Assignment 2: The Companion

A Walkthrough for Mutating Your Red-Black Tree with Claude Code

How to read, plan, prompt, test, commit, and finish with tokens to spare

Version 1.0 September 16, 2026

Version history appears in Section 16.

This document sits alongside Assignment 2. It replaces nothing. The assignment specification tells you *what* to build and *what* will be graded; this companion tells you *how* to spend your hours and your tokens while building it. If you'd rather read only one of the two? Read the assignment spec. If you read both? Read the spec first and then this document. Everything here assumes you have. Where the two seem to disagree, the spec wins.

One difference from last time is worth naming up front. The Assignment 1 companion started from an empty folder and a fresh install, because most of you were meeting C and Claude Code in the same week. That is over. This document assumes you have a working tree, a green battery, a `CLAUDE.md` you wrote yourself, and at least one evening of experience arguing with an agent about a pointer. So the C on-ramp is gone and the fixup walkthrough is gone, and what replaces them is the part that is genuinely new: three mutations that take away, one at a time, three conveniences your tree has been quietly leaning on since the day you wrote it.

Contents

1 What This Document Is and Is Not

The spec defines the destination. The description here is one way of walking there without blisters. You could ignore this companion entirely and still earn a full score on the assignment; the tools that grade you don't care how you got to a clean valgrind report. But the students who land at the top of the transcript rubric mostly work the way this document describes, whether or not anyone told them to.

So treat what follows as a strong recommendation, not a decree. It is organized the way the work for the assignment is: you read, then you plan, then you build across three weeks, and while you build you commit, you test, you debug, and you write things down. Each section leans on the ones before it. The token discipline in Section 11 is mostly a summary of habits you will have already picked up by then for other reasons. That is not an accident. Almost everything that makes you a good engineer here also makes you a frugal one.

If you read the Assignment 1 companion, large parts of this one will look familiar, and deliberately so. The habits did not change because the assignment got harder. They got more valuable.

2 Glossary: The Words I Keep Throwing Around

Some of what follows is *jargon* you have met before. Some of it is jargon this document invented for its own convenience. And some of it is the kind of shorthand that everybody in the computer science field uses but nobody ever defines out loud, which is a genuinely annoying way to learn a field.

So here is the dictionary (I have put them in gray boxes). Skim it now. Come back when a phrase in a later section makes you squint. Entries that carried over from the Assignment 1 companion are here unchanged, so you can skim those fast; the new words are the ones about allocation, slabs, and stacks.

Words about tests and tools

green: Everything passed. No failures, no leaks, no sanitizer complaints. The word comes from test runners that print failures in red and successes in green. So "commit at green" means: commit when nothing is broken. It is a state you can check, not a feeling you can have at 1 am.

red: The other one. Something failed. Stop and read what it said before you touch anything.

the battery: The full set of checks, run back to back: `make test`, then `make asan`, then `make memcheck`. Borrowed from "a battery of tests," which is old testing jargon rather than electricity. Nothing is charging. "Battery green" means all of it came back clean.

regression suite: Tests that exist to prove you did not break something that used to work. Your entire Assignment 1 suite is now this, and it is not negotiable: if a mutation breaks a test you wrote three weeks ago, the mutation is wrong.

table-driven test: One array of cases and one loop that runs them. Each row is an input plus what should happen. Adding a case means adding a row, not writing a whole new function. "The table is green" means every row passed, including the mean ones you wrote on purpose.

fault injection: Making something fail on purpose so you can watch what your code does about it. Here it is `rb_malloc` returning `NULL` on the n-th call. Real kernels ship this facility for the same reason: a failure that happens once every six months is a terrible test plan.

the sweep: Short for the fault-injection sweep in Mutation 1. Run one fixed scenario over and over: fail the first allocation, then the second, then the third, and keep going until you run out of allocations to fail. "Sweep green" means every one of those runs unwound without leaking a byte.

repro: Short for reproduction. The exact recipe that makes the bug appear on demand: same seed, same operations, same failure, every single run. Without one, you do not have a bug report. You have a rumor. In this assignment the sweep hands you repros for free: the failing n *is* the recipe.

shrink (or minimize): Cutting a repro down until there is almost nothing left of it. Ten thousand operations become twelve. The bug survives, the noise does not. This is probably the highest-value four minutes in all of debugging.

seed: The number you hand the random generator so it produces the same "random" sequence twice. Fixed seeds are what turn a fuzzer from a slot machine into a test, and an unfixed one is what turns your sweep into a coin toss.

fuzzer: A driver that hammers your tree with piles of random operations and checks that the answers are still right.

Words about tests and tools

reference model: A second, dumber implementation that you trust completely. A sorted array is perfect for this. You perform the same operations on both and compare. Sometimes called an *oracle*, because it knows the truth and your tree is the one on trial.

harness: The scaffolding around the thing being tested. It sets up the scenario, runs it, checks the outcome, cleans up after itself. Your fault sweep runs inside a harness you write.

scenario: The fixed sequence of tree operations the sweep re-runs at every n. Deterministic, in one place, called from both the plain and the pooled build. If yours quietly calls `rand()` without a fixed seed, it is not a scenario. It is weather.

Words about the code

slice: One small piece of work you can finish, test, and commit in a single sitting. Not a feature. Not a milestone. "The insert path survives the sweep" is a slice.

diff: The lines that changed. VS Code's Source Control panel shows it to you, colored and staged, before you commit. Read it. Every time. Test files especially.

seam: A boundary you can swap something in behind without the code above ever noticing. `rb_malloc` is the seam in this assignment, and this is the assignment where it earns its keep. The tree calls it and asks no questions. Production, the fault injector, and the pool take turns sitting underneath.

ownership: Who is responsible for eventually freeing an allocation. Several pointers can reach an object; exactly one owner frees it. Confuse reachability with ownership and you get a double free; forget an owner entirely and you get a leak. Mutation 1 is thirty evenings of this idea compressed into one week.

unwind: Undo the partial work when something fails halfway through. The second allocation failed, so the first one has to go back where it came from. Get this wrong and you leak on exactly the path nobody tests.

commit point: The instant after which the operation cannot fail. Everything before it is preparation and may be abandoned; everything after it must complete. The whole trick of Mutation 1 is moving your allocations to the left of this line so that there is nothing interesting to unwind.

failure atomicity: The property that an operation either completes or leaves the structure exactly as it found it. No half-built states, ever observable. You are meeting a very large idea in a very small place: two `malloc` calls and a tree.

slab: One wholesale chunk of memory, here 4096 bytes, carved into many same-sized slots. The slab is the storage. The allocator is the policy that hands the slots out.

intrusive free list: A linked list of dead objects whose "next" pointers live *inside the dead objects themselves.* No side table, no extra allocation, one store to free and one load to allocate. It is the cleverest four lines in the assignment and the most alignment-sensitive.

slack: The bytes at the tail of a slab too few to hold another whole object. They exist, they are yours, and they must never be handed out. Everybody's first carving loop hands them out.

Words about the code

poisoning: Deliberately scribbling `0xDD` over a freed object in debug builds, so anybody still holding a stale pointer reads obvious garbage instead of plausible-looking data. It is a booby trap you set for your own bugs, and it exists because once the pool owns the slab, ASan can no longer tell a dead slot from a live one.

invariant: Something that is true every time the loop reaches the top. Not usually true. Always. State it and you can usually show the loop must end. Fail to state it and you have learned something uncomfortable about your own code. `live + free_objs == slabs * objs_per_slab` is an invariant. Assert it constantly.

measure: The nonnegative quantity a loop strictly decreases every iteration. It is how "this obviously terminates" becomes "this terminates, and here is why." Mutation 3 will ask you for one out loud.

spine: A tree that has been rotated until it is a single descending chain with no left children. Teardown by flattening a tree into its right spine is one of the two techniques the spec points you at, and it needs exactly one pointer of scratch space.

Morris traversal: An in-order walk that borrows the tree's own right pointers to remember where to come back to, then puts them back. $O(1)$ auxiliary space, and a structure that is temporarily telling lies while the walk is in progress.

teardown: Destroying the whole structure and handing every byte back. Different from deleting one key, and much easier to get subtly wrong, especially when it is not allowed to allocate, recurse, or fail.

refcount: A per-node count of how many handles currently share it, used only in the Reach. Raise it when a handle arrives, lower it when one leaves, free at zero. Get the decrement order wrong and you have written a kernel CVE in miniature.

stub: A function that exists and compiles and does nothing useful yet. An honest placeholder, not a lie.

spelunking: What the agent does when you do not tell it where to look. It wanders off into your repo with a headlamp, opening files, reading tests, billing you for the tour.

Words about git

commit: A saved snapshot of your repo with a message attached. Cheap, fast, and boring. Do it constantly.

commit at green: Only save snapshots that actually work. Your history then becomes a list of states you can safely return to, which is the entire reason for having a history in the first place.

`git restore`: Throw away your uncommitted changes and snap back to the last commit. Instant and free. Vastly better than paying the agent to un-write code it wrote forty minutes ago.

`git bisect`: Binary search through your history for the commit that broke things. Magnificent when your commits are small. Completely useless when your history is one commit called "final."

clean clone: Cloning your own repo into an empty directory and building from that. It answers one question, and it is a question worth asking before you submit: is the project I am handing in the same project I have been building, or does it only compile because of some untracked file sitting in my working directory?

Words about the agent and the meter

token: The chunk of text the model actually reads. Somewhere between a syllable and a short word. Call it 4 characters of English prose, and closer to 3 for C source, because your punctuation counts.

context: Everything the model can see on this turn: your prompt, your `CLAUDE.md`, every file it opened, the output of every command it ran, and all of its own earlier replies. It gets re-sent and re-read on every single message. So context is not a purchase. It is a subscription.

context window: The size of the container. How much context fits at once.

usage limit: Your budget, which is a completely different thing. You can be nowhere near filling the window and still be out of allowance for the week.

session: One continuous conversation, from your first message until you `/clear` it.

`/clear`: Wipes the conversation and starts clean. Your `CLAUDE.md` survives, so your project rules survive with it.

`/compact`: Summarizes what came before and carries on. You can steer what lives through it: `/compact keep the failing test output and the current diff`.

plan mode: Shift+Tab until the status line says so. The agent can read and analyze but cannot touch a file. This is where arguments are cheap and mistakes cost nothing.

`CLAUDE.md`: Your project's standing law, loaded at the start of every session. Which makes every line in it a recurring charge, so keep it lean and keep it about the project.

slash command: A saved prompt you fire with one keystroke, like the `/battery` loop. It lives as a file in `.claude/commands/`.

adversarial review: Asking a fresh session to attack code it did not write. A reviewer who shares the author's assumptions also inherits the author's blind spots, and the author here is sometimes you.

devlog: `DEVLOG.md`. Five minutes at the end of each session. It is memory that never once sends you a bill.

3 Please Read the Assignment

The spec is long. Read it the way you would read your housing lease. You are about to live in the spec for three weeks.

3.1 Pass one: read for shape, hands off the keyboard

Sit somewhere without a terminal and read the whole thing. Start to finish in one sitting. Budget 45 minutes. Do not open VS Code. Do not open Claude. The urge to start typing somewhere around the workflow section will be strong, and resisting it is the first test of the semester.

You are reading for shape on this pass: what the milestones are, what order they come in, what gets graded by machine and what gets graded by a human looking you in the eye. You are also reading for tone, because the spec telegraphs where the danger is. When a document pauses for three paragraphs to explain stack frames before asking you to tear down a tree, that is not padding. That is a flare over a minefield. The same goes for the figure of the insert path with a dashed line through it, and for the two places where the spec quietly tells you that an operation should not allocate at all.

You have one advantage this time that you did not have last time: you already know what the tree does. So read this spec asking a different question than you asked the last one. Not "how does the algorithm work," which you have settled, but "which of my current assumptions is this paragraph about to revoke."

3.2 Pass two: read with a pen, extract into notes

Now read it again (slower) with `NOTES.md` open in VS Code (if you kept the one from Assignment 1, append; it is yours, not a deliverable). This pass is extraction. You are pulling four kinds of things out of the prose, and every one of them gets the spec's section name written next to it so you can jump back later:

Extract	What it looks like	Where it ends up
Contracts	The header is frozen and has grown two declarations. All allocation goes through `rb_malloc`/`rb_free`. A failed insert leaves the tree unchanged *and* does not consume the caller's value. `pool_destroy` frees every slab regardless of live objects.	Lines in `CLAUDE.md`; the rules the agent must never break.
Thresholds	4096-byte slabs. $O(1)$ for `pool_alloc` and `pool_free`. 10^6 nodes under a 64 KB stack. `live + free_objs == slabs * objs_per_slab`. Zero leaks, not few.	Assertions in your tests. Every threshold is a test waiting to be written.
Choices left to you	Where the slab header lives. What alignment you guarantee and how the stride enforces it. Right-spine teardown or pointer reversal? Morris or parent pointers? What your `rb_foreach` contract says about callbacks that call back in?	Plan-mode discussions with Claude, and one-line decision records in your devlog (Section 12).
Confusions	"Why does pool reuse hide use-after-free from ASan?" "What exactly happens between the node alloc and the key copy?" "Why can't the teardown loop spin forever?"	Your first Claude session: exploration questions, before any code exists.

Notice what just happened? Your notes *are* the plan. The contracts become your `CLAUDE.md` amendments. The thresholds become your test list. The choices become the agenda for plan mode, and, not coincidentally, three of the four things `REFLECTION.md` asks you to defend. The confusions become your opening conversation. When Section 6 tells you to "plan Mutation 1," you will not be staring at a blank prompt box wondering what to say; you will be reading from a page you already wrote.

3.3 Revisit; don't rely on memory

Nobody holds a spec this size in their head for three weeks, and nobody should try. The spec is a reference manual, and reference manuals are for referencing. Build the habit now: before each milestone, reread the matching part of the spec even if you are sure you remember it. Especially if you are sure.

Before you start...	...reread, in the spec
M0 (setup)	Part-0 Setup and all three appendices. Diff the new frozen header against your Assignment 1 copy line by line; there are two new declarations and two amended comments, and you should be able to say what each one costs you.
M1 (fault sweep)	Mutation 1 in full, including the magenta termination guarantee, the "exactly as it was is a testable claim" paragraph, and the unwind figure with the dashed line. The three bullets under "where the design work actually is" are the whole milestone in miniature.
M2 (pool)	Mutation 2, twice: once for the API and the constraints, once for the three decisions the API leaves vague and the poisoning paragraph. Look at the slab figure long enough to see both layers, the physical slots and the free list that ignores them.
M3 (teardown)	Mutation 3, twice. It contains the loop invariant you will defend out loud at the walkthrough, and the `const` wrinkle in `rb_foreach` that ASan will otherwise explain to you personally.
Every session	Skim your own notes from pass two. Thirty seconds. Cheaper than rediscovery.

And when something surprises you mid-implementation, the reflex is: spec first, notes second, Claude third, and a test to settle whatever survives all three. The order matters. Claude will cheerfully answer a question the spec already settled and you would have paid tokens for an answer that you owned in print.

4 The Three Mutations, One Idea at a Time

Here is the unusual thing the spec does, and it deserves to be used rather than admired. It does not merely list three requirements. It tells you, for each one, what property of your existing tree it is attacking and what systems idea the attack is teaching. Mutation 1 removes the assumption that allocation succeeds. Mutation 2 removes the assumption that nodes come from `malloc`. Mutation 3 removes the assumption that there is a call stack to walk on. The tree computes exactly the

same thing at the end as at the beginning, which is the point, and is also why it is so easy to underestimate how much work is involved.

This section is about extracting the value from those three ideas before you write code, because each of them has one insight that makes the implementation half as hard, and all three of those insights are things the walkthrough will ask you to say out loud.

4.1 Constraints accumulate; nothing gets repealed

Read the milestones as a stack, not a queue. Mutation 2 does not retire Mutation 1: your pool calls `rb_malloc` for its slabs, that call can fail, and the sweep now has to survive a pooled build too. Mutation 3 does not retire either one: your teardown has to work on plain and pooled trees alike, and it is not allowed to allocate its way out of trouble. And underneath all of it, your entire Assignment 1 suite is still running.

There is a scheduling consequence and you should plan for it now. Every milestone ends by re-running everything that came before, and the reruns are where the surprises live. If you budget only for writing new code, week two will feel like it went badly when in fact it went normally. Put the rerun in the plan.

4.2 Mutation 1: design the failure paths out of existence

The spec's figure of `rb_insert` with a dashed line through it is the single most useful picture in the assignment, and the lesson under it is worth restating in the bluntest form available: **the cheapest way to survive the sweep is to make most of the failure paths impossible rather than correct**.

So do this before you prompt anything, with a pen, on a printout or a whiteboard. Open your own `rb_insert` and mark two things: every call that can fail, and the first line that modifies the tree. If any of the failure points sits after that line, you have work to do, and the work is not "write better cleanup code." It is "move the allocation earlier." Do the same for `rb_delete` and for the overwrite branch. When you are finished you should be able to say, in one sentence per function, where the commit point is.

That exercise takes ten minutes and it is the whole plan. When you then go into plan mode and ask Claude for the unwind analysis, you are not asking it to design the milestone; you are checking its picture against yours, which is what Step 2 of the spec's workflow has been telling you to do since Assignment 1.

Three practical notes the spec implies and does not spell out:

- **Write the "unchanged" assertion once, as a helper.** The spec enumerates what "exactly as it was" means: `rb_validate` passes, `rb_size` is what it was, `rb_find` of the failed key returns what it returned, every model key is still findable with its value, and the caller's value was not freed. That is five checks. Write them as one function that takes a before-snapshot and call it after every failed operation. Otherwise you will write four of the five and lose the fifth in a copy-paste at 11 pm.
- **Prove the value survived with a counting destructor.** Ownership is invisible to `rb_validate`, so give yourself eyes: a `value_free` that increments a global counter, values allocated with a

known pattern, and an assertion that the counter did not move across a failed insert. This is the check our scenario makes, and it is the one students most often discover at grading time rather than at 9 pm on a Tuesday.

- **Make n mean something.** When the sweep fails at $n = 61$, your first question is "which allocation is 61?" Answer it cheaply: have the injector, in debug builds, print or record a one-line label for each allocation it counts. A failing n then arrives already localized, and the prompt you send the agent says "$n = 61$ is the slab allocation inside `pool_create`" instead of "the sweep fails at 61." That difference is worth about forty minutes and a lot of tokens.

And when the sweep appears not to terminate, reread the magenta paragraph before you reread your harness. The guarantee is airtight: a fixed scenario makes finitely many allocations. Nondeterminism here is a message about your code, usually about uninitialized memory, and the spec says so precisely because everyone's first instinct is to suspect the harness.

4.3 Mutation 2: the slab is storage, the free list is policy

Look at the spec's slab figure twice, once along each axis. Horizontally, a slab is an array of identical slots and nothing more interesting than that. Vertically, the free list threads through whichever slots happen to be dead right now, in whatever order they died, cheerfully ignoring the physical layout. Two questions, two mechanisms: the slab answers *where is the storage*, the free list answers *which slot do I hand out next.* Almost every confusing pool bug comes from a student who has fused those two questions into one.

The three decisions the spec leaves to you (alignment, where the header lives, what `pool_stats` counts) are not trivia. They are the milestone. Settle all three in plan mode, on the first evening, before a line of `pool.c` exists, and write each one into your devlog as a single sentence with its reason attached. You will be asked about them, and "that's what the agent generated" is the wrong answer in a way that costs points twice, once on the quiz and once on the reflection.

Some advice for making them go smoothly:

- **Compute the geometry once, in one function, and test it directly.** Slots per slab, stride, header offset. Then write a test that takes the numbers the pool reports and checks `live + free_objs == slabs * objs_per_slab` after every scenario, and an assertion inside the pool that checks it constantly in debug builds. The earliest assertion failure is enormously more informative than the corpse you find ten thousand operations later.

- **Test the pool through its own API before you test it through the tree.** Allocate a thousand objects, free every third one, allocate five hundred more, check the invariant, destroy with objects live. That is twenty lines of test and it localizes bugs that a tree-level failure would smear across two files. It is also the cheapest possible input to a prompt: "`pool_alloc` returns an address inside the header on the second slab, here is the failing pool unit test" is a question with one obvious place to look.

- **Decide who owns the first bytes of a dead object.** The poison wants them and the free-list link wants them, and only one of them can have them. Poison the whole slot, then write the link over the front of it; now a stale reader of anything past the link sees `0xDD` and a stale reader of

the link sees a pointer that at least belongs to the pool. Write the ordering down as a comment. It is exactly the kind of two-line decision the walkthrough loves.

- **Remember that keys still come from `rb_malloc`.** "Pooled tree" does not mean "allocation-free tree." Nodes are fixed-size and go to the pool; key copies are variable-length and do not. Which means teardown still has to visit every node to release keys and values, which is precisely the setup for Mutation 3.

One sentence about why ASan stops helping you here, because it is the reflection question and it is worth getting right in your own words: ASan tracks the 4096-byte slab, which is still very much alive, and knows nothing about the sub-allocation boundaries you invented inside it. You have started subletting rooms the sanitizer believes you still occupy. Poisoning is how you make the sublet visible.

4.4 Mutation 3: find the quantity that decreases

Before you write the teardown, run the drill the Assignment 1 companion used on the fixup figures, because it works just as well here. Draw a seven-node tree on paper. Run the spec's loop by hand, one iteration at a time, and at each step write down two numbers: how many nodes are left, and how many nodes still hang to the left of the root. Watch which one is allowed to stay the same and which one is not. By the time you have done six iterations you will have discovered the measure yourself, and discovering it is worth ten times being told it.

Then write the invariant comment *before* the code, not after. Your own `CLAUDE.md` style rules already require one on every non-obvious loop. This is the loop those rules were written for.

Two things that bite people, both of which the spec warns about and both of which are worth repeating because they cost an evening each:

- **Save the pointer before you free the node holding it.** Panel (c) of the spec's teardown figure is where this happens: you free the root and then advance to its right child, and the address of that child had better already be in a local variable. A pointer kept "just for a second" does not receive a grace period from C.
- **Morris traversal deforms the tree and must put it back.** During the callback, the structure is temporarily lying, so decide what your contract says about a callback that calls `rb_find`, write the decision in a comment, and then *test the restoration*: walk the tree, collect the keys in order, run `rb_foreach`, walk again, and assert the two walks agree. A traversal that leaves one right pointer rewired is a traversal that passes its own test and destroys the next one.

And a scheduling note that the spec makes explicitly and that students still get wrong. Valgrind is many times slower than native, and a million-node teardown under valgrind is not a coffee break, it is a commute. Put the large case behind a flag: run it big under `asan`, run a smaller one under `memcheck`. What you must not do is quietly shrink the large case because the timing was inconvenient and then say nothing. If your "million-node" test runs at ten thousand nodes, that belongs in `REFLECTION.md`, and the walkthrough will ask.

Last, the honest caveat the spec offers and you should take to heart: a red-black tree of a million nodes is about forty levels deep, so passing the small-stack test does not by itself prove you removed recursion. It proves your tree is balanced. The proof that you removed recursion is the code, and the explanation of where your traversal state lives is the thing you will be asked for.

4.5 Keeping the three straight

	Mutation 1	Mutation 2	Mutation 3
What it takes away	Allocation always succeeds	Nodes come from `malloc`	There is a stack to recurse on
The one insight	Move the allocations left of the commit point and there is nothing to unwind	The slab is storage; the free list is policy; keep them separate	Borrow the tree's own pointers for bookkeeping
Where it bites	Between two allocations, on a path no ordinary test visits	Slab arithmetic: header bytes, tail slack, alignment	Ordering: the pointer you need is inside the node you just freed
What you must prove	Every failure exit leaves the tree and the caller's value untouched	`live + free_objs == slabs * objs_per_slab`, always	A nonnegative measure that strictly decreases every iteration
The walk-through question	"What frees this on the error path?"	"Why is this slot aligned, and what does `free_objs` count?"	"Why can't this loop run forever?"

If you can reproduce that table from memory, with the reasons attached, most of the quiz will feel like a formality. If you cannot, you have just found your study list.

4.6 And if you take the Reach

Mutation 4 carries no points, and the spec is right that it is the part you will be proudest to have gotten right. One warning that is easy to miss: it collides head-on with Mutation 3. A teardown that must decrement a reference count and stop descending wherever the count stays positive is a considerably more delicate loop than the one you just wrote, because it can no longer assume the nodes it is walking are its own to rearrange. So do it *after* Mutation 3 is green and committed, on a branch or behind a flag, and expect Claude to get it almost right on the first try. "Almost" is the lesson.

5 Plan the Modules Before the Code

Here is a truth that will follow you into every job you ever hold: humans maintain what they can hold in their head, and now, agents bill you for what *they* have to read. Those are the same problem at two different scales. Modularity is the answer to both. A 1,400-line `rbtree.c` is expensive for you to reason about and expensive for Claude to ingest, and you will be doing both, repeatedly, for three weeks.

5.1 The file-level map is given; the function-level map is yours

The spec fixes the file layout, so the coarse modularity is already decided for you. What is yours to design is the decomposition *inside* those files, and it is worth twenty minutes of thought before any code exists.

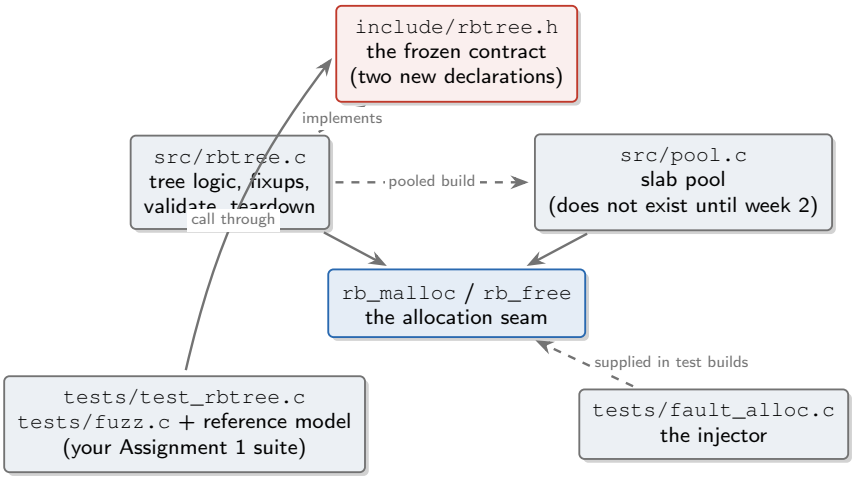

Figure 1: The module map. Every box is something you can point Claude at *by name*, and every arrow is a boundary that keeps a prompt small. The header is red because it is frozen. The seam is blue because it is the cleverest boundary in the assignment: production, the pool, and the fault injector all meet there, and none of them know about the others. If you took the tip at the end of Assignment 1 and installed the seam already, the whole left half of this picture is a week old and you may feel justified.

Inside `rbtree.c` and `pool.c`, aim for static helpers with one duty each. Something like this:

Internal module	Its one job
`node_alloc`, `node_release`	The *entire* multi-allocation dance (node, then key copy) and its unwinding, in one place. If ownership is ever confusing, it is confusing in exactly one function. This is the function Mutation 1 is really about.
`rotate_left`, `rotate_right`	The pointer surgery, unchanged from Assignment 1, still owning the `t->root` update. Mutation 3 borrows one of them for a purpose it was never designed for.
`insert_fixup`, `delete_fixup`, `transplant`	Your existing machinery. It allocates nothing, which is exactly why the commit point can sit in front of it. Do not let anyone teach it to allocate.
`pool_geometry` (or whatever you call it)	Slots per slab, stride, header offset, alignment. One function, one set of numbers, directly unit-tested. Every off-by-one in this milestone lives here.
`freelist_push`, `freelist_pop`	Two stores and two loads, plus the poison in debug builds. Kept apart from the carving code so the "which bytes does the link own" question has one address.
teardown / traversal core	The M3 loops isolated with their invariant comments on top, ready to be defended at the walkthrough.

Why does this granularity matter so much here? Because a prompt like "`freelist_pop` in `src/pool.c` returns an address four bytes inside the slab header on the second slab; here is the failing geometry test" costs Claude one function's worth of reading. "Something is wrong with the pool" costs it the file, and "fix my tree" costs it your repository, your dignity, your Tuesday, and a measurable slice of the weekly cap.

Put the map in writing. A short "Code map" block in your `CLAUDE.md` (five or six lines: what lives where) means the agent stops spelunking through your tests to find your carving loop. You pay for that block once per session. You save it back the first time the agent *doesn't* read `fuzz.c` on its way to an alignment bug.

5.2 You don't need all of it on day one

Look again at Figure 1. `pool.c` doesn't exist in week 1, and this is not a gap to feel guilty about; it is a schedule working as intended. The seam is what makes this safe: because every allocation already goes through `rb_malloc`, the pool can arrive a week late and slot in underneath without the tree noticing. The same goes for `rb_snapshot`. Mutation 4 is ungraded, so until the day you attempt it (if you attempt it), leaving it undefined is a perfectly honest state; the declaration in the frozen header costs you nothing as long as nothing calls it.

Build the smallest thing that lets the next test pass. Then the next. Remember the module map is a destination, not a birth certificate!

5.3 The design will wiggle. That is fine.

Your node struct will probably not survive the assignment unchanged. Mutation 3 may talk you into parent pointers, and if it does, the memory cost belongs in `pool_stats` and in your reflection. The Reach would add a refcount, and a refcount changes what `node_release` means. The teardown you wrote in Assignment 1 gets replaced wholesale in week three, on purpose, by the spec's own design. So hold your design decisions firmly but not permanently, and notice that small modules are exactly what make revision cheap: when the struct grows a field, `node_alloc` changes, `pool_geometry` changes, and almost nothing else does. A design that wiggles inside clean boundaries is called iteration. A design that wiggles without them is called a rewrite, and rewrites are billed at full price: in tokens and in evenings.

6 The Three Weeks: No Software Is Developed in a Day

The spec estimates 12–16 hours. That number is honest. But it hides the part that matters: those hours behave completely differently depending on *how* they are distributed. Sixteen hours across nine unhurried evenings is a comfortable project. Sixteen hours starting Wednesday night is a disaster with a countdown timer, and the weekly token cap makes it arithmetic rather than opinion: three calendar weeks means three weekly allowances. The night before the deadline means one, shared with your panic.

So here is what three weeks actually looks like, evening by evening. Each evening is 90 minutes to two hours. Each one ends with a green state, a commit, five minutes of devlog (Section 12), and a closed laptop. None of them ends with "I'll just get this one thing working," because that sentence is how 11 pm becomes 2 am.

6.1 Week 1: Carry it forward, then make allocation fail (M0 + M1)

Evening 1: the new header, the seam, and a gentle session. Open the Assignment 1 repository (or clone it into a fresh one; either is fine, but the commit count is counted for *this* assignment, so a fresh repo that opens with one enormous import commit had better be followed by ten real ones). Drop in the new frozen header from Appendix A and diff it against your old copy so you can see exactly what grew. Build. Your Assignment 1 suite should still be green against the new header, unmodified, and if it is not, stop and find out why before anything else happens tonight. Commit. That is commit #1 and you have written no new logic. Good.

Then the seam: `rb_malloc` and `rb_free`, four lines each, and every allocation in `src/` routed through them. If you took the tip at the end of Assignment 1, this is already done and you may skip to the next paragraph feeling smug. If not, this is a mechanical change and a perfect first slice: make it, run the battery, commit.

Now open `claude`. Carry your `CLAUDE.md` forward and *amend* it rather than starting over, because the diff itself is evidence of how your constraints evolved, and the graders read that diff. Add the seam rule, the "any allocation may fail" rule, and the code map. Commit it. Then spend the rest of the evening on the spec's Step 2: exploration, zero edits, working from your confusions list.

> *Read src/rbtree.c and include/rbtree.h. Do not edit anything. For rb_ insert, list every call that can fail and the first line that modifies the tree, in source order. Then tell me, for each failure point that sits after that line, what would have to be undone. I have my own answer written down; I want to compare.*

Take notes on the answers. Argue with anything that smells off. End the session with `/clear`. You have now used the seam, an amended `CLAUDE.md`, and context hygiene, on day one, at nearly zero cost.

Evening 2: the sweep harness, and your first real argument. Reread Mutation 1, including the magenta termination guarantee. Fresh session. Plan mode, and this time the plan is the one the spec scripts for you: every allocation site in `rb_insert`, what unwinds on each failure, the goto-cleanup structure, the harness design. **Argue with the plan.** Ask what breaks if the key copy succeeds but the node allocation fails, or the reverse. Ask why the overwrite path allocates at all, and whether it needs to. A first plan accepted unmodified is a red flag the graders look for, but more to the point, first plans are usually wrong somewhere, and finding where is the entire skill. Record the revision in `PROMPTLOG.md` tonight, while you still remember why you pushed back.

Then, out of plan mode: harness first and failing, the way Step 4 tells you. The injector, the fixed scenario in one place, the loop over n, the "unchanged" helper, the counting destructor. Then the insert path, smallest diff, until the sweep is green for insert. Commit.

Evening 3: the rest of the sweep. Delete and overwrite paths. The overwrite path is sneakier than it looks; the header comments say exactly who owns what, and your Assignment 1 exploration notes already summarize them. This is also where you may discover that the honest fix is to delete an allocation rather than to handle its failure, which is a very satisfying diff to write. Full sweep green under both `asan` and `memcheck`, with the Assignment 1 battery still green beside it. Commit. `/clear`. If you are feeling ambitious, spend ten minutes creating the `/battery` slash command from the spec's Step 9; with three suites to re-run at every milestone, it will pay for itself daily from here on. Last chore: copy this week's `.jsonl` transcripts out of `~/.claude/projects/` into a safe place. The 30-day purge is real (the spec warned you) and "the tool ate my homework" has never once worked on a grader.

6.2 Week 2: The pool (M2)

Evening 4: geometry before code. Reread Mutation 2. Plan mode, and bring the sharp questions to the planning table yourself rather than waiting for the agent to raise them: where does the intrusive free-list pointer live inside a dead object, and what guarantees its alignment? In-band header or out-of-band, and what does each cost in failure paths? What exactly does `free_objs` count? How does `pool_destroy` free slabs while objects are live? Settle all of it, write the three decisions into your devlog with reasons, and then implement only the geometry: one function that computes stride, slots, and offsets, with a unit test that checks the arithmetic directly. Commit.

Evening 5: carving, the free list, and poisoning. `pool_create`, `pool_alloc`, `pool_free`, `pool_stats`, `pool_destroy`, in slices, tested through the pool's own API before the tree ever sees it. Add the debug poisoning and decide, in a comment, who owns the first bytes of a dead object. Assert the `live + free_objs` invariant after every scenario. Commit at each green rung.

Evening 6: the moment of truth. `rb_create_pooled`, and then the thing the spec builds toward: rerun the *entire* Assignment 1 + Mutation 1 battery on the pooled build, sweep included, because slab allocation can fail too and `rb_create_pooled` now has its own unwind path. Expect the sweep to find something here; that is what it is for. When it is green, commit, and notice the quiet satisfaction of the thing the spec promised: the tree never noticed.

6.3 Week 3: Teardown, hardening, and the paperwork (M3)

Evening 7: the teardown. Reread Mutation 3. Twice. It tells you, in so many words, that its loop invariant is a walkthrough question. Run the paper drill from Section 4 before you prompt. Plan mode: choose right-spine rotation or pointer reversal, and write down *why* in one sentence in your devlog, because "why did you choose this" is also a walkthrough question. Implement. Then the test the spec specifies: a million nodes, torn down inside a `pthread` with a 64 KB stack, no crash, zero leaks, and the big case behind a flag so `make memcheck` still finishes inside a coffee break. Commit. And before you close the laptop, make Claude do the thing you will later do live:

> *State the invariant the teardown loop in rb_ destroy maintains at the top of each iteration, and name the nonnegative quantity that strictly decreases every iteration. Show why termination follows. If you cannot state one, the code is wrong; say so.*

If the answer does not convince you, the walkthrough version of you, three weeks from now, will not convince anyone either.

Evening 8: then the hostile reviewer. The traversal under the same $O(1)$ constraints, plus the restoration test: keys in order before, `rb_foreach`, keys in order after, assert they agree. Decide and document what your contract says about a callback that reaches back into the tree. Then the spec's Step 7: `/clear` into a fresh context and run the adversarial review, hunting the exact bug families the spec names: use-after-free enabled by pool reuse, off-by-one in slab carving, alignment bugs for the intrusive free-list pointer, paths where `pool_destroy` is called with live objects. Triage the findings yourself. One real finding and one false positive go into `PROMPTLOG.md` with your reasoning, and doing it tonight takes five minutes; doing it Sunday from memory takes an hour and reads like fiction.

Evening 9: the paperwork, and the buffer. Finish annotating `PROMPTLOG.md` (6–10 episodes; your devlog makes this transcription, not archaeology), and check the required four are present: the plan you revised, the diff you rejected, the debugging loop with real tool output, the review finding you triaged. Write `REFLECTION.md`; half of it already exists as devlog entries, and the spec is specific about what it wants: the slab comparison with `kmem_cache`, your `pool_stats` definitions, your slab-header and alignment decisions, your poisoning rationale, where the agent was most and least reliable, and one bug it introduced that you caught. Copy the remaining `.jsonl` transcripts in as-is. Check the git log: well over 10 commits for this assignment, each one telling a small true story. Build from a clean clone of your own repo to make sure the tar you submit is the project you think it is. Then stop. If the evening ends early and you are still having fun? The Reach is sitting right there, and the spec is correct that you will be proud of it. But the buffer comes first. The buffer is load-bearing.

Add it up: nine evenings, none heroic, and the estimate lands inside the spec's 12–16 hours with slack to spare. That slack is not free time you failed to use. It is the difference between a schedule and a gamble.

7 Commit Wisely. Commit Often

The spec requires 10 meaningful commits for this assignment and promises a process-grade of zero for a single "final submission" commit. Treat that as a floor with a trapdoor under it, not a target. Working the way this document describes produces 25 commits without trying, because a commit is not a ceremony. It is a checkpoint of a green state. And green states happen constantly when your slices are small.

What "wisely" means, concretely:

- One logical change per commit. "M2: pool survives the fault sweep" is one story. "M2: various fixes" is a shrug with a hash.
- Commit only from green. The relevant battery for the slice passes, and this assignment has three of them stacked. Your history then becomes a sequence of known-good states, which is the entire point of having one.
- Commit *before* anything risky. About to let Claude restructure the teardown? Commit first. If the restructuring goes sideways, `git restore` puts it back in two seconds, and it is the only undo button in this workflow that does not bill you. Paying tokens to have Claude un-write code it wrote an hour ago is the saddest line item in this course.
- Read the diff before you bless it. VS Code's Source Control panel shows you exactly what changed, staged and colored. Reading it there, before committing, is where you catch the agent's "helpful" edit to a test. Every diff. Especially test diffs, and especially the Assignment 1 tests, which are now regression tests and are the single most tempting thing in the repository for an agent to soften.
- Let Claude drive git, conversationally, after you have read. "Run the full battery; if clean, commit as 'M2: pool survives fault sweep'." That is the spec's Step 8, and it is a fine habit, with one non-negotiable clause: the diff passed your eyes first.

A healthy history for this assignment reads like a lab notebook that happens to compile:

```
M0: import A1 tree; new frozen header; A1 battery still green
M0: rb_malloc/rb_free seam; all allocation in src/ routed through it
M0: CLAUDE.md amended for A2 (seam rule, failure rule, code map)
M1: fault_alloc injector + sweep harness (failing, on purpose)
M1: assert_unchanged helper + counting destructor for value ownership
M1: insert allocates before the commit point; sweep green n<=44
M1: overwrite path allocates nothing; delete path allocates nothing
M1: full sweep green, asan + memcheck, A1 battery still green
M2: pool geometry (stride, slots, header offset) + unit test
M2: slab carving + intrusive free list, O(1) alloc/free
```

```
M2: 0xDD poisoning in debug; link written over the front of the poison
M2: rb_create_pooled unwinds on slab failure; sweep green on pooled build
M3: right-spine teardown, invariant comment, 1e6 nodes under 64KB
M3: Morris rb_foreach, O(1) space, restoration test
M3: post-review fixes from adversarial pass (2 real, 1 rejected)
docs: PROMPTLOG episodes 1-8 annotated, REFLECTION drafted
```

Notice what this buys you beyond the grade. When Thursday's "fix" regresses the sweep, `git bisect` finds the guilty commit in minutes *because the commits are small*. Your history is also your best answer at the walkthrough: asked how a piece of code came to be, you can literally show its biography! And it is corroboration for the transcript rubric, which explicitly scores claims against the repository. The grader is checking whether your story and your history agree. Make them agree by having them be the same thing.

8 Test in Increments, Not in Ceremonies

Testing on this assignment is a ladder, and you climb it one rung per slice. Never all at once at the end. The end is where testing goes to become debugging.

1. **The Assignment 1 suite, constantly.** It is a regression suite now. Run it after every slice, not as a formality but because the whole premise of this assignment is that the tree's behavior does not change while its foundations do. A mutation that breaks a three-week-old test has told you something important, and the test is not the thing that needs adjusting.

2. `rb_validate` after every mutation inside your tests, exactly as before. It is the cheapest oracle you own. It is also, note, no help at all in the middle of a teardown, because a tree being destroyed is allowed to stop being a legal red-black tree; that is what makes the teardown loop's invariant your own responsibility.

3. **The sweep, early and small.** Do not wait until the whole insert path is perfect. Arm the injector, run a three-operation scenario, watch it fail, fix one path. The sweep's great virtue is that it manufactures the rare states on purpose, which is exactly what random fuzzing is bad at.

4. **Pool unit tests before pooled tree tests.** Twenty lines that allocate, free, and check the invariant will localize a carving bug that a tree-level failure would smear across two files.

5. **The rerun.** Each mutation's test lands with the mutation, and each rerun of the older battery is the proof that the new ground did not shift the old building. The pooled rerun of the sweep is the single most informative test in this assignment. Budget an evening for it, not ten minutes.

Cadence matters as much as coverage. `make test` runs in seconds; run it constantly, after every slice. Like breathing. `make asan` after each meaningful change, and it is your friend for the big cases because it is fast enough to run them. `make memcheck` once per evening, as the closing ritual, on the smaller cases; valgrind is slow, and that is fine. Let it be slow at the end of the evening, like dessert!

Two standing rules, both borrowed from the spec because they are that important. First: never accept "the tests should pass now." The agent runs the commands and shows output (every time) and you paste the interesting lines back when they are not clean. Second: watch for tests getting quietly weaker. An assertion that loosens, a case that vanishes, a threshold that drifts from 10^6 nodes to 10^4 because the run was slow: these are real failure modes of real agents, and your `CLAUDE.md` forbids them in writing. Your eyes on every test diff are the enforcement mechanism. The tests are the contract you hold the code to. Guard them like one.

9 Fixing Bugs Without Burning Down the House

There is a prompt you will be tempted to write around week two, at roughly 10:40 pm. It goes: "here is my whole codebase, something leaks, fix it." Don't write it!!! It fails on both axes at once. It is expensive, because the agent must read everything you have ever written to answer anything at all, and then you carry that entire reading forward in context for the rest of the session. And it is ineffective because an agent handed everything and told nothing will guess! And its guesses arrive as a confident multi-file diff that you now have to review, which was the work you were trying to avoid, except larger.

Debugging with an agent is the same loop debugging has always been; the agent just makes each step faster *if you do the steps in order*:

1. Reproduce deterministically. Same seed, same scenario, same failure, every run. If it will not reproduce, that fact *is* the bug report (the magenta paragraph in Mutation 1 tells you what nondeterminism usually means: your code, not the harness). This assignment is unusually generous here: a failing n in the sweep is a repro you were handed for free.

2. Minimize. Shrink the failing input until it is embarrassing. Twelve operations. Four. The smaller the repro, the sharper everything downstream. And name the allocation: "$n = 61$" is a number, "$n = 61$ is the slab allocation inside `pool_create`" is a location.

3. Localize with the tools you already run. ASan and valgrind hand you file and line and an allocation backtrace. That is not decoration; that is the address of the crime scene.

4. Prompt with the artifact, scoped to the module. Name the file. Name the function. Paste the twenty lines that matter, not the two thousand that say `PASS`. `grep -A6 'definitely lost'` is free; context is not.

5. Smallest diff, verify with the same tool, commit, log it. The episode goes in `PROMPTLOG.md` while it is warm, because tool-output debugging loops are a required episode and tonight's version of you writes it in five honest minutes.

   Which turns the 10:40 pm prompt into this instead:

   > *make memcheck on the fault sweep reports: "40 bytes definitely lost, allocated at rbtree.c:212 in node_alloc, during sweep n=7". n=7 is the key-copy allocation inside an overwrite of an existing key. Read only node_alloc and rb_insert in src/rbtree.c. Trace ownership of that allocation through the overwrite path when the fault fires, propose the smallest fix to that path only, then re-run make memcheck and show me the sweep summary.*

Same bug. A fraction of the reading. And an answer you can actually review, because it is the size of the question.

Two more habits for the bad nights. Press Escape the moment a response heads somewhere you did not intend. It is the cheapest key on your keyboard and it depreciates by the second. And when a long debugging thread finally lands but has left three dead theories in its wake, `/compact keep the failing test output and the current diff`, so the session stops paying rent on furniture it no longer owns.

10 Tokens, Limits, and the Meter You Cannot See

You met this in Assignment 1, and some of you met it the hard way, on a Sunday, staring at a reset timer. The short version is repeated here because the numbers get worse this time: you now have two source files, three test suites, and a spec that assumes a prior spec. The opportunities to be profligate have multiplied.

10.1 What a token actually is

A language model does not read letters. And it doesn't quite read words either. It reads *tokens*: chunks of text carved up by a tokenizer, usually somewhere between a syllable and a short word. "Tree" is one token. "Rebalancing" is probably three. A space, a newline, and a semicolon each cost you something!

For ordinary English prose, the rule of thumb is about 4 characters per token, so a thousand tokens is roughly 750 words. Call it a page and a half double-spaced.

And code is worse. And this surprises people. C source is dense with punctuation and with identifiers that no tokenizer has ever seen, so `rb_insert_fixup` lands as 4-5 tokens rather than one, and every level of indentation is billed by the character. Reckon on 3 characters per token for source, which is a polite way of saying your files are *bigger* than they look.

Thing	Rough size	In tokens, about
`include/rbtree.h` with its comments	150 lines	1,500
A finished `src/rbtree.c`	1,100 lines	15,000
A finished `src/pool.c`	250 lines	3,500
This assignment's PDF	35 pages	35,000
Both specs, because this one builds on the last	65+ pages	65,000+
A full sweep run, unfiltered	one line per n, times a few hundred	20,000
The one valgrind line you cared about	1 line	30

Look at the last two rows for a second. That ratio is the entire lesson of this section. Everything below is just elaboration with a plot.

10.2 The bill has two sides, and one of them repeats

Input tokens are everything the model reads: your prompt, your `CLAUDE.md`, every file it opened, the output of every command it ran, and its own earlier replies. **Output tokens** are what it writes back, including the reasoning it does before answering. Output is *pricier* per token. Input is where the volume lives, and volume wins by a mile!

Now the part that catches everyone. I mean everyone. The model has no memory between turns. None. Each time you send a message, the whole conversation is shipped back over and read again from the top, because that is the only way the thing knows what you were talking about. Your context is not paid for once when it arrives. It is paid for on every single turn it survives.

Which means a bloated session does not cost you extra once. It taxes you on message 4, and on message 19, and on message 40, quietly, at the same rate, *forever*, until you **clear** it.

One big file read on turn 3 is a purchase. The same file still sitting in context on turn 30 is a subscription.

Two things people confuse. It's worth ten seconds to separate them. The **context window** is a container: how much text the model can hold at once. Your **usage limit** is a budget: how much reading and writing your plan lets you do this week. You can be nowhere near filling the window and still be broke. Filling the window just means the session starts forgetting things, usually the thing you needed.

10.3 Two ways to spend a fortune before dinner

Scenario one: you attach the spec. It seems reasonable. The agent should know what the assignment says, so you drag the PDF into the chat and start working. That is roughly 35,000 tokens for the text version, and considerably more if the PDF went in as page images. And this time there is a special temptation, which is to attach Assignment 1's spec as well, on the grounds that this one refers to it. Now you are at 65,000 before you have asked a question. Fine, once.

But it is not once. You then have a normal working evening, thirty messages of back and forth about the slab carving loop, and every one of those thirty messages carries both specs along for the ride. Two million tokens, spent on documents that are sitting open on your second monitor, that you have already read twice with a pen, and that you could have quoted the relevant paragraph from for about 200 tokens!

And here is the insult on top of the injury: it did not even help. The model now has twenty pages of Mutation 4 and grading rubrics competing for attention with your actual question. You paid a fortune to make the answer slightly worse. This is what Section 3 was quietly protecting you from. Your notes and your `CLAUDE.md` are the compressed version of the spec, written by the one person who knows which parts matter to tonight.

Scenario two: the 10:40 pm shrug. You type "I have a leak somewhere, fix it." Here is what that actually buys.

	"I have a leak, fix it"	The scoped version (Section 9)
Agent reads	`rbtree.c`, `pool.c`, both test files, the injector, the Makefile, the header. Call it 50k.	`node_alloc` and `rb_insert`. Under 2k.
Tool output	Full `make test` plus the entire unfiltered sweep. Another 45k.	The four lines you grepped. Call it 100.
What comes back	A confident 300-line diff across four files, one of which is a test.	A 15-line change to one path.
What you do next	Read 300 lines at 11 pm, or bless them unread and fib about it in `PROMPTLOG.md`.	Read 15 lines. Run `make memcheck`. Commit.
Turn cost	∼95k, and every later turn in the session pays it again.	∼2k, and the session stays light.
After ten turns	Comfortably past a million tokens. Leak still there.	Bug fixed on turn two. You went to bed.

Same bug. Same tools. Roughly two orders of magnitude between them, and the cheap one is also the one you can actually review, which was the real goal the whole time. Frugality here is not a virtue you practice at the expense of quality. It is the same behavior, seen from the billing department.

10.4 Sessions, caps, and a clock that has never heard of your deadline

Your Pro plan runs on two timers at once, and you already know both. They have not become more generous since Assignment 1.

The first is a **session window** of about 5 hours. It starts with your first message, not at some tidy hour, and when you exhaust it you wait for it to roll over. The second is a **weekly cap** that sits across everything you do. Anthropic assigns your reset day and time to your account, so it is the same slot every week *no matter when* you happen to start working. `/usage` in Claude Code tells you where you stand, as does Settings → Usage on the web. Anthropic does not publish an exact token number for these, partly because the honest answer is "it depends on what you are doing," which is unsatisfying but true.

A few consequences worth writing on your hand:

- It is one allowance, not several. Chatting on claude.ai, working in the desktop app, and running Claude Code in your VS Code terminal all draw from the same pot. The pot does not care which window you were in.
- Attachments and conversation length are the variables. Message count is a terrible proxy. Forty short scoped prompts can cost less than five messages in a session dragging two PDFs and a repository behind it.

- Model choice moves the needle hard. The heavyweight model burns the allowance several times faster than Sonnet for the same work. Routine implementation does not need it. `/model` is one deliberate decision per task, not a setting you pick once out of vanity.
- The clock is wall-clock. Nothing about "this is due at 11:59 pm" persuades the meter to refill at 11:58. It is a thermostat, not a professor, and thermostats don't grant extensions.

Which brings us to the arithmetic that Section 6 was really about. Three calendar weeks of work means three weekly allowances, spent by a rested person asking narrow questions. The same work compressed into the final weekend means one allowance, shared with panic, spent by someone who has stopped reading diffs. Those are not the same project with different start dates. They are different projects.

And when you hit the cap, the failure mode is specific and awful: your threads go read-only, you can look but not ask, and there is no appeal. Support cannot reset it. Your TA cannot reset it. You sit there with a sweep failing at $n = 61$ and a countdown, which is a genuinely miserable way to spend a Sunday. I say that with sympathy not smugness, because everyone learns this exactly once, and some of you learned it last month.

10.5 Why brute force always loses

There is a specific pattern I want you to recognize while it is happening to you, because it feels productive right up until the lights go out.

The bug persists. You say "that didn't work, try again." It tries again. Still broken. "Still broken, are you sure?" It apologizes, tries a third thing. Now it is 12:30, you have four rejected diffs in your history, your working tree is a collage of half-reverted edits, and the leak is exactly where it was at ten o'clock.

That loop cannot converge, and the reason is structural rather than a matter of luck:

1. Every failed attempt stays in the context. So turn six is more expensive than turn five, which was more expensive than turn four. Cost climbs monotonically while progress does not.
2. You added no information. "Try again" is not a new fact. It is the same question, asked louder, and the model is drawing from exactly the same well it drew from the first time.
3. The failures are now evidence, and they are misleading evidence! Your transcript contains four confident wrong theories, written in the model's own voice, and models condition on their own text. You have been steadily building a case for the wrong answer.
4. The code degrades underneath you. By attempt five you may well be debugging bugs that did not exist at attempt one, which is a special kind of hell to be paying by the token for.

So brute force does not merely fail. It fails while accelerating! Its terminal condition is your weekly cap. Guessing is a fine strategy against a four-digit padlock, where the space is small and enumerable. A bug is not a padlock. A bug is a wrong belief you currently hold about your own code, and you cannot enumerate your way out of a wrong belief. You have to go and look.

The exit is unglamorous and takes about four minutes. Press Escape. `git restore` back to the last green commit, which costs nothing, unlike asking the agent to un-write its own work. Get

a deterministic repro, shrink it until it is embarrassing, and come back with a smaller question and a file name. Section 9 spells out the loop. It is worth noticing that the disciplined version is not slower. It only feels slower, because typing furiously feels like progress and thinking looks like sitting still.

10.6 A brief and unflattering history of token maxing

A little history, because the habit you are about to be tempted by has a lineage.

In the early days the context windows were small. A couple of thousand tokens and prompting was mostly the art of deciding what to leave out. Then the windows grew. Eight thousand, thirty-two, a hundred, a million! Vendors advertised them like horsepower. "Paste your entire codebase" became a feature bullet rather than a warning.

People did what people do. They pasted the entire codebase. They pasted the novel, the transcript, the forty-file retrieval dump, all of it hauled in "just in case,". On the theory that context is information and information is good. Tooling grew up to help. Which is to say tooling grew up to make it effortless to be profligate! By the time agentic tools could read files on their own initiative, in a loop, unattended, a genuine subculture had formed around leaving one running overnight to see how much it could chew through. A small number of enthusiasts were burning thousands of dollars of compute on twenty dollar subscriptions and sharing logins to do it. Which is roughly why the five-hour window and the weekly cap exist. The sport acquired a scoreboard, and then it acquired a referee.

The punchline arrived from the research side, and it was not kind. More context frequently makes answers *worse*. Models lose material buried in the middle of a long window, and irrelevant filler dilutes *attention* away from the thing you actually asked about. Feed a model your whole repository to fix one carving bug and you have paid a premium for a diluted answer, and then paid again to review the extra code it wrote because it noticed some unrelated thing on the way past.

It is the all-you-can-eat buffet strategy of loading one plate with everything so you never have to walk back! The price is the same either way, sure. But the shrimp now tastes of gravy, the ice cream has gone into the soup, and you cannot actually find the shrimp. The model is standing at that plate. You built it for them.

Or if you prefer here's the version I see in the library every semester: the textbook where every single sentence has been highlighted in yellow. Enormous effort. Real money spent on markers. A page where everything is highlighted is a page with no highlighting on it at all.

Token maxing is what you do when you have confused thoroughness with volume. Thoroughness is knowing which forty lines matter. Anyone can paste four thousand.

10.7 Odds, ends, and cheap wins

Things that cost more than students expect:

- Screenshots of your terminal. Images tokenize expensively, and you had the text right there, selectable, greppable. Paste the text. Paste the twenty lines of it that matter.
- Letting the agent explore instead of telling it where to look. Every `grep` and speculative file read it does on your behalf is input you are buying. Name the file. Name the function. That

"Code map" block in `CLAUDE.md` from Section 5 exists precisely to stop the spelunking.

- The unfiltered sweep. It prints one line per n and there are hundreds of them. Pipe it. `grep` for the failures and paste those.
- Build artifacts. Keep `build/` in `.gitignore` and out of the agent's way. Nobody needs to read an object file, least of all at these prices!
- Sessions that never end. `/clear` between milestones, `/compact` after a long thread lands. A session you keep open across three evenings is carrying two evenings of dead theories on every message.
- A `CLAUDE.md` that has grown a personality. It loads every session. Keep it to project law. It is a standing charge, and standing charges deserve scrutiny.

Things that are free, or near enough:

- Reading the spec yourself. Rereading your own `DEVLOG.md`. Both of these replace questions you would otherwise pay to ask.
- `grep`, `head`, `tail`, and a moment's thought about which lines of output are the interesting ones.
- `git restore`, `git diff`, `git bisect`. Your history is the undo button that does not bill you.
- The Escape key, the instant a response starts heading somewhere you did not ask for. It depreciates by the second.
- `/usage` at the start of every work session. Ten seconds. The point is to never be surprised.

One last thing, and it is the one I would like you to carry past this assignment. Everyone in this class has the same cap. Nobody is issued extra. So when you notice, in week three, that some of your classmates are finishing calmly while others are rationing messages and staring at a reset timer, understand that you are not looking at a difference in resources. You are looking at a difference in question quality, compounded over nine evenings. That gap is the whole skill, and it is being graded whether or not anyone is watching.

11 Tokens: Spending Like It Is Your Money, Because It Is

You have read this far, which means the token lecture is mostly over; you just heard it wearing other hats. That is the quiet trick of this assignment, and the spec says it outright: cost scales with context. So nearly every habit that makes you a disciplined engineer also makes you a frugal one. Here is the ledger, assembled in one place:

Habit	Why it saves tokens	You were doing it anyway because...
Small modules, named in prompts	The agent reads one function, not a repository.	Section 5: maintainability.
Plan mode before code	Reading and thinking are cheap; un-writing 400 confident wrong lines is not.	The spec's Step 3, and your grade.
Tests-first, smallest slice	Small diffs are reviewed once, not three times.	Section 8.
`/clear` per milestone, `/compact` after long threads	Stale context is billed on every message, forever, like a storage unit.	Section 6: fresh sessions keep the agent sharp.
Filtered tool output	Twenty relevant lines beat four hundred lines of sweep summary. `grep` is free.	Section 9.
Labelling the failing n	One sentence of localization replaces a file's worth of reading.	You wanted the answer tonight.
Commit before risk; `git restore` over regeneration	Reverting is free. Regenerating is the same wrong answer, purchased twice.	Section 7.
A devlog instead of asking Claude what you did	Your notebook is free memory. The context window is the expensive kind.	Section 12, and the quiz.
Lean `CLAUDE.md`	It loads every session; every line is a standing charge.	You dislike relitigating brace style at breakfast.

To that ledger, add the meter itself. `/usage` at the start of each work session, ten seconds, so the weekly cap is never a surprise. `/context` whenever a session feels sluggish, because a ballooning context is something you want to notice before it forces a compaction, not autopsy afterwards. And `/model` deserves one deliberate choice per task: routine implementation does not need the heavyweight model. Save it for the genuinely hard reasoning in this assignment, which is roughly two things: the unwind analysis in Mutation 1, and the refcount decrement order in Mutation 4 if you take the Reach. It is not required for a `pool_stats` function that returns three numbers through three pointers. Everything else, Sonnet handles happily at a lower burn.

But the single biggest lever is not a command at all. It is the calendar. Spread across three weeks, this assignment fits inside the allowance with room to argue about tabs versus spaces. Compressed into a weekend, it meets both clocks at once. The spec has already told you which of those clocks has no email address.

12 Write It Down While It Is Warm: The Devlog

You will forget what you built. Not might. Will. The quiz at the walkthrough arrives weeks after you wrote the code it asks about, a near-twin of this tree will sit inside the Assignment 4 scheduler months from now, and if life intervenes and you have to set the project down and pick it up later, future-you inherits whatever present-you bothered to write down. Memory is not a plan. A devlog is.

This assignment has a second reason, too. Three of the things `REFLECTION.md` asks you to defend, the slab-header choice, the alignment guarantee, and the `pool_stats` definitions, are decisions you will make in about ninety seconds each, in the middle of week two, and then completely forget the reasons for. Write them down as you make them and the reflection transcribes itself.

So: `DEVLOG.md`, in the repo, five minutes at the end of every session, before the laptop closes. Not prose. A dated entry with the same small skeleton every time:

```
## 2026-10-06 (week 2, evening 2, ~2h)
STATE: pool geometry green; sweep fails at n=61 on the pooled build.
DID: slab header in-band at offset 0; slot stride =
 align_up(sizeof(rb_node_t), alignof(max_align_t)); free list.
DECIDED: in-band header. One rb_malloc per slab instead of two, so
 one fewer failure path for Mutation 1 to find. Costs me 2 slots
 per slab, which pool_stats now reports honestly.
LEARNED: n=61 is the first slab allocation inside pool_create, not
 a node alloc. rb_create_pooled was returning a half-built tree
 instead of NULL, and the sweep caught it on the very first run
 that touched it. The injector's per-allocation label paid for
 itself tonight.
NEXT FIRST STEP: make rb_create_pooled unwind (free pool, free
 handle, return NULL), then re-run the sweep from n=55.
OPEN: poison vs free-list link -- both want the first 8 bytes.
 Plan: memset 0xDD over the whole slot, then write the link over
 the front of it. Confirm ASan/UBSan are quiet about the store.
```

The `NEXT FIRST STEP` line is the one that pays compound interest. Every session opens by reading it, which means no session opens with twenty minutes of "where was I". No resumed project, after a week or a month away, opens with an evening of rediscovery. It is the difference between a five-minute restart and an archaeology dig through your own git log.

Alongside the log, keep decision records: one-liners, written the moment you choose. "Right-spine teardown over pointer reversal: reversal during *destruction* re-threads pointers I am about to free, and I don't trust myself to sequence that under pressure." One sentence. That sentence is your walkthrough answer, pre-written by the only person who ever knew the real reason. And the invariant comments your `CLAUDE.md` style rules already require are the same idea living inside the code, sitting exactly where the quiz will point.

Because here is the compounding part: the devlog is not extra paperwork on top of the deliverables. It is what the deliverables are smelted from. `PROMPTLOG.md` annotation becomes transcription instead of archaeology. `REFLECTION.md` is half-written before you start it. The live modification at the walkthrough starts from your notes instead of from zero. Five minutes a night,

and every downstream document gets cheaper. That is the best exchange rate in this course.

13 Your Prompt Log Is a Professional Artifact

One last reframe, and it is the one to carry out of this course. The transcript you are producing here is not homework exhaust. It is a genre of document your career is going to be full of.

In teams that work with coding agents, and that is rapidly becoming most teams, session transcripts get shared, reviewed, inherited, and mined the way pull requests are. A tech lead reads them to understand how a problem actually got solved. A teammate inherits your session to continue your work, and inherits your reasoning with it. Token spend shows up as a line item that somebody with a budget looks at. And when a bug ships, the transcript is the accountability trail: it shows what was reviewed, what was merged on faith, who decided, and when. "The agent wrote it" is not a defense in any code review on Earth, which is exactly the spec's you-own-the-bugs rule, wearing a badge.

Which means your prompts are legible evidence of how you think. A transcript full of "fix it" and "make it work" reads one way. A transcript where every prompt names a file, cites a failing test, states the constraint, and pushes back on a hand-wavy plan reads another way entirely, and the person reading it is forming an opinion of you either way. The habits are the same ones that get sanded into you privately; the stakes just grow teeth once the audience does. Because habits are precisely the things that follow you into rooms where they matter.

Notice also that the transcript point has doubled since last time, from half a point to two, and that the rubric did not change one word. That is not an accident either. The thing being measured got more valuable, not more lenient. Write every prompt as if a senior engineer you respect will read it later, because the TA literally will, and someday the senior engineer literally will too. Annotate `PROMPTLOG.md` the way you would annotate a design doc for a teammate: here is what I asked, here is what came back, here is my judgment and my reasons. Look at the transcript rubric's level 5 and notice that it is not describing a grading gimmick. It is describing what a good engineering session looks like anywhere. The rubric is a job description in disguise.

14 How AI was Used in Developing This Companion

This companion document was developed with assistance from AI tools. Its structure, recommendations, examples, sequencing, and pedagogical choices are mine. I used Claude and Claude Code primarily to stress-test explanations, check technical details, identify ambiguities, and see what happened when the guidance was followed literally. I then revised. The document then went through several additional rounds of revision, including review by the TAs, who caught issues and edge cases that I had overlooked.

That process is worth mentioning because it mirrors what I am asking you to do. Good work rarely arrives fully formed. You build something, test it, let other eyes find what you missed, revise it, and then do the whole thing again. This companion got better through exactly that process. Yours should too.

15 A Last Word

The spec grades two things: the tree and you. This companion has been almost entirely about the second one, because the second one is the part that transfers. Trees are forgotten. The shape of *how* you work is not.

So: read the whole thing before you type. Turn your reading into notes, and your notes into plans. Mark your own commit points with a pen before you ask anyone else where they are. Build in modules, and only the modules the current week needs. Let tests lead and keep the diffs small; guard the Assignment 1 suite like the contract it has become; make the agent show its work, every time. Commit every green state like the checkpoint it is. Debug with a repro and a file name, never with a shrug and a repository. Watch the meter without worshipping it. And spend five minutes each night telling the future-you what happened, because that future-you is the one taking the quiz.

None of what I have laid out here is heroic. That is the point! Nine calm evenings beat one "legendary" weekend every single time. Only one of those makes a good story at the walkthrough. Go open your repo!

16 Version History

This section reflects the change history for this companion document. Changes clarify the walkthrough; the spirit of it will not change, and nothing in it ever overrides the assignment specification.

Version	Date	Comments
1.0	9/16/2026	First public release of the companion walkthrough for Assignment 2.