Source: cppcc docs/b3ubot_workflow.md @ ff32fc8 — curated 2026-08-08; edits land in the source first.

Adaptive Programming

A user's guide, with b3ubot as the worked example

Draft 0.1 — 2026-08-08

If you have opened one of the published `b3ubot` step files and wondered what you were looking at — who wrote it, what it was derived from, and why it is shaped that way — this document is the answer.

It describes a way of building software in which a human writes specifications in prose and a machine writes the code. b3ubot is used throughout as the worked example because its entire record is published: 29 specifications, 29 retrospectives, and the two living documents they descend from.

---

1. The question this answers

A reader who opens `step.1.txt` cold sees a file that begins:

```
step.1.txt -- b3ubot U0.4: dev environment + smoke check (P0.M1)

Step:     1  -- promote UoW **U0.4** (end_to_end.md §2, P0.M1) to
                execution depth …
Parent:   end_to_end.md §2 P0.M1 … + design.md §1.3 … + §2.8/B-1 …
UoW:      U0.4 -- "Dev environment + smoke check". Depends on U0.1-U0.3
                (done, founding commit 03ba794). Sole blocker of U1.1.
Date:     2026-07-18  (SKELETON -- not executed.)
Origin:   User directive 2026-07-18: "create the first step file…
                based on the current project status"
```

Every question a newcomer has is answered by that header, once you know how to read it.

Who wrote it, and why this text? `Origin:` records the human instruction, quoted. A step file always says what caused it to exist.

Where did its content come from? `Parent:` cites the two documents it derives from, by section. Nothing in a step file is invented; it is a *promotion* of something already agreed.

What is a "UoW"? A unit of work — one entry in a plan that existed before this file did, with its dependencies and its dependents named.

So the honest answer to "why does this text say what it says" is: it was derived, under rules, from two documents that came first. Those documents are published alongside this one.

2. Three kinds of document

Adaptive Programming as practised here uses exactly three, and keeping them distinct is most of the discipline.

document	answers	changes when
design.md	what are we building and why	the SHAPE changes
end_to_end.md	in what order, and where are we now	the ROAD changes — far more often
step.N.txt	this one piece, in enough detail to do	never; a step is written once and closed

The rule that keeps them from collapsing into each other is one sentence, and b3ubot's own design document states it: *nothing in the plan redefines the design, it only sequences it.*

A step file may not invent either. If executing a step reveals that the design was wrong, the design document changes — deliberately, as its own act — and the step records that it happened.

3. What existed on day one

b3ubot's founding commit, 12 July 2026, contained no code at all. It contained three files: a README, a design document of 442 lines, and an implementation plan of 327 lines.

The design document already had six sections, and the fourth was titled *The Adaptive Programming Workflow* — the method was specified before the thing it would build.

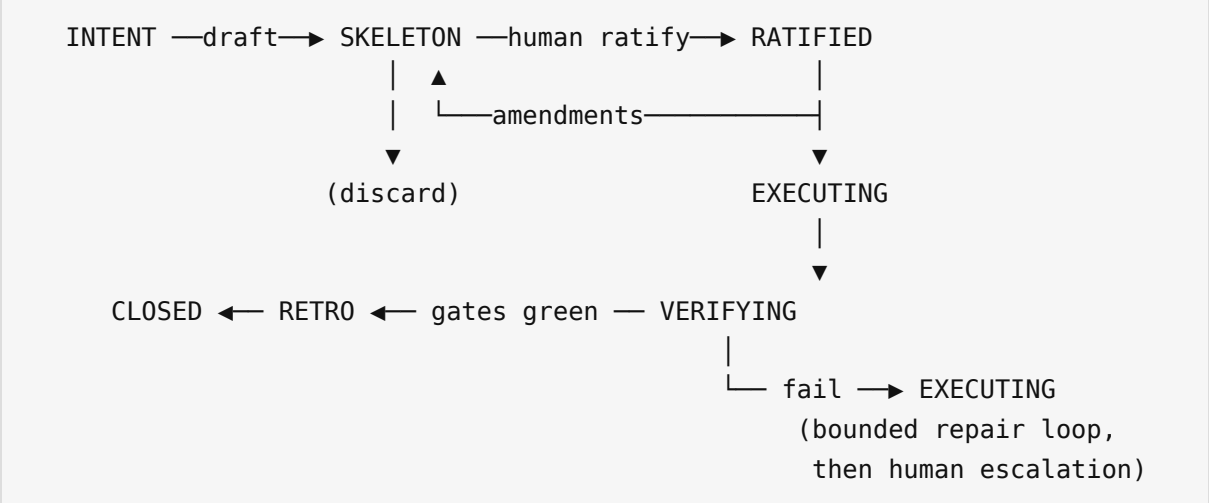
The plan already had the whole road: phases P0 through P8, about thirty units of work, and a ledger table with every one marked `todo`.

Nothing was built for six more days. The first step file, `step.1.txt`, promoted `U0.4` — the fourth unit of work — and its own status line reads `SKELETON -- not executed. Execute on explicit request.`

That gap is not incidental. **The specification was complete, agreed, and idle before any code existed.**

4. How a step is executed

The design document's §4 gives the state machine. In prose:



Two human gates, and only two. *Ratify* — nothing executes unratified, and the draft is presented with its open questions and a recommendation for each, so ratifying is a real decision rather than a rubber stamp. *Escalation* — a repair loop that exhausts its budget stops and reports; it never grinds silently.

Everything between those gates is machine work: assembling context, running the provider, acting through tools, invoking oracles, recording evidence, drafting the retrospective.

The bounded repair loop is what makes it *adaptive*. When verification fails, the failure's diagnostics are fed back as input to the next attempt. The loop is not "try again"; it is "try again knowing precisely what was wrong". Where the workspace contains a grammar, that signal is a deterministic round-trip verdict rather than a test that merely passed or failed.

5. What the record actually shows

Both living documents are published beside this one. Their history is the evidence for everything above.

document	founding	today	commits that touched it
design.md	442 ln	1 154 ln	27
end_to_end.md	327 ln	590 ln	63

The interesting number is the ratio. The plan was edited **more than twice as often** as the design, while growing far less. That is the separation of §2 working: the road is re-sequenced constantly — units added by amendment, statuses moved, dependencies discovered — while the design changes only when something is genuinely learned about the shape of the thing.

The design document grew by 2.6× nonetheless. It is a *living* document, not a specification frozen at the start. Sections were added as decisions resolved; its §5 carries thirteen numbered decisions with their status, several still open.

6. Reading a published step

Take any file from the `b3ubot` use case. The shape is consistent:

- **The header block** — `Step`, `Parent`, `UoW`, `Date`, `Origin`, as §1 described. Provenance before content.
- **Status** — `SKELETON` (drafted, not executed), or a later state.
- **What it proves / explicitly out of scope** — the second is as important as the first. A step that does not say what it is *not* doing will absorb neighbouring work until it cannot be verified.
- **The body** — the specification proper.

- **Gates** — how anyone will know it worked, written *before* it is done.

Then read its retrospective, `step.N.diff.txt`. That is where plan meets reality: what was assumed and turned out false, what the tests caught, what had to be redone.

Read the retrospectives. A record of plans alone would show a method that always works. b3ubot's retros show taxonomies catching stale state, skeletons refused by their own tests, ceremonies that failed and were re-run. That is what the method looks like from inside, and it is the part worth learning from.

7. Why the gates come first

The most transferable habit here is small and unglamorous: **a step file states how it will be verified before it states what will be built.**

The reason is specific to machine-written code. A human writing code carries an implicit model of correctness and will notice, vaguely, when something feels wrong. A generated implementation offers no such signal — it is equally fluent whether right or wrong. The gates are the only thing standing between "it produced something" and "it produced the right thing", so they cannot be an afterthought without becoming a rationalisation.

This generalises past software. Any specification whose satisfaction cannot be checked is a wish.

8. What this is not

Not automation of design. A human wrote every specification here, and the two human gates are load-bearing. What is automated is implementation, not intent.

Not a claim that generated code is good code. The published material lets you judge the correspondence between 29 specifications and the program they produced. It does not assert that the program is well-made — that is what reading it would be for, and b3ubot's code is not published.

Not friction-free. The retrospectives are in the record precisely so the cost is visible: specifications that had to be amended mid-execution, gates that failed, work that was redone. The method's claim is that this cost is *visible and bounded*, not that it is absent.

9. Applying it

Nothing here requires the tools that produced it. The transferable parts are four:

1. **Separate the three documents.** What we are building; in what order; this one piece. Do not let a task tracker become a design, or a design accumulate a schedule.
2. **Write provenance into every unit of work.** `Parent` and `Origin` cost two lines and answer, months later, the only question anyone asks about an old artifact: why does this exist.
3. **State the gates before the work.** If you cannot say how you will know it worked, the specification is not finished.
4. **Keep the retrospective.** Plan versus actual, including what went wrong. It is the only part of the record that teaches anything.

Further reading

- **b3ubot — Design** — the live authoritative design document.
- **b3ubot — End-to-End Implementation Plan** — the live plan and ledger.
- **The `b3ubot` use case** — 29 specifications and 29 retrospectives, published in full.

Both living documents continue to change; the copies published here are snapshots of a moving thing, which is itself the point.