
BLAST_KEYWORDS = ("security", "credential", "password", "secret", "token",
                  "auth", "data-loss", "data loss", "production", "deploy",
                  "delete", "schema")

def rank(d, now: float) → float:
    sev = SEVERITY_WEIGHT.get(d.severity, 1.0)
    age_days = max(0.0, (now - d.first_seen_ts) / EPOCH_DAY) if d.first_seen_ts else 0.0
    age_factor = 1.0 + age_days / 7.0
    if d.is_time_bound and d.days_overdue > 0:        # only follow-ups / audits
        age_factor += d.days_overdue / 7.0
    blob = (d.title + " " + d.detail).lower()
    blast = 2.0 if any(k in blob for k in BLAST_KEYWORDS) else 1.0
    ack_penalty = 0.5 if d.acknowledged else 1.0
    return sev * age_factor * d.irreversibility * blast * ack_penalty
```

The lifecycle ledger

Acting on a decision is an append-only event - `resolve`, `defer (until a date)`, `drop`, or `ack` - written to a ledger that the desk reads back with *latest-action-wins*. Resolving or dropping an item makes it disappear; deferring hides it until the date; acknowledging keeps it but halves its rank. An `ack` after a `resolve` re-opens the item. The findings stream itself is never mutated.

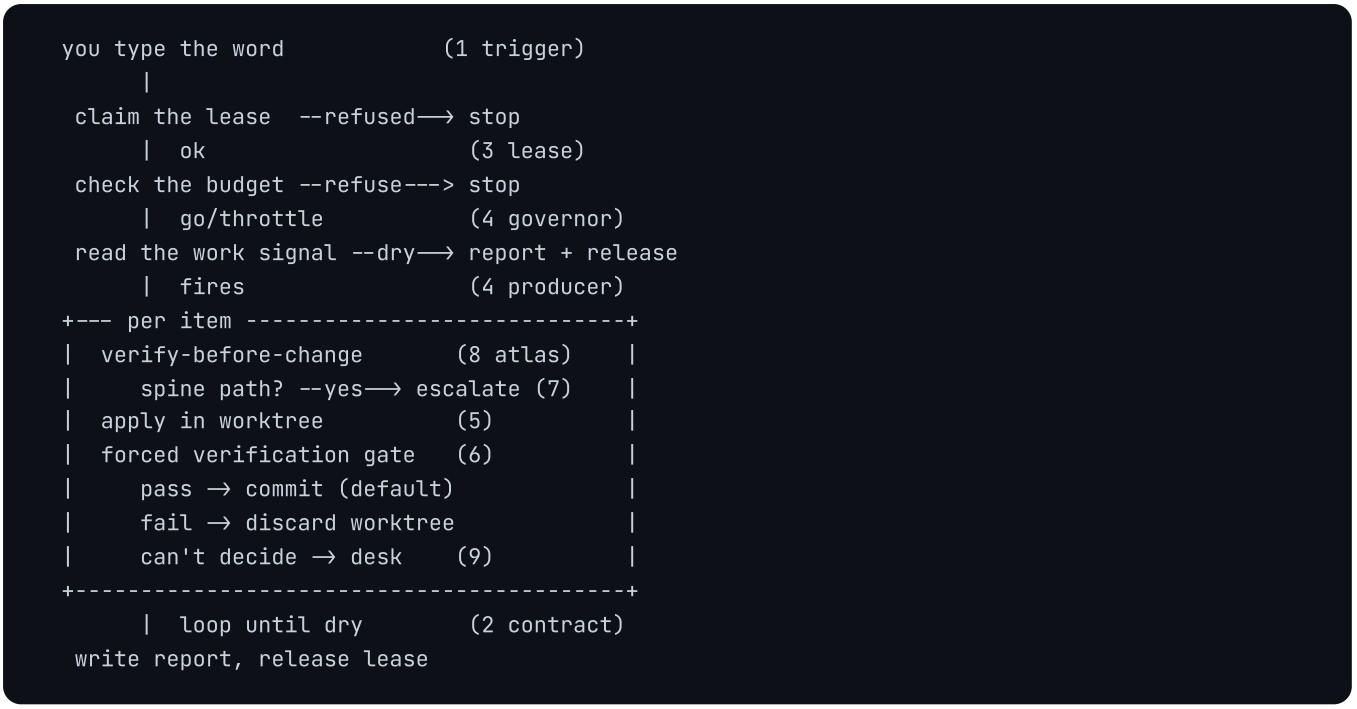
A stable key lets every decision be acted on even when it has no native id: `module:item_id`, or `module:sha1(module|title|source)[:16]` when the id is missing.

```
def is_open(key, ledger_latest: dict, today) → tuple[bool, bool]:
    row = ledger_latest.get(key)
    if not row:                       return (True, False)     # never touched
    action = row["action"]
    if action in ("resolve", "drop"): return (False, False)
    if action == "ack":               return (True, True)      # visible, acknowledged
    if action == "defer":
        until = row.get("until")
        if not until:                 return (False, False)    # deferred indefinitely
        return (today ⩾ until, False)                          # resurfaces on the date
    return (True, False)
```

The badge is silent when the desk is empty - if nothing needs you, you see nothing.

How the layers gate each other

The nine layers are not a checklist, they are a chain of gates. Each one can stop the flow.



The trigger bounds the blast radius. The lease and governor decide *whether* to run. The producer decides *if there is work*. Verify-before-change and the spine guard decide *what is safe to touch*. The worktree makes every attempt *undoable*. The verification gate decides *what counts as done*. And the desk catches *everything the agent could not safely decide alone*.

Build your own: the checklist

- [ ] Activation is a human-typed word. No cron, no self-spawn.
- [ ] The loop decides and executes; it never ends with a question.
- [ ] A file lease makes runs mutually exclusive, with a stale-TTL.
- [ ] A governor refuses below a hard reserve and throttles below a soft one.
- [ ] Every change is reversible and lands in an isolated worktree first.
- [ ] A verification gate demands a 3-leg proof before any "done."
- [ ] High-risk changes get a separate-model adversarial judge, conservative by default.
- [ ] The verifier, lease, and governor are a protected spine the agent cannot self-edit.
- [ ] Nothing is removed without a six-pathway consumer check.
- [ ] Human-required decisions are ranked and surfaced, never left to rot.

Everything here is a reference architecture, not a library to install. The nine layers compose into one system: each gates the next, the verifier sits outside what it judges, and the human sees only what genuinely needs a decision. Build it in whatever stack you run, on top of whatever agent harness you already have.

Read the two-part deep-dive on each half: primeline.cc/blog