Source: b3ubot design.md @ 4fc60ef — the LIVE authoritative design document, converted 2026-08-08. Edits land in the source first.

b3ubot — Adaptive Programming Engine — Design

This is the **live, authoritative design document** for b3ubot. It evolves as the engine's architecture, implementation, and capabilities mature; §5 defines the open decisions and §6 the maintenance rules. The implementation road that realizes this design is `end_to_end.md` (phases → milestones → units of work); nothing there redefines design, only sequences it.

---

1. Vision and methodology

1.1 Product thesis

AI coding assistants reason powerfully and verify weakly. They propose code with fluent confidence, but the confidence is not evidence — every serious user re-derives trust by hand: run the tests, read the diff, try the build. CCS sits at the opposite pole: deterministic, grammar-driven, and equipped with byte-stable round-trip oracles that make correctness a *checkable* property rather than a claimed one.

b3ubot fuses the two. It is an Adaptive Programming Engine in which:

- **AI proposes** — through whichever provider is currently behind the abstraction layer;
- **oracles dispose** — deterministic verification (test suites, live gates, and CCS round-trip validation) decides what is real;
- **the ledger remembers** — every unit of work leaves a skeleton, execution evidence, and a retro pair, so the project's history is a structured, replayable record rather than a chat transcript.

The developer's relationship is with **b3ubot**, not with a model. Providers are interchangeable components (§2.4); swapping or upgrading one must never change the workflow, the ledger format, or the verification bar.

1.2 Methodology inherited (and productized)

b3ubot's development follows the same discipline the b3u sibling used — and its *product* is that discipline, engineered:

- **The step paradigm**: `steps/step.N.txt` skeletons (ground-truth probe → deliverables → verification gates → LOCKs → open questions → acceptance → hash backfill), ratified before execution, closed with a dual retro (`.diff.txt` narrative + `.gdiff.txt` automated, via `scripts/step_gdiff` — corrected 2026-07-20/step.13 from `.cdiff.txt`/`claude_diff`, retired repo-wide for an off-by-one range + self-referential growth bug; this repo's own steps 1-12 have used `step_gdiff`/`.gdiff.txt` exclusively).
- **Ledger truth**: the end_to_end §11 table is the single live progress record, updated in the same commit as the work.
- **Oracle-first**: every UoW names its verification gates before its implementation; suites are promoted to regression forever.
- **The isomorphic lockstep**: the data model (§3) is built through the five-layer pipeline (verbal → SGDL → SQLite → OO API → CLI), each layer 1:1 with its neighbors, closed by a round-trip chain oracle.
- **Honest interims**: partial mechanisms are named at creation (with their redemption plan), never silently deferred.

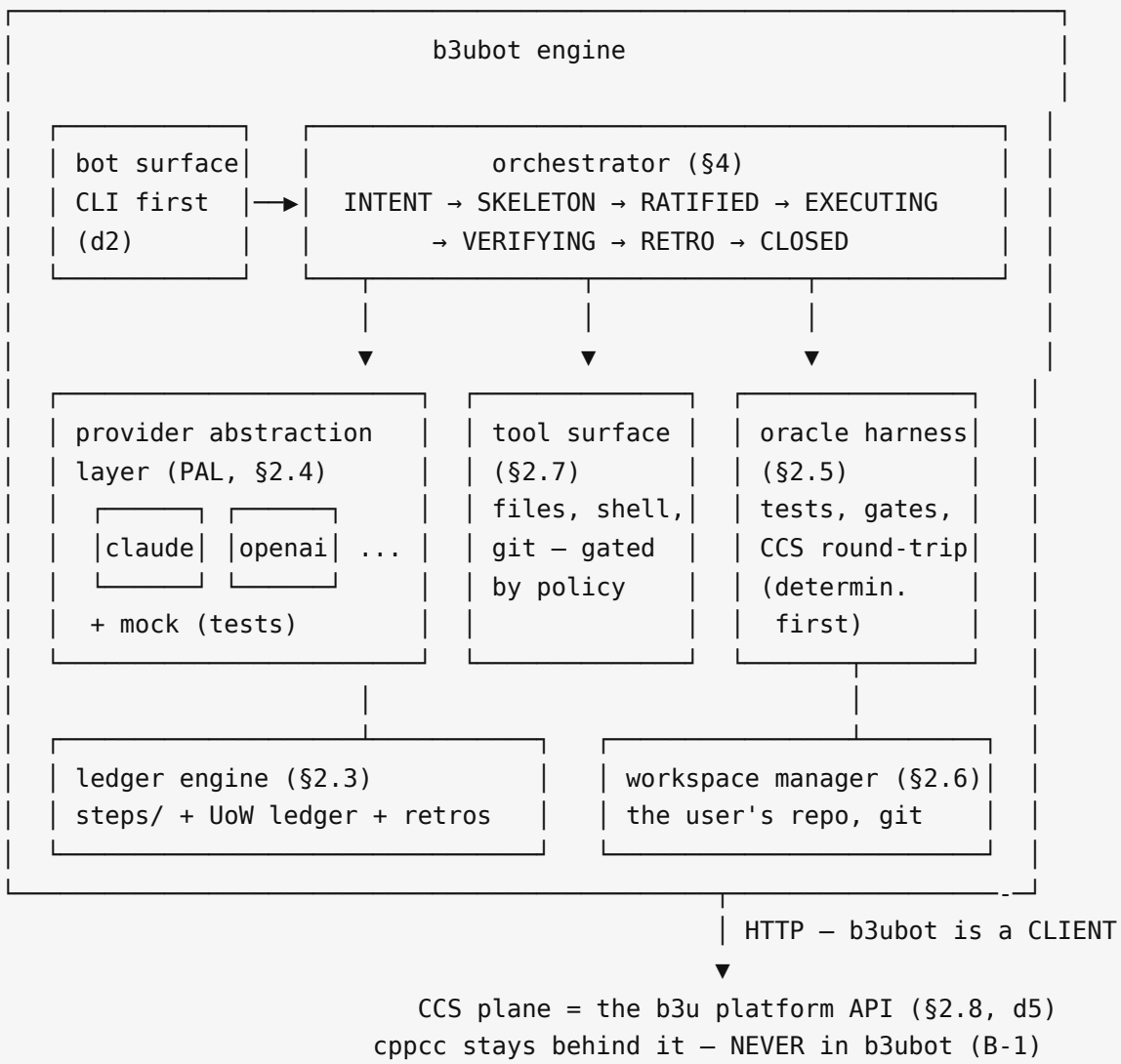
What b3u practiced by hand between a developer and an assistant, b3ubot implements as machinery: the skeleton drafts, the ratification gates, the execution loop, the oracle verdicts, and the retro closure become engine states (§4), not conventions.

1.3 What already exists to build on

- **cppcc** — the CCS compiler-compiler (sibling repo), with `-b` populate-on-target, the `.fgr`/`.bfgr` formats, and per-grammar generated parsers. Private, server-side only (LOCK B-1).
- **libscbcpp.a + the SCB API quartet** (C++/Python/JS/Rust) — pure SCB processors for reading/validating grammar binaries without cppcc itself.
- **The b3u.dev platform patterns** — generation-service shape (submit SGDL, run cppcc server-side, deliver artifacts), unit store with spec-only retention, entitlement gating, local-mode deployment (the PL side-track), the four-layer test architecture. Since PL closed (2026-07-12) this is not just a pattern but a **running service**: the local-mode b3u instance is the CCS plane b3ubot consumes as a client (d5 — §2.8).
- **The ledger practice itself** — 34 b3u steps of skeleton/execute/retro rhythm are the behavioral spec for the ledger engine (§2.3): b3ubot's first job is to be able to run *that* loop.

2. System Architecture

2.1 High-level architecture



2.2 Major subsystems

Subsystem	Role	Design section
Bot surface	The developer's interface (CLI/TUI first — d2)	§2.9

Subsystem	Role	Design section
Orchestrator	The adaptive-loop state machine	§4
Ledger engine	Step files, UoW ledger, retro pairs — parse/emit/round-trip	§2.3
Provider abstraction layer (PAL)	Uniform envelope over interchangeable AI providers	§2.4
Tool surface	Gated file/shell/git operations providers may request	§2.7
Oracle harness	Deterministic verification, tiered	§2.5
Workspace manager	The user's repo under management	§2.6
CCS plane client	The CCSService port over the b3u API (d5: b3ubot is a client)	§2.8
Workflow DAG substrate	On-the-fly DAG instances = the workflow's materialized form	§2.13
Data model	Engine state through the five-layer pipeline	§3

2.3 The ledger engine

The ledger is the engine's memory and its contract with the developer. File-based first (d4): `steps/step.N.txt` with the canonical sections (Status / ground truth / deliverables / gates / LOCKs / open questions / acceptance / hash backfill), the end_to_end §11 UoW table (corrected 2026-07-20/step.13 from "§9" — stale, same root cause as the "34→50" corpus-count and cdiff/gdiff staleness above: section numbers shift as the plan grows), and retro pairs.

The engine must **round-trip its own ledger**: parse any step file it emits, emit any step file it parses, byte-stable. That property is P2's oracle and is what makes the ledger machine-manageable instead of merely machine-written. The 50 b3u step files (`step.1.txt`–`step.50.txt`; corrected 2026-07-20, step.10 — the corpus grew since P2 was first scoped at "34") are the corpus the parser must accept (they are the behavioral spec — §1.3).

Scope, precisely (2026-07-20, after U2.1/step.10 landed): the step-file format is a convention **shared with b3u** — hand-authored by the operator and Claude in both repos — not something b3ubot invented or governs. U2.1 built the READ half only: a parser (`app/ledger/parser.py`) plus a schema doc (`docs/step_file_schema.md`) that is **descriptive**, derived by grepping the

FULL corpus for what is actually structurally invariant (section `(0)` is always titled "Status"; header fields are near-universal but not rigid — e.g. `b3u/steps/step.8.txt` titles its gates section differently, and the parser had to tolerate that, not reject it). "The canonical sections" means "the sections that are structurally invariant across real usage," not "the format b3ubot now controls." Nothing in b3ubot currently *writes* a step file or drives this format as part of a live process — that is U2.2 (the emitter + round-trip oracle), U2.3 (ledger operations / status transitions as an API), and U2.4 (retro-pair machinery). Actual execution/control — the engine autonomously driving skeleton → ratify → execute → verify → retro and dogfooding this exact format on its own workflow (§2.13) — is P3's job, several UoWs out.

Ledger operations the orchestrator consumes: - `draft(intent) -> skeleton` (via PAL, §4 SKELETON state); - `ratify(skeleton, amendments) -> active UoW` (human gate); - `record(evidence)` during execution; - `close(retro) -> done row + hash backfill`.

Delivered state (U3.3, step.16; U3.4, step.17; U3.5, step.18; U3.7, step.20): ALL FOUR ops exist. `draft(intent) -> skeleton` is `app/orchestrator/drafting.py`'s `draft_skeleton()`: a bounded workspace probe through the §2.7 tool surface (trail-recorded), engine-owned context assembly (§2.4), a parser-validated provider draft (one bounded retry — NOT d9's budget system, which stays open), the engine's own probe record stamped into the drafted skeleton's ground-truth section, and the engine state advanced INTENT→SKELETON. `ratify(skeleton, amendments) -> active UoW` is `app/orchestrator/ratify.py`'s `ratify_skeleton()` (`b3ubot ratify`): summary + LEANs presented for override through injectable I/O, ratify / amend / discard collected, the engine's ratification record stamped into §(0) on ratify (SKELETON→RATIFIED), amendment notes recorded beside the skeleton on a decline (no state change — the pre-ratification amendments cycle is file-level), discard double-confirmed (SKELETON→DISCARDED). `record(evidence)` exists since U3.5 (step.18): `app/orchestrator/executing.py`'s loop stamps the engine's EXECUTION RECORD into the step file's §(0) on completion and writes per-node status/timestamps/run-ids into the execution DAG instance's kv attrs on every change (§2.13's live progress surface). `close(retro) -> done row + hash backfill` exists since U3.7 (step.20): `app/orchestrator/closing.py`'s `close_uow()` (`b3ubot close`) drafts the retro narrative DETERMINISTICALLY from the real evidence records (engine-state history, both instances' node kvs, the verifications.jsonl verdicts — machine-drafted V0; a PAL-narrated retro is a named post-M1 quality upgrade), freezes both DAG instances (§2.13's RETRO→CLOSED row), commits the work in a git workspace (the 2-commit corpus shape, commit-as-machine-work per §2.6) with the REAL short hash backfilled into §(7) AND the target workspace's own end_to_end.md ledger row (detected via the U2.3 machinery; absent/unparseable/row-less ⇒ a NAMED skip in the retro — never this repo's own

ledger for a target-workspace UoW), honest "(not a git repository)" markers otherwise, and advances RETRO→CLOSED. U2.4's `b3ubot ledger` CLI remains the manual form.

2.4 The provider abstraction layer (PAL)

The requirement that defines the product: users interact with b3ubot, never with the model. Everything provider-specific lives behind one interface; nothing provider-specific may leak into the ledger, the workflow, or the oracle harness.

```
┌──────────────────────────────────────────────────────────────────────┐
│ PAL contract (the envelope)                                          │
│                                                                      │
│   request:  { role-tagged context, tool schema, budget, policy }     │
│   response: { text | tool_calls[] | done, usage, provider_meta }     │
│                                                                      │
│   adapter interface (every provider implements exactly this):       │
│     capabilities() -> { tools?, streaming?, context_window,          │
│                         modalities, cost_model }                     │
│     run(request)   -> response stream                                │
│     abort(run_id)                                                    │
└──────────────────────────────────────────────────────────────────────┘
```

- **Adapters** are thin: translate the envelope to the provider's native API (Anthropic API / Claude Code headless, OpenAI API, future systems) and back. No adapter contains workflow logic.
- **The first real adapter is `claude-code`** (d1 RESOLVED 2026-07-13): it drives the **local Claude Code client** on the developer's machine (headless invocation). In the local-trinity deployment (§2.9) that client is the ONLY component that communicates with the internet — the engine's entire external-AI egress is consolidated in one process. An Anthropic-API-direct adapter is a later candidate alongside `openai` (P6).
- **The mock provider** is a first-class adapter (scripted responses from fixtures) — the whole engine must run and be tested with zero network and zero keys, the b3u keyless-mode discipline applied to AI.
- **The human-relay provider** (U1.5, step.7) is a third first-class adapter: the developer manually relays context to (and a response back from) an AI coding tool running in a SEPARATE session, standing in for the `Adapter` interface by hand. It validates the whole Adaptive Programming workflow end to end — skeleton → ratify → execute → verify → retro — before any AI-tool-specific automated adapter exists, and decouples b3ubot's

architecture from any single AI vendor's API: the workflow model is proven without waiting on `claude-code`, `openai`, or any future automated integration. It shares the mock adapter's offline, zero-network, zero-keys testability via constructor-injected `output_fn`/`input_fn` functions (default `print`/`input`), so the contract suite exercises it with zero real interactivity (d13).

- **Capability negotiation**: the orchestrator asks `capabilities()` and degrades honestly (a provider without native tool-calling gets the text-protocol fallback; one with a smaller context window gets tighter context assembly). Capabilities differences are the PAL's problem, never the workflow's.
- **Context assembly** is engine-owned: the ledger slice, workspace state, and step file are composed into the envelope by b3ubot, identically for every provider. Providers see what the policy layer (§2.10) allows — no more.
- **Provider identity appears in exactly one place**: the ProviderRun record (§3), for audit and cost accounting. The developer-facing surfaces say "b3ubot".

2.5 The oracle harness

Verification is tiered, deterministic-first:

```
T0  static      — parse/lint/type gates; ledger round-trip
T1  unit        — the workspace's test suite (pytest/ctest/... adapters)
T2  gates       — live end-to-end scripts (the p1_gates.sh pattern)
T3  CCS         — grammar round-trip via the b3u validate operation
                  (§2.8): compile the .sgr on the b3u side, decompile,
                  byte-compare (.urt/.ubr diff); verdict + cppcc-log
                  diagnostics returned; SCB byte-identity across the
                  API quartet where relevant
T4  AI review   — a SECOND provider pass critiques the diff (advisory:
                  T4 may comment, only T0-T3 may block — LOCK B-6)
```

Every ratified UoW names its gates from these tiers before execution; the orchestrator refuses to enter VERIFYING with an empty gate set. Verdicts are recorded as VerificationRecords (§3) — the ledger's acceptance checkboxes become machine-checked facts.

Delivered state (U3.6, step.19): T0/T1 are WIRED — `app/orchestrator/verifying.py`'s `verify_uow()` extracts a skeleton's machine-runnable gates (the corpus `G<n>` convention plus bracketed tier markers: `[T0:py_compile]`, `[T0:ledger_roundtrip]`, `[T1:pytest]`, `[T2:<script>.sh]` — the drafting template teaches the same vocabulary), REFUSES an empty

machine gate set (the rule above, enforced at the pass level since the state itself arrives at VERIFYING on U3.5's completion), compiles the gates into a `verification`-kind DAG (§2.13) written beside the skeleton, runs the tier chain (T0 before T1 before T2 — the ladder is a cost/determinism ordering) via the shared frontier engine with exit-as-data verdicts through the scrubbed allowlisted shell, and records every verdict BOTH into the instance's node kvs and into the file-based VerificationRecord precursor `~/.b3ubot/verifications.jsonl` (Track B's real row is P4). Green ⇒ VERIFYING → RETRO with the engine's verification record stamped into §(0); a red gate VERDICT drives the bounded repair loop (VERIFYING → EXECUTING, diagnostics fed back through the provider context — §4; d9's V0 fixed-attempts default, 2 repairs per invocation); budget exhausted ⇒ loud escalation (state stays VERIFYING). T0 lint/type checks and non-pytest T1 suite adapters are named future vocabulary; T2 runs exactly as far as U3.2's `run_gate`; T4 markers refuse naming B-6.

Delivered state (U5.2, step.23): **T3 is WIRED** — `[T3:validate]` runs in the VERIFYING pass through the CCSService port (§2.8): every `.sgr` the execution APPLIED (the py_compile applied-set symmetry; unit = the file's stem, the b3u path-identity rule) is validated on the b3u plane; a FAIL verdict is a normal red gate whose **full** cppcc .log-tail diagnostics ride the repair loop's oracle_feedback (pulled from the verifications.jsonl row, not the substrate-truncated node kv — a uniform upgrade, all tiers); a CCS environment error (unreachable plane / auth / entitlement / missing replay fixture) is a verdict-less DEFECT, never repaired. The service is injectable (`ccs=` — tests run ReplayCCSService over the shipped fixture set); a real run resolves B3uDevClient and refuses loudly with the named next action when no plane is available. A machine-marked T3 gate is owned by the VERIFYING pass ALONE — the execution-DAG oracle_gate node remains only for prose-named checks (the planning shape). Unknown `[T3:...]` checks refuse naming U5.3 (SCB quartet byte-identity, the named future vocabulary). Proven live at step.23: the engine's pass drove the real local b3u end to end — PASS verdict → RETRO, and a broken spec's genuine path-scrubbed cppcc tail → escalation.

Delivered state (U5.3, step.24): **`[T3:scb_identity]`** — the workspace's delivered bundles (the python/+js/ layout) decompile byte-identically through the plane's OWN delivered SCB runtimes, fetched via the port's `package()` op (the contract's /packages, an additive port extension) and cached in the data home — client posture end to end, the sibling repos never on any path (b3u C-14-B inherited). Divergence/decompile failure = FAIL verdict (with the divergence byte offset); missing node / package fetch / ambiguous bundles = defects; no bundle = named vacuous pass; the workspace gains nothing (data-home scratch, removed in a finally). V0 covers the interpreted pair (python+js — the b3u first-run precedent); cpp/rust identity legs are the named, toolchain-gated future. Proven live at step.24 on a plane-generated bundle.

Delivered state (U5.4, step.25): **the CCS-unit workflow arcs** — `app/ccs/workflows.py`: `validate_spec` (the cheap arc), `deliver_unit` (the paid arc: generate through the plane, unpack INTO the workspace through the tool surface's new `write_bytes` — trail-recorded, C-4-B-scoped; regenerate = re-run, the revision moves server-side), `plane_status` — surfaced as the **`b3ubot ccs`** CLI group (validate/generate/status; exit codes 0 done / 1 spec-at-fault / 2 environment-at-fault). Round-trip is deliberately NOT an arc: oracles gate ([T3:validate]/[T3:scb_identity]), arcs assist — a `ccs validate` PASS is assistance, never evidence. Where delivery sits in the ORCHESTRATED loop is U5.5's decision (its first real consumer).

Delivered state (U5.4a, step.26): **the foreign-workspace proof + guard.** Engine-state records are STAMPED with their workspace at creation (`EngineStateRecord.workspace`; back-compat: unstamped records pass), and every stage entry refuses a cross-workspace invocation loudly — a UoW id is only unique within one project, and one data home is one id-space (per-project `$B3UBOT_HOME` is the named multi-project remedy). Proven: a COMPLETE UoW in a foreign git project (its own ledger, its own history) — ratify → execute → verify (with a `[T3:validate]` verdict from the plane, replay AND live) → close with real work+gdiff commits in the FOREIGN repo, b3ubot's own tree untouched. The U5.4a ledger row itself was added through `b3ubot ledger add-row` — the amendment path, dogfooded.

Delivered state (U5.5 Phase A, step.27) + Q-25-A RESOLVED: **delivery is an EXPLICIT post-close arc** — convergence = gates green = close (evidence frozen offline); delivery = the operator's act of consuming the converged spec (`b3ubot ccs generate`), idempotent and re-runnable, after closure. Never wired into closing.py: coupling a network delivery into the close would make evidence-freezing hostage to the plane's availability, and pay-per-use delivery belongs to the operator (LOCK C-27-C). **THE TRINITY GATE, Phase A (scripted leg): ALL PASS** — one full AP arc against the real plane: propose(broken) → real cppcc FAIL over HTTP → repair carrying the genuine tail in the provider context → converge (PASS) → close (real commits) → deliver; egress witnessed (runs/ == provider calls; the plane loopback-only). Phase B — the HUMAN-ADAPTER live leg — is the operator ceremony (`scripts/trinity_ceremony.sh` prepare/check); the gate and P5 close when its check passes (LOCK C-27-B: a machine may not declare a human-witnessed gate green). **Phase B PASSED 8/8, 2026-07-22** — the operator ran the relay end to end (ratify → paste-relay → live T3 PASS from their own instance → the engine's 2-commit closure in the ceremony repo → delivery), and the first real ceremony surfaced and fixed TWO live-fire defects no injected-I/O test could reach (Q-27-D: the one-line paste reader; Q-27-E: the check's clean-tree leg contradicting Q-25-A's post-close delivery). **P5 IS**

COMPLETE: the local trinity runs the Adaptive Programming loop end to end with deterministic CCS verdicts — the design's central claim, now a passed gate.

Delivered state (U5.6, step.28 — THE REMOTE TRINITY, by amendment, the (p1.1) plan): §2.8's "the hosted b3u.dev later is this same class pointed at a different base URL" is now a PASSED gate, both phases. B3uDevClient carries the Cloudflare Access service token as session-level headers when `$CF_ACCESS_CLIENT_ID`/`_SECRET` are set (C-28-B: credentials, env-only; absent env = the local client byte-for-byte). The egress witness asserts the DECLARED plane (C-28-A): loopback claim intact locally, https + `$B3U_BASE_URL` equality remotely — where CCS bytes went is asserted, never assumed. `scripts/remote_gates.sh` (boots nothing, C-28-C) ran the same 3-leg trinity oracle ALL PASS against https://staging.b3u.dev on the operator's real paid-TEST account; live-fire finding Q-28-D (staging ran pre-UB.1 code; the verdict-less-defect taxonomy refused to call the 404 a red gate — redeploy was the fix). **Phase B PASSED 8/8, 2026-07-23** — the remote relay ceremony under the fresh `~/.b3ubot.1` home: T3 PASS from the remote plane, 2-commit closure in the ceremony repo, 22-file 4-language delivery from staging; Q-28-E recorded the relay paste channel's fragility (V0 shape guard + TTY guard both held; the cached-done relay node needed the engine-record repair path once). The trinity is now host-agnostic in fact, not just by design.

Delivered state (U5.7, step.29 — THE MULTI-HOME DISCIPLINE, by amendment, (p1.1.4)): one dev box now serves many staging accounts without a mis-pairing hazard. `scripts/b3ubot_profile.sh <N>` binds the data home `~/.b3ubot.N` (exported as `$B3UBOT_HOME` — the engine-wide contract) to the credentials living INSIDE that home (`~/.b3ubot.N/profile.env`, 0600-enforced, sourced LAST so the profile beats a polluted outer environment); there are no flags and no creds arguments, so account identity FOLLOWS the home (C-29-B) — the b3u lifecycle plan's TC-16 asserted on the wrapper, not the hope. Ceremony-proven 2026-07-24: profile 1 = alex/betthreetesting, profile 2 = actor/uafco (the A3 actor, created live: real signup, gmail verify, checkout run THROUGH the wrapper's own env, 4242 via Link), `ccs status` through each serving its OWN account including the polluted-outer-env leg; TC-14 census: 3 accounts, subscriptions isolated, A1/A2 untouched. Zero engine-code changes (C-29-C): the wrapper only consumes the existing `$B3UBOT_HOME` + env-only-creds contracts.

2.6 The workspace manager

The user's project is a git repository under management. Rules:

- b3ubot works on branches/commits with the same porcelain discipline the siblings use (clean tree per closed UoW; commit messages carry the step trail).
- The engine never rewrites history it did not create.
- Workspace reads feed context assembly (§2.4) through the egress policy (§2.10); workspace writes happen only through the gated tool surface (§2.7) in EXECUTING state.

2.7 The tool surface

What providers may *request*; the orchestrator is what *acts*:

- file read/write/edit, directory listing;
- shell execution (allowlist + workspace-scoped, timeouts, output capture);
- git operations (status/diff/add/commit on the working branch);
- oracle invocation (run a named gate tier).

Every tool call is policy-gated (§2.10), logged to the ProviderRun trail, and replayable. Destructive operations (delete, history rewrite, pushes) require explicit human approval — the same "refuse with a named next action, no --force" posture the b3u launcher established.

Delivered state (U3.2, step.15; approval U3.4, step.17; DAG wiring + env scrub U3.5, step.18): `app/tools/` implements this surface, and since U3.5 the execution DAG's nodes act through it exactly as designed (`executing.py`'s handlers: `write_file` for tool_call, `run_gate` for oracle_gate, the probe reads for context_assembly — all trail-recorded). Q-15-B is resolved: allowlisted children get a SCRUBBED environment (secret-named vars dropped — B-4's provider keys never reach a child; `scrubbed_env()` in `app/tools/shell.py`). "Require explicit human approval" is real since U3.4: an injectable approval callback (`app/tools/approval.py`) gates `delete_file` and `git_delete_branch` — no callback refuses (unattended paths can never destroy), a decline refuses with the human-declined message, a grant executes with the approval on the trail; `git_push` / `git_rewrite_history` still refuse unconditionally, each for its own named reason (remote/refspec policy undesigned; no concrete rewrite verb + no §2.6 provenance record) and take no callback — no fake flows. One interim remains, named honestly: the "ProviderRun trail" here is the file-based precursor `~/.b3ubot/tool_calls.jsonl` (`app/tools/trail.py`, unconditional — refusals included) until Track B's real row exists. The shell allowlist is the `shell_allowlist` key in `app/config/config.txt`. Full inventory: `docs/tool_surface.md`.

2.8 The CCS plane — b3ubot is a CLIENT of the b3u platform

d5 RESOLVED (2026-07-13, the combined-architecture decision): b3ubot does not embed a CCS plane. **The b3u platform IS the plane, and b3ubot is one more authenticated client of its API — exactly like b3u's web frontend.** One service, two clients: the boundary that already protects cppcc, libruntime.a, and libgenerate.a from web users (b3u LOCK C-1-C) protects them from b3ubot by the same mechanism, for free. b3ubot sends specifications and receives generated artifacts plus oracle verdicts — the same bytes a web user could get, and nothing more. b3u stays mode-agnostic: no "b3ubot mode" on the server, ever.

The engine depends on an *abstract* client port, not raw HTTP:

```
CCSService (port)
  generate(unit, spec)  -> artifacts (the b3u generate flow)
  validate(unit, spec)  -> Verdict{pass, diagnostics}
                           [b3u UB.1 — verdict-only, no materialize/
                            deliver; the cppcc .log tail is exactly
                            what the repair loop consumes]
  deliver(unit)         -> download handle (on convergence)
  entitlement()         -> {tier, quota}

Implementations:
  B3uDevClient     — real; HTTP to the b3u API, token-authenticated
                     (the b3u /login HMAC token). FIRST TARGET: the
                     LOCAL b3u instance (127.0.0.1:8941, the PL
                     runbook); the hosted b3u.dev later is the SAME
                     client pointed at a different base URL — the
                     API is the seam.
  ReplayCCSService — fixtures; zero network. Oracle verdicts are
                     deterministic ⇒ ideal fixtures.
```

Consequences:

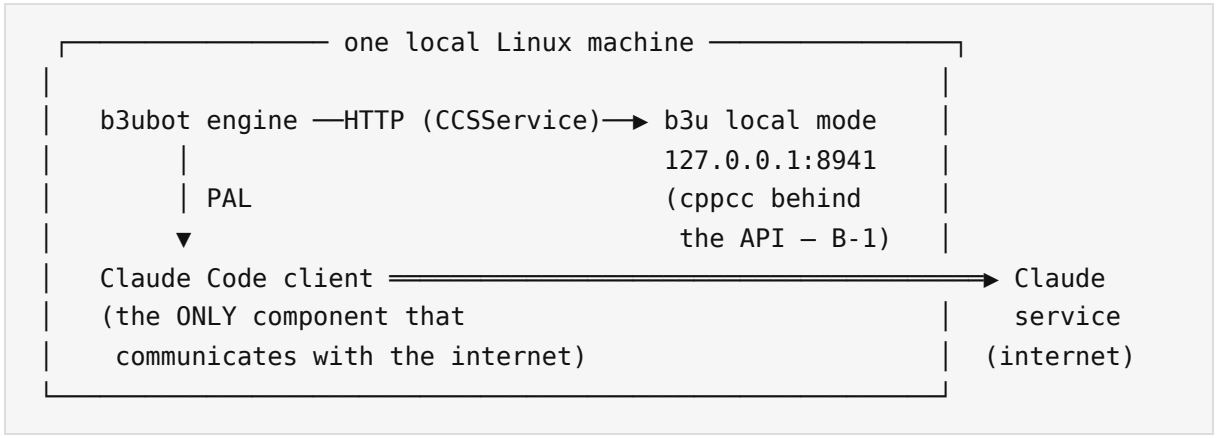
- **b3ubot contains no CCS code.** It is a coordinator, not an executor; LOCK B-1 holds *by construction*. (This also makes b3ubot independently distributable in principle — a commercial question, counsel-gated: d8, B-3.)
- **Both external dependencies sit behind ports with record/replay implementations** (AIProvider §2.4, CCSService here) — so the entire engine, the whole Adaptive Programming loop, is testable offline with zero network: scripted AI responses against scripted oracle verdicts.
- **No direct AI ↔ b3u path.** b3ubot is the sole coordinator: the AI never calls the b3u API, and b3u never calls an AI. CCS stays free of any third-party AI dependency.

- The plane serves the T3 oracle (§2.5) and CCS-unit workflows (a workspace that contains .sgr specs gets grammar-aware assistance: compile, validate, round-trip, regenerate). The oracle can only run where cppcc lives — which is a feature: the fitness signal is private, deterministic, and cannot be forged client-side.
- The iterate-cheaply/deliver-once shape dovetails with b3u's spec-only retention (b3u §2.10): the AP loop pays for verdicts many times and for materialize+deliver once, on convergence.

U5.1 REALIZED (step.22, 2026-07-22) — the port as implemented, one amendment (Q-22-A). `app/ccs/`: `service.py` (the CCSService ABC + frozen `Verdict{unit, passed, diagnostics}` / `Artifacts{unit, revision, spec_sha256, zip_bytes}`), `replay.py` (ReplayCCSService — content-keyed sha256 lookups, refuse-loudly on missing fixtures; shipped set under `tests/fixtures/ccs/`), `client.py` (B3uDevClient — requests-based, written against **b3u/docs/api_contract.md v1** exclusively, lazy /login with one transparent re-login on 401, credentials env-only per B-4). The amendment: the sketch's `deliver(unit)` cannot exist against contract v1 — generate's 200 response IS the delivery and b3u keeps no artifact at rest to fetch later — so `generate()` returns the Artifacts handle directly and `health()` (the liveness probe a trinity client needs) takes the fourth port slot; a server-side deliver op, if ever wanted, is additive under the contract's own law. Proven live at step.22: `scripts/p5_gates.sh` boots a real local b3u via the sibling's own PL runbook and runs the 7 live client legs 7/7 — the trinity's b3ubot→b3u leg exists for real.

2.9 Deployment shapes (phased) — THE LOCAL TRINITY first

The development deployment is the local trinity (user directive 2026-07-13): b3u already operates successfully in local mode, so b3ubot's first working configuration puts all three components on one Linux machine —



b3u and b3ubot remain **entirely local**; the Claude Code client is the sole internet-communicating component. This provides a fully functional development environment from day one while preserving the privacy of CCS operations and establishing a clean separation between local orchestration and external AI services.

```
P1-P6 (development)     the local trinity, single developer — the bot
  local-first             runs where the workspace lives (the b3u PL
                          lesson: local mode first made everything else
                          honest). CLI surface; provider keys env-only;
                          b3u via its scripts/b3u_local.sh runbook.

production (post-P7)    the SAME topology with the hosted b3u.dev as
  [milestone-level]       the CCS plane (B3uDevClient pointed at a
                          different base URL — the API is the seam);
                          the engine as a long-running process;
                          multi-workspace; exposure gated on hardening
                          (LOCK B-2) exactly as b3u P5 gated b3u.dev.
```

2.10 Security model

Standing locks (the founding set; steps add C-N-x locks as usual):

```
+------+-------------------------------------------------------------+
| B-1  | cppcc + CCS runtime libraries NEVER leave the server / ship   |
|      | in any artifact (the C-1-C inheritance).                      |
| B-2  | No public exposure before a hardening phase closes (the       |
|      | C-1-D inheritance): development posture is TRUSTED INPUT +    |
|      | loopback-only surfaces.                                       |
| B-3  | Local-only repo; no disclosure of CCS architecture, bench     |
|      | numbers, SCB format, or the patent without explicit approval  |
|      | (the C-1-E inheritance).                                      |
| B-4  | Provider credentials are environment-only: never committed,   |
|      | never logged, never echoed into gate output (the C-21-C/D     |
|      | family, generalized to every provider).                       |
| B-5  | EGRESS POLICY: nothing from a workspace reaches a provider    |
|      | except through the policy layer — path allow/deny lists,      |
|      | secret redaction, per-workspace consent. The developer can    |
|      | always answer "what left this machine, when, to whom" from    |
|      | the ProviderRun trail.                                        |
| B-6  | Deterministic oracles gate; AI review advises. T4 may comment |
```

```
|      | on a diff; only T0-T3 verdicts can block a UoW's closure.     |
+------+---------------------------------------------------------------+
```

B-5 is the lock that is NEW relative to b3u: b3u's privacy story protected the platform's IP; b3ubot must additionally protect the *user's* code from over-sharing with whichever AI provider is active.

The local trinity (§2.9) makes B-5 auditable **by topology**: exactly one component — the Claude Code client behind the `claude-code` adapter — communicates with the internet at all; b3u and b3ubot never do. What the policy layer allows into a prompt is therefore also the complete list of what can leave the machine.

2.11 Separation of concerns (the load-bearing boundaries)

- **PAL ⊥ workflow**: no provider name outside adapters + ProviderRun rows.
- **Orchestrator ⊥ ledger**: the state machine consumes ledger operations; the ledger never drives state.
- **Oracles ⊥ AI**: T0-T3 run without any provider present (mock-provider CI must exercise them all).
- **Engine ⊥ workspace**: b3ubot state lives in its own store (§2.12); workspace repos carry only their own code + the step trail in commits.
- **CCS plane ⊥ everything**: reachable only through its service boundary.

2.12 Repository layout & storage

```
b3ubot/
  README.md  design.md  end_to_end.md
  app/            — the engine (orchestrator, PAL, ledger, oracles)
  adapters/       — provider adapters (claude/, openai/, mock/)
  dag/            — the workflow DAG substrate (§2.13): cloned FORK of
                    bet3/bot/dag (PROVENANCE.md; decision d11) —
                    dagModel.sgr/dagCommands.sgr, cppcc frontend, SQL
                    schema, C++/Py/JS/Rust SCB OO APIs, cctdag CLI,
                    regression suite
  b3ubot_sgdl/    — the data-model grammar (Track B, §3)
  b3ubot_sqlite/  — schema + OO API (Track B)
  b3ubot_cli/     — the model CLI (Track B)
  docs/           — live inventory docs
  scripts/        — launchers, gates, oracles
  steps/          — step.N.txt + retro pairs
```

```
    tests/          — one suite per UoW oracle, regression forever
    journals/       — local scratch (never under git)
```

Engine state: SQLite via the Track B pipeline (§3) for modeled entities; file trees for ledgers, artifacts, AND workflow DAG instances (the b3u BLOB/URI split: the DB stores pointers, files store bytes; `.dag` files store graphs). Per-developer data home (`~/.b3ubot`) with the b3u local-mode conventions: repo never written, backup/restore as one relocatable tarball.

2.13 The workflow DAG substrate

The model: a b3ubot workflow is a collection of DAG instances created on the fly while engine activities execute. The substrate is the `dag/` subtree — a fork of the `bet3/bot/dag` multi-target project (d11): one SGDL source of truth (`dagModel.sgr`) for the canonical `.dag` textual form, plus a cppcc-generated parser, an SQLite schema, OO APIs in all four SCB languages (byte-identity matrix), the `cctdag` CLI, and a regression suite with a 2-pass round-trip oracle.

How the orchestrator (§4) uses it:

lifecycle event	DAG activity (on the fly)
RATIFIED → EXECUTING	compile the ratified skeleton into an EXECUTION DAG: nodes = tasks (context assembly, provider runs, tool calls, oracle gates, human gates); edges = dependencies; kv attrs carry engine state (status, run ids, timestamps)
during EXECUTING/VERIFYING	node kv state updated as tasks complete; the DAG IS the live progress surface — what may run next = the ready frontier
RETRO → CLOSED	the instance is FROZEN as evidence and pointed at by the Step's WorkflowDag row
cross-step / project level	standing DAGs (dependency graphs, TODO graphs — the bootstrap kinds' lineage) updated incrementally as UoWs close

Why a CCS grammar and not an internal object graph:

- **The engine's own workflow state is CCS-verifiable.** `.dag` files parse under the generated parser, load in four languages byte- identically, and round-trip through the dag regression

oracle — so the T3 tier applies to *b3ubot's own workflow store*, not just to user grammars. The engine dogfoods its flagship oracle.

- **Files-as-truth extends naturally** (d4): DAG instances are text files beside the ledger; the DB (§3) stores pointers.
- **Tooling is free**: `cctdag` CRUD, the bootstrap generators' kind lineage (Hierarchy / Dependencies / TODO), and the byte-identity matrix arrive with the fork instead of being built.

Delivered state (U3.1, step.14; U3.5, step.18; U3.6, step.19; U3.7, step.20): the first THREE rows of the lifecycle table are code. RATIFIED→EXECUTING compiles the ratified skeleton (`compiler.py`, the fixed V0 template) and U3.5's `executing.py` writes the instance to `steps/<step>.execution.dag` BESIDE the skeleton (files-as-truth: per-step evidence lives where the step file lives), schedules from the ready frontier — generically, any DAG shape — and rewrites the instance's node kvs on every status change, verifying the written form through the substrate's own `bin/dagModel` round trip. The `verification` kind is live since U3.6 (step.19): each VERIFYING pass compiles the skeleton's machine-runnable gates into a `verification`-kind instance at `steps/<step>.verification.dag` (same beside-the-skeleton, same round-trip verification, same shared frontier engine; `version` = the pass number, prior passes' verdicts in `verifications.jsonl`). The RETRO→CLOSED freeze is live since U3.7 (step.20): `closing.close_uow()` stamps a `frozen_at` dag-level kv into BOTH instances (design §3's WorkflowDag field name, verbatim) with a CODE-ENFORCED no-further-writes rule — `executing.refuse_if_frozen()` makes executing.py/verifying.py refuse loudly (trail-recorded) before rewriting frozen evidence, the file-level second wall behind the state machine's terminal CLOSED — and records the file-based WorkflowDag POINTER precursor (`~/.b3ubot/workflow_dags.jsonl`: uow/step/kind/dag_uri/status/ frozen_at — Track B's real row is P4), so "pointed at by the Step's WorkflowDag row" has an honest interim form.

b3ubot extends the substrate (in THIS tree — the fork rule) with workflow kinds: `execution` (one per EXECUTING entry), `verification` (the gate graph of a VERIFYING pass), `dependencies` and `todo` at project scope. The v0 grammar is deliberately permissive about kinds; kind vocabulary and per-kind structural validation live at the loader/OO-API layer, exactly as upstream designed it.

2.14 Licensing & subscription posture — a b3ubot subscription IS-A b3u subscription

The licensing decision (2026-07-13; b3u/journals/b3ubot_license.md, ToS `cppcc/docs/b3u-saas-terms-example.0.1.md` §5.3.5). b3ubot follows the **same licensing model as b3u.dev** — it is not a separate commercial product. A **b3ubot subscription is an is-a specialization of a b3u subscription**: it inherits the base and extends it with the b3ubot capability.

```
b3u subscription (base)  ──is-a──▶  b3ubot subscription
  registration   ──────────────────  a b3u.dev Account (inherited)
  authentication ──────────────────  the b3u.dev infrastructure (inherited)
  billing        ──────────────────  Stripe Level I seat sub (inherited)
  royalty        ──────────────────  Level II on CCS-Unit Revenue (inherited)
                  + EXTENDS with ──  the Adaptive Programming Engine capability
```

- **Registration & authentication.** A b3ubot installation is **registered through a b3u.dev Account, using the b3u.dev website**, with authentication and licensing managed by the **same infrastructure**. No second account, no second auth system. This is the concrete form of the §2.8 "b3ubot is a client of the b3u API" posture: b3ubot authenticates AS a b3u.dev subscriber (design §2.12 of the b3u sibling; b3u §4.8).
- **Royalty is channel-neutral.** If b3ubot is used to develop or maintain production software that incorporates artifacts generated by **cppcc**, the **same Level II royalty and licensing terms apply** as for any b3u.dev Account. The model is consistent whether CCS operations are initiated **directly** through the b3u.dev website or **indirectly** through b3ubot as part of an Adaptive Programming workflow — the round-trip validate/generate calls b3ubot makes (§2.8) are the same licensed CCS operations, metered against the same subscription.
- **What b3ubot adds** is the Adaptive Programming Engine itself (the orchestrator, PAL, ledger, oracle harness, DAG substrate) — a *capability* on top of the inherited b3u subscription, not a new billing/auth/royalty spine. This is the RESOLUTION of §5 d8 (commercial framing): **specialization, not a separate product.** The remaining d8 question — whether the b3ubot capability is a distinct b3u *tier* or an add-on marker on an existing subscription — is a pricing detail for the b3u side (b3u §4.8: a capability marker on the Subscription), still counsel-gated (B-3).

This does NOT change the security posture: LOCK B-1 (cppcc server-side) and the §2.8 client boundary hold; b3ubot exercises a b3u subscription's CCS entitlement, it does not gain privileged access to CCS.

2.15 Payment flows

(This section of the live document sets out the two independent payment relationships a user maintains — the b3u.dev subscription and the user's own provider account — and is held out of the published copy pending a commercial-disclosure decision. Its operative consequence appears throughout this document and in the published step files: b3ubot is bring-your-own-key. No provider credential is held on the user's behalf, and no provider spend flows through b3u.dev.)

---

3. Data Model

Built through the five-layer isomorphic pipeline (the §3.6-style lockstep, non-negotiable): verbal description → `B3UBOT.sgr` SGDL → SQLite schema → C++ OO API → model CLI, each layer 1:1 with its neighbors, closed by a round-trip chain oracle. Post-hoc entities land at all five layers in one step (the b3u RoyaltyDeclaration precedent).

First-cut entities (the verbal layer's seed; Track B refines):

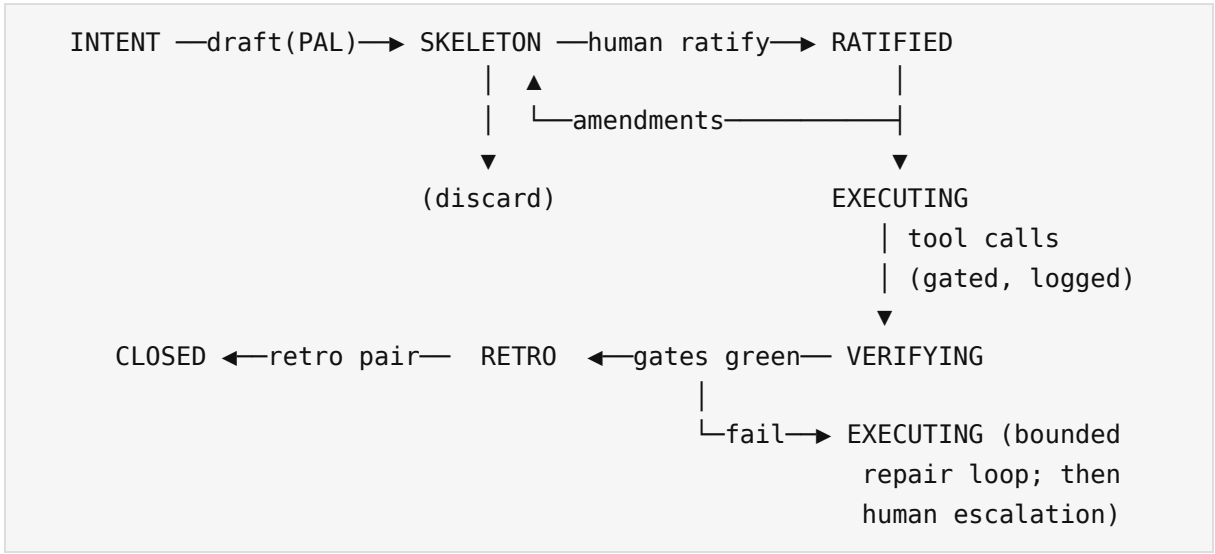
```
+--------------------+-----------------------------------------------------+
| Workspace          | a managed repo: root, vcs state, egress policy      |
| Project            | a goal arc inside a workspace; owns Steps           |
| Step (UoW)         | the ledger unit: skeleton, state, gates, hashes     |
| ProviderRun        | one PAL invocation: provider id, usage, tool        |
|                    | trail, egress digest (THE audit row — B-4/B-5)      |
| VerificationRecord | one oracle verdict: tier, gate name, pass/fail,     |
|                    | evidence pointer                                    |
| Artifact           | a produced file set: kind, uri, sha256, retention   |
| WorkflowDag        | a materialized workflow graph (§2.13): kind         |
|                    | (execution/verification/dependencies/todo),         |
|                    | dag_uri (the .dag file), status, frozen_at          |
+--------------------+-----------------------------------------------------+
```

Relationships: Workspace 1—n Project 1—n Step; Step 1—n ProviderRun and 1—n VerificationRecord; Artifact and WorkflowDag rows hang off Steps (project-scoped WorkflowDags — dependencies/todo — hang off Projects). Validation rules (state graphs, uniqueness, referential guards) follow the b3u V1..Vn pattern and are typed in the OO API. The WorkflowDag row stores a POINTER; the graph's truth is the `.dag` file under the dag substrate's round-trip oracle (§2.13).

---

4. The Adaptive Programming Workflow

The orchestrator's state machine — the step paradigm as executable states:



Human gates are load-bearing and few: **ratify** (a skeleton never executes unratified — LEANs are presented for override exactly as the b3u steps present them) and **escalation** (a repair loop that exhausts its budget stops and reports; it never grinds silently). Everything between the gates is machine: context assembly, provider runs, tool acting, oracle invocation, evidence recording, retro drafting.

The workflow's materialized form is DAG instances (§2.13). Entering EXECUTING compiles the ratified skeleton into an on-the-fly execution DAG (tasks as nodes, dependencies as edges); the orchestrator schedules from the DAG's ready frontier, writes task state into node kv attributes, and freezes the instance at CLOSED as the step's evidence graph. A workflow at any scope — one step, one verification pass, a whole project's dependency or TODO structure — is a *collection of DAG instances*, each a `.dag` file under the substrate's round-trip oracle.

The CCS integration point: for workspaces containing SGDL, the VERIFYING state's T3 tier runs the grammar round-trip on the b3u side (the validate operation — §2.8): compile, decompile, diff — so grammar-touching changes carry a deterministic correctness verdict no general-purpose assistant can offer. The verdict's diagnostics (the cppcc .log tail) feed the bounded repair loop as `oracle_feedback` on the next PAL proposal — the fitness signal that makes the loop *adaptive*.

This 7(+1)-state machine vs the coarser `end_to_end.md` §11 ledger vocabulary (`todo`/`active`/`done`/`dropped`) — resolved (step.14, U3.1,

`app/orchestrator/state_machine.py`): ledger `active` is the superstate covering SKELETON through RETRO (from the moment a UoW is promoted to a `steps/step.N.txt` file — §0/§12's "step.N is assigned when a UoW goes active" — up to, but not including, CLOSED); `done` = CLOSED; `dropped` = the discarded/abandoned terminal (SKELETON's own `(discard)` branch, or — no distinct diagram edge of its own — an `active → dropped` ledger fact for a UoW abandoned later in the span); `todo` = INTENT. Full reasoning, citations, and the empirical confirmation (this repo's own steps 7–13 all collapse SKELETON through RETRO into one single-pass session) in `docs/orchestrator_state_ machine.md` and the state-machine module's own docstring.

---

5. Open decisions (carried forward)

#	Decision	Status
d1	First provider adapter (RESOLVED 2026-07-13, the trinity directive): the `claude-code` adapter drives the LOCAL Claude Code client — the trinity's sole internet-touching component (§2.9). Anthropic-API-direct = a P6 candidate alongside `openai`.	RESOLVED 2026-07-13 (CC client)
d2	Bot surface: CLI/TUI first vs web-first (the b3u PL lesson argues local CLI first; web rides a later phase)	LEANING CLI
d3	Engine stack: Python-first (b3u's proven convenience, C-1-F style non-lock) vs compiled core from day one	LEANING Python
d4	Ledger storage: files-as-truth (steps/ dirs, DB indexes) vs DB-as-truth with file export (files won everywhere so far)	LEANING files
d5	CCS plane (RESOLVED 2026-07-13, the combined-architecture decision): b3ubot is a CLIENT of the b3u API through the CCSService port (§2.8) — no embedded cppcc, no CCS code in this repo. First plane = the LOCAL b3u instance (its PL runbook). Lifecycle coupling ACCEPTED; b3u's §8b BI side-track (the UB.1 validate op) is the other half of the contract. ReplayCCSService keeps the engine offline-testable.	RESOLVED 2026-07-13 (client)
d6	Egress policy default: allowlist (nothing leaves unless named) vs denylist (everything but secrets). Allowlist is the honest default for other people's code	LEANING allowlist
d7	Multi-workspace/multi-user timing: single-developer engine until production phase, or tenancy earlier	OPEN
d8	Commercial framing (RESOLVED 2026-07-13, user directive + b3ubot_license.md): NOT a separate product — a b3ubot subscription IS-A specialization of a b3u subscription	RESOLVED 2026-07-13

	(§2.14): same registration / auth / Stripe billing / Level II royalty, extended with the b3ubot capability. ToS §5.3.5 (b3u-saas-terms-example.0.1.md); b3u §4.8. Sub-detail (tier vs add-on marker) is a b3u-side pricing call, counsel-gated (B-3). Registration is through a b3u.dev Account.	(is-a b3u sub)
d9	The repair-loop budget: fixed attempts vs cost-budgeted vs oracle-signal-driven (stop when verdicts stop improving). V0 DEFAULT set (2026-07-21, U3.6/step.19, verifying.py): FIXED attempts, DEFAULT_MAX_REPAIR_ATTEMPTS = 2, scoped per verify_uow() invocation -- re-running verify after an escalation IS the human overriding the budget, which is exactly the escalation contract (design §4: stop and report, never grind silently). CLI override: --max-repairs. Why a default and NOT a resolution: cost-budgeting needs a cost model the PAL does not carry yet (usage tokens exist, prices do not), and oracle-signal-driven ("stop when verdicts stop improving") needs a verdict lattice richer than per-gate pass/fail -- both are real future policies with real prerequisites, and picking one against zero operational data would be speculation. The row stays OPEN until real repair-loop mileage picks a winner.	OPEN (V0 default set 2026-07-21, step.19)
d10	T4 review provider: same provider as the executor vs DIFFERENT provider by policy (diversity catches what redundancy cannot)	LEANING different
d11	dag/ co-evolution: FORK (independent evolution in this repo, user call at founding: "for a long run it would be better solution") vs mirror-with-sync-script (the scbcpp_sync pattern) vs upstream-first. Cloned from bet3/bot/dag @ 4f54126; divergence is deliberate (dag/PROVENANCE.md).	RESOLVED 2026-07-12 (fork)
d12	Production AI access path: the trinity's Claude Code client rides the OPERATOR'S subscription — fine for development, not a shippable dependency. Production = API-key adapters and/or BYO-key; decide together with d8's commercial framing. CORRECTION (step.3, U1.2, 2026-07-18): the "rides the operator's subscription — fine for development" framing held for a HUMAN interactively driving the Claude Code client, but not for b3ubot's own adapter invoking it headlessly. The safe, reproducible, automatable invocation mode (`claude --bare -p ...`) authenticates via $ANTHROPIC_API_KEY ONLY — not the interactive OAuth/keychain session — so adapters/claude/adapter.py requires the key explicitly even in DEVELOPMENT, and never falls back to an ambient session (C-3-B). RESOLVED (user decision, 2026-07-18, b3ubot session): Model 1 (BYO-key) SELECTED; Model 2 (centrally-provisioned API-key adapters) REJECTED. Reasoning (user's own): Model 2 would require the b3ubot licensing model / EULA to name a	RESOLVED 2026-07-18 (BYO-key)

	specific AI provider on the user's behalf -- real legal/contractual surface -- and provides NO business value unless BET THREE LLC charges an additional fee or percentage to cover and manage that provider relationship, which is not part of the current model. Model 1 keeps BET THREE LLC's revenue EXACTLY as already defined in the EULA -- the Level I subscription fee + Level II royalty -- no new revenue stream, no new AI-provider cost center, no new liability for a third party's pricing/availability. CONSEQUENCE: every b3ubot deployment supplies its OWN Anthropic API key (or future equivalent provider credential); this is not a dev-only placeholder to be replaced later -- adapters/claude/adapter.py's step.3 shape (an explicit $ANTHROPIC_API_KEY, never a BET THREE LLC-held key) IS the production shape too, unchanged. Downstream: b3ubot onboarding for ANY deployment (not just dev) needs a documented BYO-key setup step (later UoW, not built here); the EULA should eventually state AI-provider costs are the user's own responsibility, separate from the b3u/b3ubot subscription (attorney's call, C-49-D-style -- not drafted here). Does NOT resolve d14 (metering b3u's OWN CCS oracle/validate calls -- a different resource, b3u's compute, not AI-provider billing) or d15 (API-client auth shape) -- both stay OPEN, unaffected by this decision.		
d13	Human-relay adapter as a DEVELOPMENT/VALIDATION AID (RESOLVED 2026-07-20, U1.5/step.7, user directive): the whole Adaptive Programming workflow can be validated with the human developer standing in as the relay between b3ubot and an AI coding tool running in a SEPARATE session (e.g. Claude Code in one terminal, b3ubot in another) -- the developer copies b3ubot's request context across and pastes the tool's response back. This is NOT a replacement for d1's claude-code adapter (still the first REAL, automated adapter) and NOT a claim to be "the first adapter" -- it is a THIRD first-class adapter (adapters/human/) that proves the orchestrator/PAL/oracle workflow does not require ANY AI-specific automated integration to validate: b3ubot only ever talks to the `Adapter` interface, so a human can implement that interface by hand without the rest of the engine knowing the difference. Decouples workflow validation from adapter automation timing -- future automated adapters (openai, Anthropic-API-direct, …) can land on their own schedule without gating whether the workflow itself has been proven. Genuine architectural tension found and reported (step.7, not papered over): `run()` yields exactly one Response (kind == DONE, carrying the pasted-back text directly) since a single paste-back is both the content and the conclusion	RESOLVED 2026-07-20 (human-relay adapter)	

```
|    | -- narrower than the TEXT-then-DONE shape mock/claude-code   |          |
|    | both use, and narrower than `app/cli.py`'s `propose_ask()`,  |          |
|    | which hard-requires a TEXT-kind response. `--provider human` |          |
|    | is therefore NOT wired into the `ask` CLI subcommand this    |          |
|    | step (it would fail every invocation); the adapter is used   |          |
|    | directly against the PAL today (adapters/human/adapter.py,   |          |
|    | tests/test_human_adapter.py).                                |          |
|    | ADDENDUM 2026-07-20 (U1.6/step.8, Q-8-A/Q-8-B): the CLI is    |          |
|    | now wired. Q-8-A resolved Option A -- `propose_ask()`         |          |
|    | generalized to accept EITHER a distinct TEXT response         |          |
|    | (mock/claude-code) OR a content-bearing terminal DONE         |          |
|    | (human's shape), mirroring test_pal_contract.py's own step.7  |          |
|    | extension exactly; `HumanRelayAdapter` itself is unchanged    |          |
|    | (Option B stayed rejected). Q-8-B resolved (ii) -- `main()`   |          |
|    | refuses loudly, naming the next action, when `--provider      |          |
|    | human` is invoked with a non-TTY stdin, rather than let       |          |
|    | `input()` block forever on an unattended run (the §2.7 tool-  |          |
|    | surface posture, "refuse with a named next action, no         |          |
|    | --force"); no new flag added. The guard lives in `main()`,    |          |
|    | not `propose_ask()`/`_make_adapter()`, which stay pure and    |          |
|    | testable with injected I/O regardless of the test process's   |          |
|    | own stdin.                                                     |          |
+----+-----------------------------------------------------------------+----------+
```

---

6. How to maintain this file

- This is the **live, authoritative design document**. Update it in the same commit as the structural change it describes (the b3u/cct_repo architecture.md discipline).
- §5 decisions: resolve rows in place (mark RESOLVED + pointer to the step that resolved them); never delete rows.
- Design changes flow downhill into `end_to_end.md` in the same commit — the plan must never describe a road to an engine this document no longer specifies.
- Step files record execution; this file records what IS. When they disagree, this file is wrong exactly until the same-commit fix lands.