AI Coding Assistants Guide

PDF

TABLE OF CONTENTS

1. Getting Started With AI Coding Assistants

1.1 What AI Coding Assistants Do and Do Not Do

AI coding assistants help you write and revise code faster, but they are not a substitute for engineering judgment. Think of them as a fast collaborator that can draft code, explain it, and suggest next steps—provided you supply enough context and you verify the result.

What they *do* well

1) Draft code from a clear description If you describe inputs, outputs, and constraints, an assistant can produce a reasonable first implementation.

Example (Python):

- You ask for "a function that validates an email string and returns a boolean; accept only ASCII; reject spaces."
- The assistant can generate a function using a regex and show how to test it.

2) Convert between formats and styles They can translate patterns you already use: converting a loop to a list comprehension, rewriting a function to match your project's style, or changing error handling conventions.

Example:

- You provide a snippet that uses `Result<T, E>`-style errors.
- You ask for the same logic in a new function signature.
- The assistant can mirror the error-handling pattern so the codebase stays consistent.

3) Explain code and likely causes of issues When you paste a function and an error message, they can propose what might be wrong and which lines are suspicious.

Example:

- You paste a stack trace showing `IndexError` in a loop.
- You ask, "What condition could cause this?"
- The assistant can point to off-by-one boundaries and suggest a minimal fix.

4) Generate tests and edge cases They can draft unit tests that cover typical scenarios and common edge cases, especially when you specify expected behavior.

Example:

- You ask for tests for a "normalize path" function.
- The assistant can include cases like empty strings, trailing slashes, and `..` segments.

5) Suggest refactors that preserve behavior They can propose reorganizing code for readability: extracting helper functions, reducing duplication, or improving naming—when you ask for "no behavior change" and you provide tests.

What they *do not* do (or do unreliably)

1) Guarantee correctness without verification An assistant can produce code that looks plausible but fails in real execution. You still need to run tests, lint, and review.

Example:

- You ask for "a correct SQL query."
- The assistant may return a query that is syntactically valid but wrong for your schema or business rules.
- Running it against a test database (or at least validating assumptions) is what makes it trustworthy.

2) Understand your system's hidden context They do not automatically know your runtime environment, data shape, performance requirements, or security constraints unless you provide them.

Example:

- Your service expects timestamps in UTC, but your assistant might assume local time.
- If you don't state the requirement, the generated code can be subtly wrong.

3) Replace good debugging habits They can suggest hypotheses, but you still need to reproduce the bug, inspect inputs, and confirm the fix.

Example:

- If a function fails only under load, the assistant might suggest a logic fix while the real issue is a race condition.
- You need logs, metrics, and targeted tests to confirm.

4) Make security decisions for you They can help you write safer code patterns, but they can also miss threat models or authorization rules.

Example:

- You ask for "user search by query."
- The assistant might generate a parameterized query, but you still must ensure authorization checks happen before data access.

5) Maintain long-term consistency across a whole project by magic They can help with local changes, but large refactors require careful planning, dependency awareness, and review.

Example:

- You ask for a rename across files.
- The assistant might miss a reference in a test fixture or documentation string.
- A real search-and-verify workflow prevents broken builds.

How to get useful results (and avoid wasted effort)

Provide the "contract" State what the code must do: inputs, outputs, error behavior, and constraints.

Example prompt structure:

- "Write `parse_order_id(s)` that returns an integer for strings like `ORD-1234`. If invalid, raise `ValueError`. Only accept ASCII characters."

Include the failing evidence when debugging Paste the exact error message, the relevant code, and a minimal input that triggers the issue.

Example:

- "This function throws `TypeError: unsupported operand type(s)` when `x=None`. Here is the function and a call that reproduces it."

Ask for tests and run them Even a small change should come with a test that proves the behavior you care about.

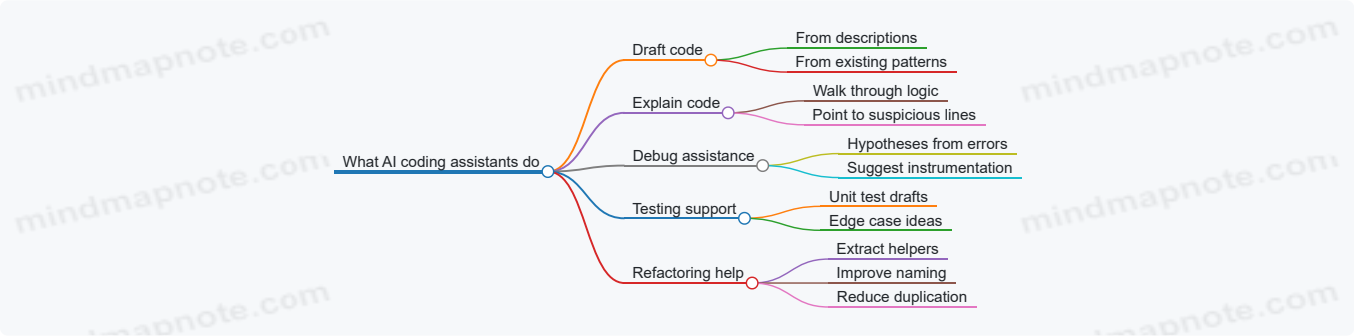
Example:

- "Add a unit test for the edge case where the list is empty, then update the implementation to make it pass."

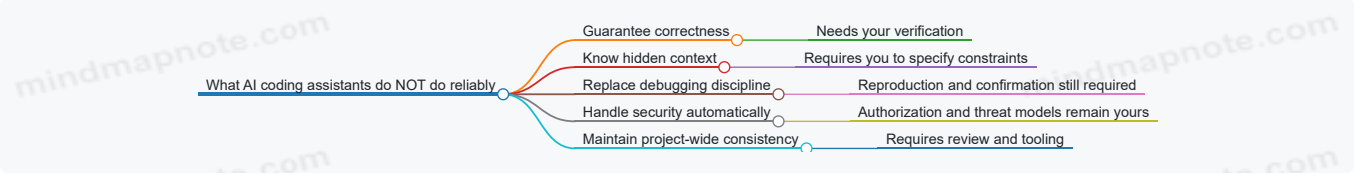
Treat suggestions as drafts Review for correctness, performance, and security. If something matters, verify it with execution.

Mind maps

Mind map: What AI coding assistants do



Mind map: What AI coding assistants do NOT do reliably



A practical “do vs. don’t” checklist

- ✅ Do ask for code that matches a stated contract.
- ✅ Do request tests for the behavior you care about.
- ✅ Do run the code and confirm the fix.
- ❌ Don’t assume the assistant’s code is correct because it compiles.
- ❌ Don’t omit environment constraints (time zones, encodings, data formats).
- ❌ Don’t skip security review for authorization and input handling.

When you use an assistant this way, you get speed without losing control: you supply the engineering constraints, and you validate the output like you would with any other contributor.

1.2 Choosing the Right Assistant for Your Workflow

Picking an AI coding assistant is less about finding “the best” tool and more about matching its strengths to the work you actually do. The right choice reduces back-and-forth, keeps changes consistent with your codebase, and makes debugging less of a guessing game.

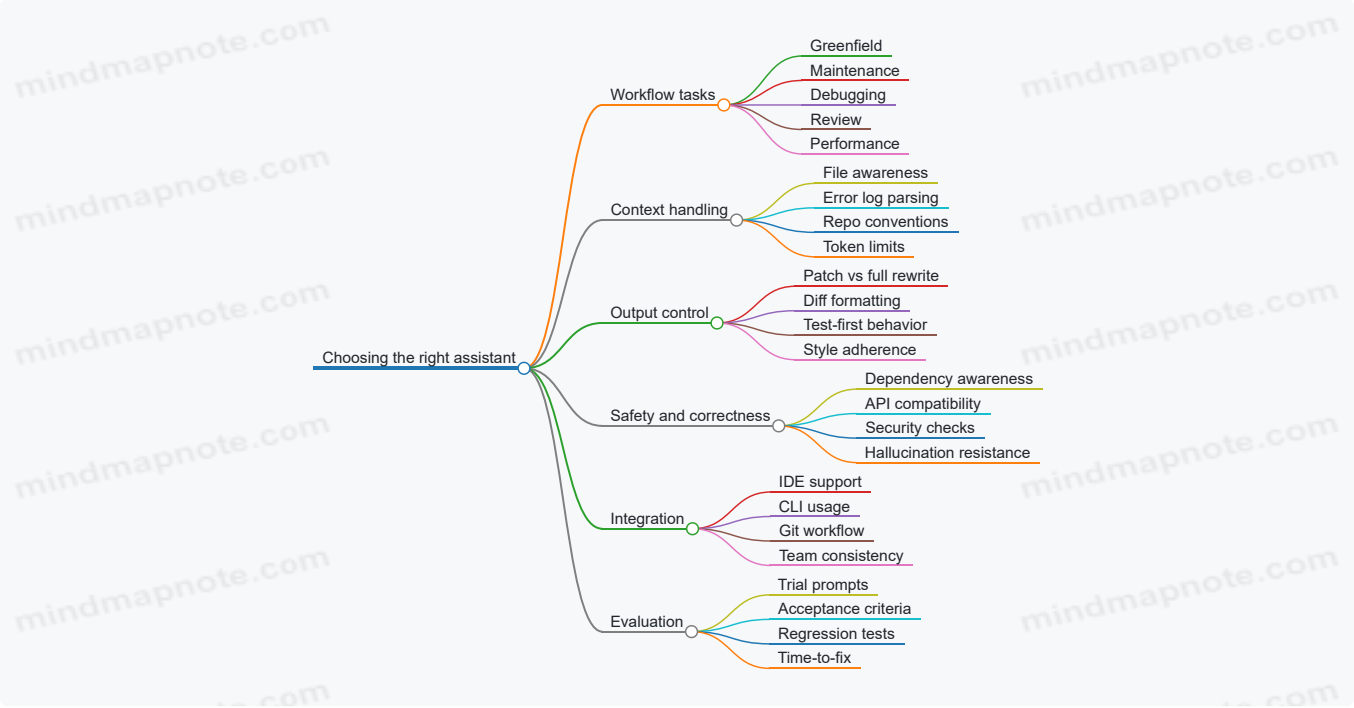
Start with your workflow shape

Before comparing tools, describe your day in terms of tasks. Most teams do a mix of these:

- **Greenfield coding**: writing new modules, endpoints, or scripts.
- **Maintenance**: fixing bugs, updating dependencies, refactoring.
- **Debugging**: interpreting stack traces, reproducing issues, narrowing causes.
- **Review and cleanup**: improving readability, tests, and edge cases.
- **Performance work**: profiling, reducing allocations, optimizing queries.

Different assistants handle these differently. Some are better at generating code from specs, while others are better at reasoning from existing files and error logs.

Mind map: how to choose



Evaluate by “fit,” not by vibes

Use a short trial that mirrors your real work. The goal is to measure how quickly you can reach a correct, reviewable result.

1) Context handling: can it see what matters?

Ask it to work with a small but realistic slice of your code.

- Provide **one file** plus a **brief description** of how it's used.
- Include **the exact error message** (or a trimmed stack trace).
- Tell it what you want changed: "fix the bug," "add validation," or "write tests."

What you're looking for:

- It should reference the right functions and variables.
- It should avoid inventing new abstractions when a small fix is enough.
- It should ask clarifying questions when the missing details would change the solution.

If the assistant frequently ignores the provided code or produces code that doesn't compile, it's a context mismatch.

2) Output control: can you steer it safely?

Some assistants produce a full rewrite even when you asked for a targeted change. That's not automatically bad, but it increases review cost.

Prefer tools that support one or more of these:

- **Patch-style edits** (or diff-like outputs)
- **Line-by-line changes** tied to the original code
- **"Keep behavior the same"** constraints
- **Test generation** that matches your test framework

A simple test: ask for a small refactor that should not change behavior, like renaming a function and updating call sites. If it changes logic too, you'll pay for it later.

3) Correctness habits: does it verify?

Good assistants don't just output code; they help you validate it.

Look for behaviors such as:

- Suggesting a minimal set of tests to run.
- Pointing out likely edge cases (nulls, empty lists, time zones, integer overflow).
- Explaining assumptions it made based on the provided snippet.

If it never mentions verification steps, you'll end up doing all the quality control yourself.

Match assistant strengths to task types

Here's a practical mapping you can use when choosing.

Greenfield coding

You want:

- Fast generation from specs
- Consistent formatting
- Ability to produce tests alongside code

Try a prompt like: "Create a small module that parses input and returns structured output. Include unit tests for normal and edge cases."

If the assistant produces tests that cover the edge cases you named, it's a good sign.

Debugging

You want:

- Strong interpretation of stack traces
- Suggestions for reproduction steps
- Minimal changes that address the root cause

Trial prompt: "Given this stack trace and this function, identify the most likely cause and propose a fix. Then list two targeted tests that would fail before the fix."

If it jumps straight to rewriting unrelated parts, it's not respecting the debugging workflow.

Review and refactoring

You want:

- Readability improvements without changing behavior
- Clear reasoning for each change
- Awareness of your style rules

Trial prompt: “Review this function for clarity and edge cases. Propose a refactor that keeps the same inputs/outputs. Provide a short list of behavior-preserving changes.”

A useful assistant will separate “style improvements” from “behavior changes” and will avoid mixing them.

Performance optimization

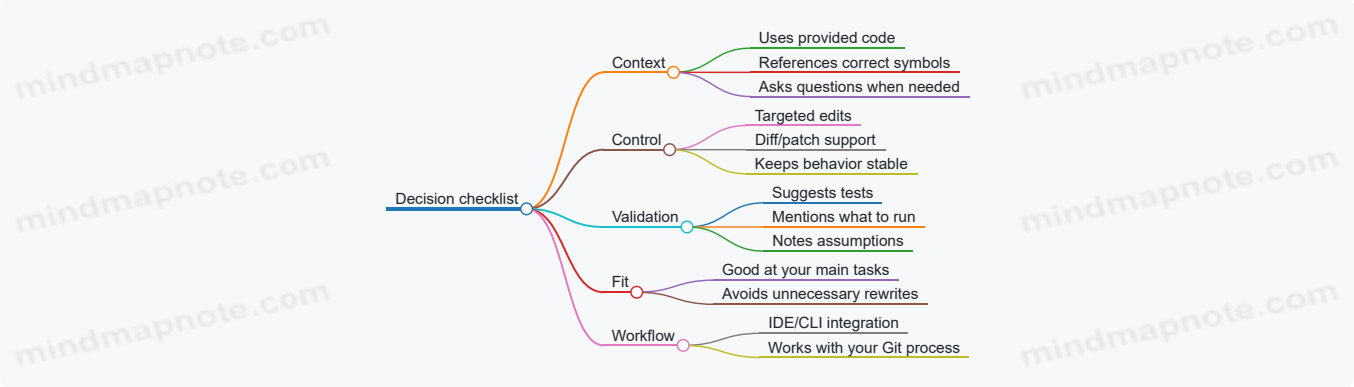
You want:

- Suggestions tied to measurable bottlenecks
- Awareness of algorithmic complexity
- Guidance on profiling and benchmarks

Trial prompt: “This endpoint is slow. Based on the code, identify likely hotspots and propose two optimizations. For each, state what metric should improve and how to measure it.”

If it can’t connect changes to metrics, it will be hard to confirm improvements.

Mind map: a quick decision checklist



Concrete example: comparing two assistants with the same task

Task: Fix a bug in a function that calculates totals.

You provide:

- A snippet of the function
- A failing test name and expected output
- The actual output

Assistant A responds with:

- A targeted fix in the calculation
- Updated test assertions
- A note: “Run `pytest -k totals` to confirm.”

Assistant B responds with:

- A full rewrite into a new structure
- New helper functions not present in your codebase
- No clear instruction on which tests to run

Even if both might eventually work, Assistant A is a better fit for maintenance workflows because it reduces review and verification overhead.

Practical scoring rubric (use it for a 1-hour trial)

Score each assistant from 1 to 5 on:

- **Context accuracy** (does it use your code correctly?)
- **Change minimality** (does it avoid unnecessary rewrites?)
- **Test usefulness** (are suggested tests relevant and runnable?)
- **Debug reasoning** (does it narrow causes logically?)
- **Edit control** (does it produce patch-like changes?)

Pick the assistant that scores highest for your most frequent task type, not the one that impresses you on the first prompt.

Final rule of thumb

Choose the assistant that consistently produces **small, verifiable changes** in the areas you work on most. If it can't reliably do that, you'll spend more time correcting it than coding.

1.3 Setting Up a Reproducible Development Environment

A reproducible development environment means: if you check out the same commit on a different machine (or a clean machine), you can run the same commands and get the same results. AI coding assistants help most when your setup is predictable, because "works on my machine" turns debugging into interpretive dance.

What "reproducible" means in practice

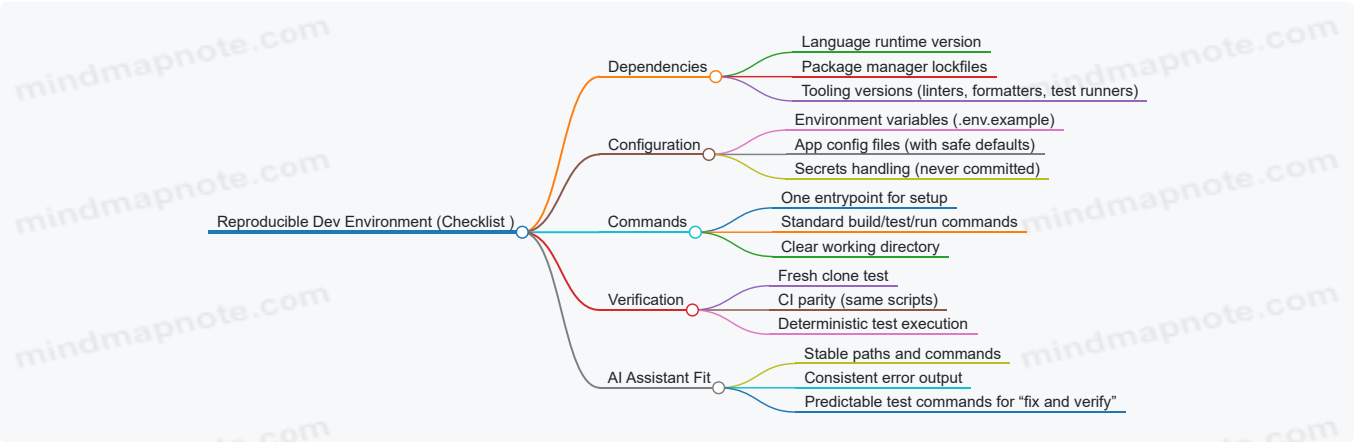
Reproducibility has three layers:

1. **Same dependencies**: the same versions of libraries and tools.
2. **Same runtime behavior**: the same environment variables, configuration files, and OS-level expectations.
3. **Same commands**: the same build/test/run steps, with the same working directory assumptions.

If you can't describe these three layers in a short checklist, your environment is probably more "suggestive" than reproducible.

Mind map: the reproducibility checklist

Reproducible Dev Environment (Checklist Mind Map)



Step 1: Pin the language runtime and tooling

Start by pinning the runtime version your project expects.

Example (Node.js):

- Use an `.nvmrc` file (or equivalent) to pin Node.
- Use `package-lock.json` (or `pnpm-lock.yaml` / `yarn.lock`) to pin dependencies.
- Pin tool versions via `devDependencies` so the project runs the same formatter and linter everywhere.

Example (Python):

- Use `pyproject.toml` and a lock mechanism (e.g., `poetry.lock` or `requirements.txt` generated from a lock).
- Pin the Python version in your project config and enforce it in your setup script.

Why this matters for AI-assisted coding: when the assistant suggests a fix, it often assumes the same language features and library APIs. If your runtime differs, the assistant's "correct" code can fail for reasons unrelated to the bug.

Step 2: Lock dependencies and keep lockfiles in version control

Lockfiles are the difference between "installing" and "re-creating."

Best practice: commit the lockfile and ensure your setup uses it.

Example (Node.js commands):

- Install with the lockfile-aware command (e.g., `npm ci` instead of `npm install`).

Example (Python commands):

- Install from a pinned requirements file generated from a lock.

If your project uses multiple package managers or multiple lockfiles, pick one for the main workflow and document it in the setup instructions.

Step 3: Standardize configuration with a template

Most "non-reproducible" setups fail because configuration differs: database URLs, feature flags, API keys, or debug modes.

Best practice: add an `.env.example` file that lists required variables with placeholder values.

Example `.env.example`:

- `DATABASE_URL=postgres://user:pass@localhost:5432/app`
- `APP_ENV=development`
- `LOG_LEVEL=info`

Then your setup script can copy `.env.example` to `.env` if `.env` doesn't exist.

Secrets rule: never commit real secrets. If a variable must be present, use a placeholder and fail fast with a clear message when it's missing.

Step 4: Provide a single entrypoint for setup

A reproducible environment needs a predictable "do this first" command.

Best practice: add a `make` target or a package script like `setup` that performs:

- dependency installation
- environment file creation (from template)
- any local code generation steps

Example (Makefile-style):

```
setup:
	cp -n .env.example .env || true
	# install dependencies using the lockfile
	# run any codegen steps if required
```

If you don't use Make, you can do the same with a script in `package.json` or `scripts/setup.sh`. The key is that the command is consistent and documented.

Step 5: Ensure deterministic test execution

Tests should behave the same way each run.

Common sources of nondeterminism:

- tests depending on time (`now()` without control)
- random IDs without seeding
- parallel tests sharing state
- database state not reset between tests

Best practice:

- seed randomness in tests
- use a fixed clock helper when possible
- isolate test databases or use transactions with rollback

Example (conceptual):

- If a test creates a user with a random email, assert using the returned ID rather than expecting a specific email string.

Step 6: Verify with a “fresh clone” run

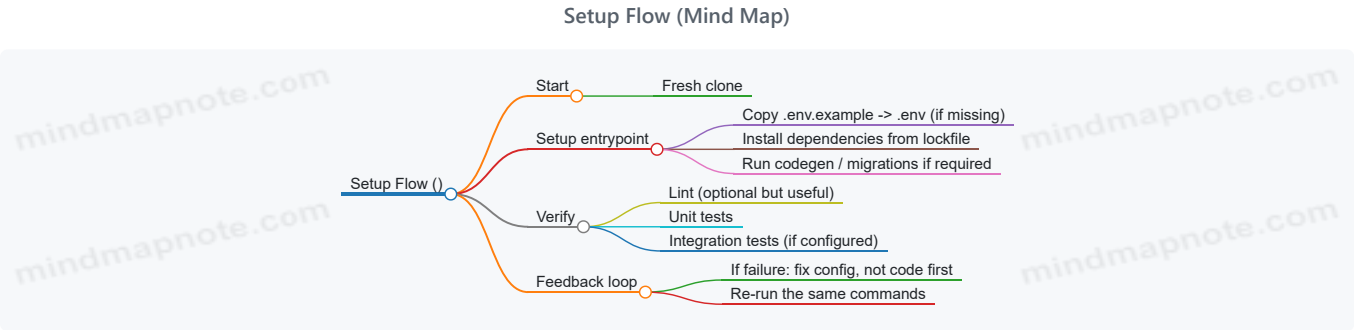
Reproducibility isn’t real until you test it.

Fresh clone checklist:

1. Create a new directory.
2. Clone the repo.
3. Run the single setup command.
4. Run the standard test command.
5. Confirm the same failures (if any) appear.

If step 4 fails, capture the exact command and error output. This is also the best input for an AI assistant: “Here is the command I ran and the full error output.”

Mind map: environment setup flow



Example: a minimal, reproducible project layout

A layout that supports reproducibility usually includes:

- `package.json` / `pyproject.toml` for scripts and dependency declarations
- a lockfile (`package-lock.json` , `poetry.lock` , etc.)
- `.env.example` for configuration templates
- a setup script (`scripts/setup.sh` or `make setup`)
- a test script (`npm test` / `pytest` / equivalent)

Example commands you want to be able to run from the repo root:

- `setup`
- `test`
- `lint` (optional)

If someone must run commands from a subdirectory, your environment is already leaking assumptions.

Practical tips that prevent common failures

- **Fail fast on missing config:** if `DATABASE_URL` is required, check it at startup and print a clear error.
- **Avoid hidden global state:** don’t rely on globally installed tools unless you also document and pin them.
- **Keep generated files consistent:** if code generation exists, make it part of setup so the repo state is the same.
- **Use consistent path handling:** prefer project-relative paths in scripts so Windows and macOS don’t diverge.

Quick self-audit

Ask these questions:

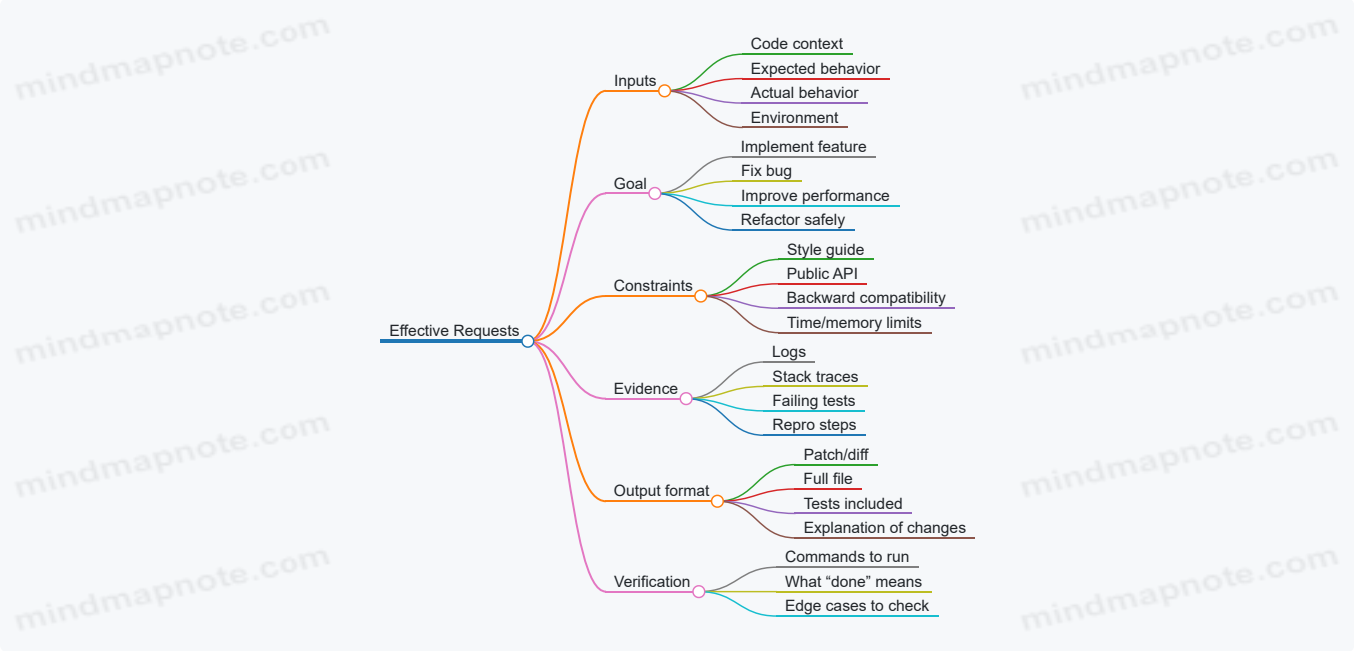
- Can a teammate run one command to set up dependencies?
- Does the project install from a lockfile?
- Is configuration templated with `.env.example`?
- Do tests run the same way after a fresh clone?
- Do AI assistant prompts benefit from stable commands and stable error output?

If the answers are mostly “yes,” your environment is doing its job: it stops the setup from becoming part of the bug.

1.4 Writing Effective Requests for Code, Errors, and Refactors

Good requests make the assistant’s job smaller and your job faster. The trick is to describe what you know, what you want, and what constraints you must not break. Below are practical patterns for code generation, error diagnosis, and refactoring—plus examples you can copy.

Mind map: request anatomy



A simple template that works for most tasks

Use this structure, even if you fill only some fields.

> **Task:** (what you want)
>
> **Context:** (relevant code snippets + how it’s used)
>
> **Goal:** (what correct behavior looks like)
>
> **Constraints:** (what must not change)
>
> **Evidence:** (error text, failing test output, repro steps)
>
> **Output format:** (diff vs full file, tests, explanation level)
>
> **Verification:** (how to confirm it works)

A good request reduces back-and-forth. A great request also prevents “helpful” changes that break assumptions.

Writing requests for code

When asking for new code, specify the contract and the boundaries. “Write a function to parse dates” is vague; “Parse ISO-8601 strings into `datetime` objects; reject invalid formats; keep timezone handling consistent with existing code” is actionable.

Example: feature request with a clear contract

Your request:

> Task: Add a function `parse_user_date(s: str) -> datetime`.
>
> Context: Current code stores user-entered dates in UTC and expects naive `datetime` objects (no tzinfo).
>
> Goal: Accept `YYYY-MM-DD` and `YYYY-MM-DDTHH:MM:SSZ`. For the first format, treat it as midnight UTC. For the second, convert to UTC and return a naive `datetime`.
>
> Constraints: Do not change existing callers. Use the project's existing date parsing utilities if present.
>
> Output format: Provide the function plus unit tests.
>
> Verification: Tests should cover valid inputs, invalid strings, and leap day.

Why this works: it states input formats, normalization rules, and the exact type the rest of the system expects.

Example: ask for incremental changes

If the codebase is large, request a small patch first.

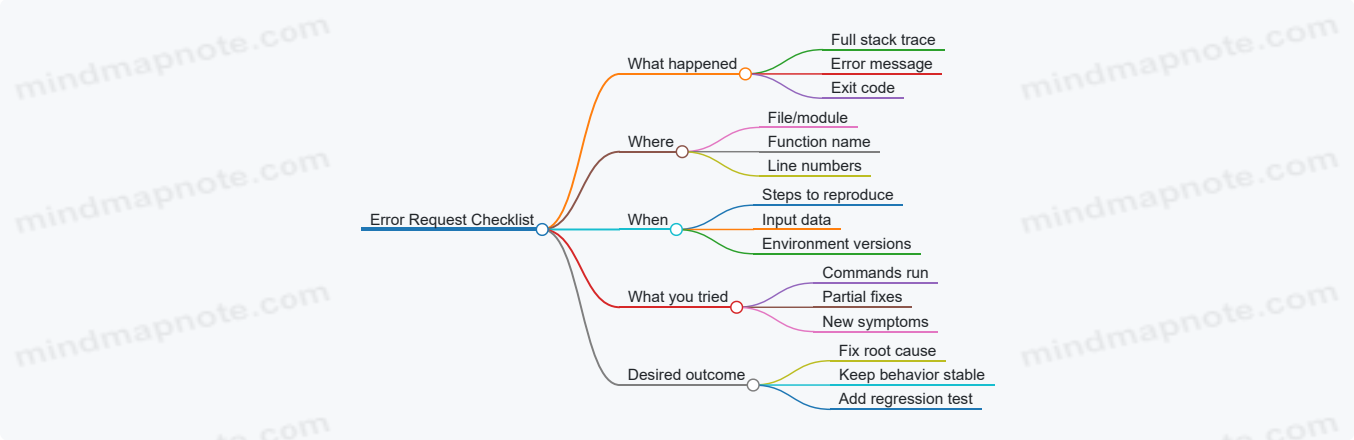
> Task: Implement `get_discounted_price`.
>
> Constraints: Keep the current signature and return type. Do not refactor unrelated modules.
>
> Output format: Provide a minimal diff for only the affected file(s).
>
> Verification: Run the existing unit tests and add one new test for the edge case "quantity = 0".

Incremental requests reduce the chance of accidental behavior changes.

Writing requests for errors

Error requests should include the smallest reproducible case and the assistant's target: "Explain why it fails" or "Propose a fix and show how to verify." The assistant can't debug what it can't see.

Mind map: error request checklist



Example: runtime error with targeted fix

Your request:

Task: Fix a crash in `checkout()`.

Evidence: Stack trace:

```
KeyError: 'currency'
  at pricing.py:118 in apply_tax
  at checkout.py:54 in checkout
```

Context: `apply_tax` expects a dict with keys `amount` and `currency`. In production, `checkout()` sometimes passes a dict missing `currency`.

Goal: Make `checkout()` robust by validating inputs before calling `apply_tax`.

Constraints: Preserve existing tax calculation logic. If `currency` is missing, raise `ValueError` with message `Missing currency`.

Output format: Provide the code change and a unit test that reproduces the missing key case.

Verification: Run the test suite; confirm the new test fails before the change and passes after.

Why this works: it identifies the failing line, the missing key, the desired behavior, and the exact error type/message.

Example: logic bug request with expected vs actual

Task: Diagnose why `is_valid_email` returns `True` for `a@b`.

Context: Current regex is `r"^\w+@\w+\.\w+$"`.

Evidence: In a unit test, `is_valid_email("a@b")` returns `True`.

Goal: Return `False` for missing top-level domain. Keep support for `user.name+tag@example.co.uk`.

Output format: Explain the likely cause, propose a corrected regex, and add tests for the failing case and two valid examples.

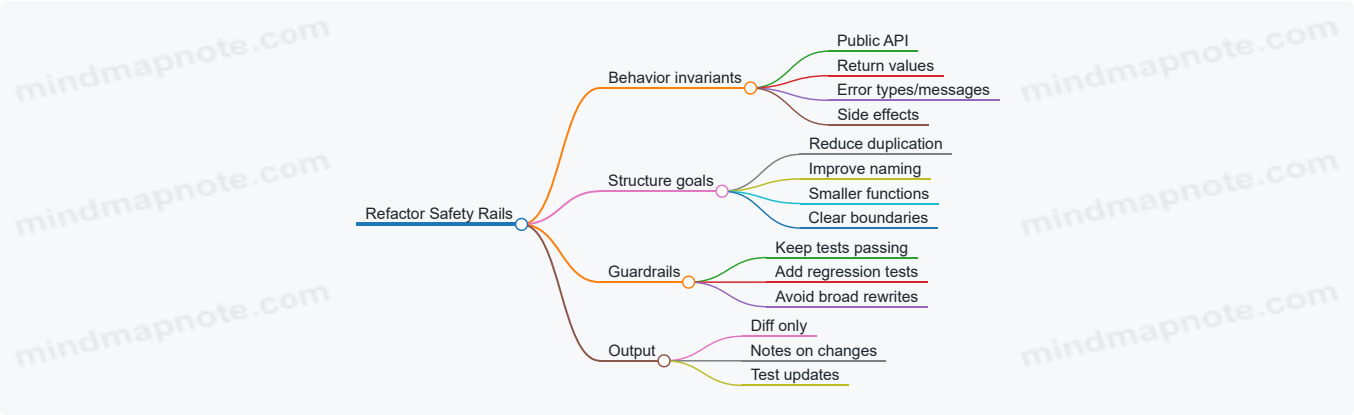
Verification: Run only the email validation tests.

This request forces a comparison between expected and actual behavior.

Writing requests for refactors

Refactors are where “works on my machine” becomes “works, but not the same.” Your request should explicitly define what must remain unchanged: public interfaces, output formats, error types, and performance characteristics.

Mind map: refactor safety rails



Example: refactor with explicit invariants

> Task: Refactor `process_orders(orders)` to improve readability.
>
> Context: The function currently mixes validation, transformation, and persistence.
>
> Goal: Split into three helpers: `validate_orders`, `transform_orders`, and `persist_orders`.
>
> Constraints (must not change):
>
> - Function signature stays the same.
> - Output format stays the same.
> - Existing exceptions and messages must remain identical.
> - No new database calls.
>
> Output format: Provide a diff for the file and update tests only if needed for coverage.
>
> Verification: Run the existing unit tests; add one test that ensures exception messages are unchanged.

Example: refactor request that asks for “no behavior change” proof

> Task: Refactor `calculate_totals`.
>
> Constraints: No behavior change. Keep rounding rules exactly as they are.
>
> Output format: Provide the refactor plus a short list of invariants you preserved (e.g., rounding order, handling of empty lists).
>
> Verification: Add a regression test that compares old and new outputs for a fixed set of inputs.

Asking for invariants makes the assistant’s reasoning visible and helps you review quickly.

Common mistakes to avoid (and what to do instead)

1. **Missing the contract.** If you don’t state expected input/output, the assistant will guess. Fix: include examples of inputs and expected outputs.
2. **Over-sharing irrelevant code.** Huge files drown the useful parts. Fix: paste only the functions involved plus their immediate callers.
3. **No verification plan.** Without “how to check,” changes are hard to trust. Fix: specify test commands or the exact assertions to add.
4. **Unstated constraints.** “Refactor it” invites broad rewrites. Fix: list what must not change (API, exceptions, side effects).

Quick copy-paste request starters

Code

> Task: Implement ___.
>
> Context: ___ (paste relevant snippet).
>
> Goal: ___ (exact behavior + edge cases).
>
> Constraints: ___ (what must not change).
>
> Output format: Provide diff + tests.
>
> Verification: Run ___.

Error

> Task: Fix ___.
>
> Evidence: stack trace / failing test output: ___.
>
> Context: ___ (where it’s called + relevant code).
>
> Goal: ___ (desired behavior or error type/message).
>
> Output format: Provide fix + regression test.
>
> Verification: Run ___.

Refactor

> Task: Refactor ___ for readability.
>
> Constraints: signature, outputs, exceptions, side effects unchanged.
>
> Goal: split into ___ helpers / reduce duplication.
>
> Output format: diff + notes on preserved invariants.
>
> Verification: Run ___; add test for unchanged exception messages.

1.5 Using Context Wisely With Files, Logs, and Constraints

AI coding assistants are best at transforming what you already know into something you can run. The trick is feeding them the right context—enough to be accurate, not so much that you drown in noise. This section shows how to package context using files, logs, and constraints, with examples you can copy.

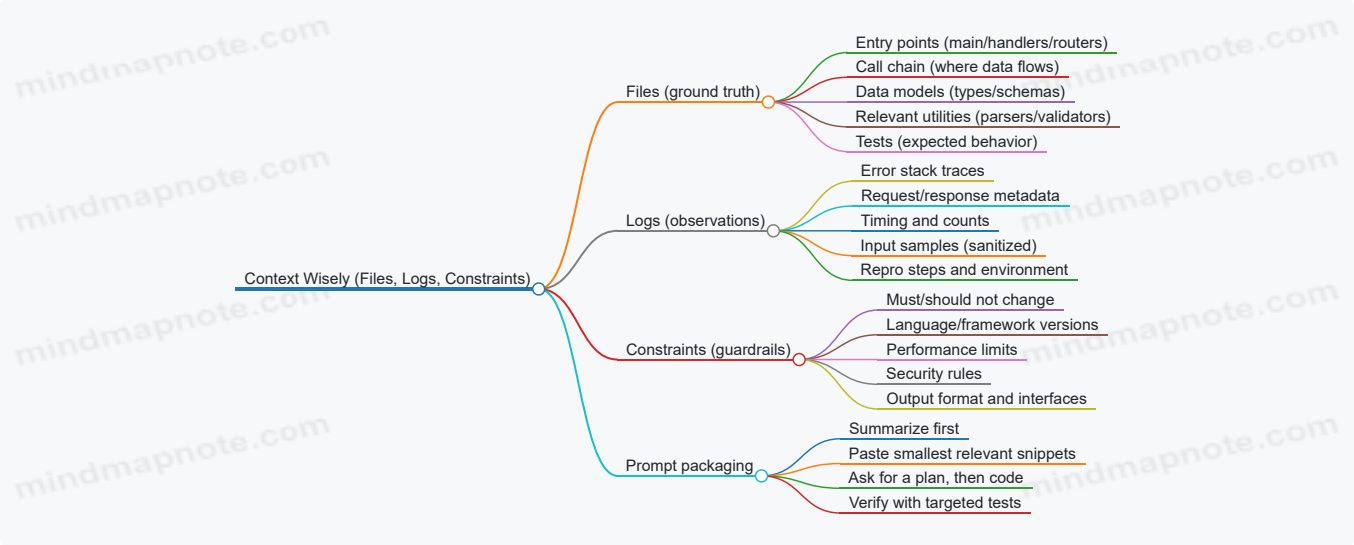
The context goal: "minimum useful truth"

Your assistant needs three things:

1. **What the system is supposed to do** (requirements and invariants).
2. **What it is doing right now** (current code behavior, errors, and observations).
3. **What boundaries it must respect** (style, performance, compatibility, and "don't change X").

If you provide only code, it may guess the intent. If you provide only an error, it may guess the design. If you provide only requirements, it may guess the implementation. Good context combines all three.

Mind map: context sources and how to package them



Files: choose the smallest set that still tells the story

When you paste code, include the parts that explain **why** the bug exists or **where** the change should land.

Good file selection

- The function that fails (or the handler that receives the request).
- The immediate callers and callees that affect inputs/outputs.
- Any types or schemas that define the data shape.
- One or two related tests that show expected behavior.

Avoid dumping the whole repository Large context increases the chance the assistant "fixes" the wrong thing. It also makes it harder for you to verify what changed.

Example: narrowing to the call chain

You're debugging a failing endpoint. Instead of pasting the entire service, paste:

- `routes.py` (the endpoint)

- `service.py` (the function called by the endpoint)
- `models.py` (the request/response types)
- the failing test (or a snippet of it)

Then add a short note:

- "The endpoint calls `create_user(payload)` and returns `UserResponse`."

That note tells the assistant how to connect the snippets.

Logs: extract signal, not just text

Logs are useful when they show a pattern: the exact error, the inputs involved, and the execution path.

What to include from logs

- The full stack trace for the failing request.
- The log lines immediately before and after the failure.
- Any structured fields (request id, user id, route, status code).
- Timing info if performance is the issue.
- A sanitized sample of the input payload (remove secrets).

What to omit

- Repeated noise from unrelated requests.
- Entire log files when you only need one failing trace.

Example: turning a stack trace into a targeted request

Bad prompt: "Fix this error."

Better prompt:

- Paste the stack trace.
- Paste the relevant function.
- Add: "Please identify the failing line, explain the likely cause, and propose a minimal patch with a test."

Even if the assistant is unsure, it will focus on the exact failure location.

Constraints: tell the assistant what not to break

Constraints prevent "helpful" changes that compile but violate your expectations.

Use constraints in three layers:

1. **Interface constraints**: function signatures, API contracts, response formats.
2. **Behavior constraints**: invariants like "must not change ordering" or "must preserve existing error codes."
3. **Operational constraints**: performance, memory, compatibility, and security rules.

Example: constraints for a safe refactor

You want to refactor a parser but keep behavior identical.

Prompt snippet:

- "Do not change the public function signature."
- "Keep the same error messages and error codes."
- "Add tests for the existing edge cases; do not change them."

This turns refactoring into a controlled transformation.

Prompt packaging: a reliable template

A good context prompt usually follows this order:

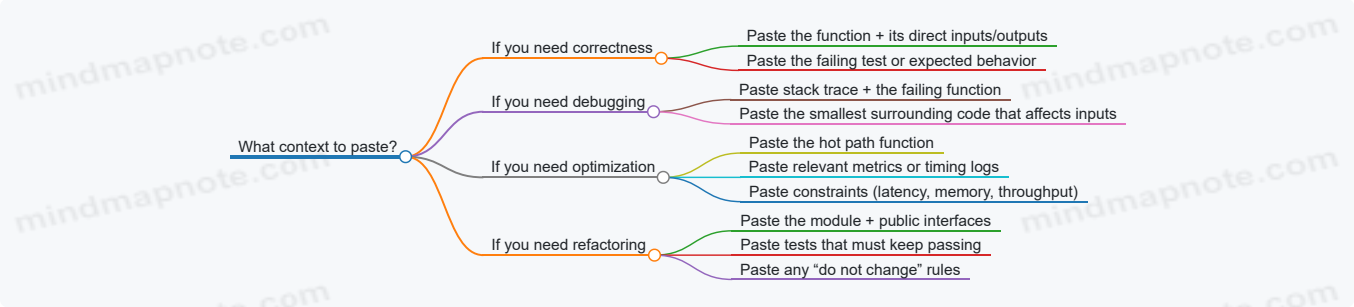
1. **One-paragraph summary** of the problem and what you observed.

2. **Relevant snippets** (files and logs) with clear labels.
3. **Constraints** as bullet points.
4. **A specific request**: diagnosis, patch, or test update.

Example prompt (debugging)

```
Problem: The /checkout endpoint returns 500 when the cart has 0 items.
Observed: Stack trace shows failure inside validate_cart(payload).

File: service/checkout.py
[PASTE validate_cart and the caller]

File: models/cart.py
[PASTE Cart and validation rules]

Log (sanitized):
[PASTE the stack trace and the 5 lines before/after]

Constraints:
- Keep the endpoint response schema unchanged.
- Return HTTP 400 with error code CART_EMPTY.
- Do not change how other validation errors are formatted.

Task:
1) Explain the cause.
2) Provide a minimal code patch.
3) Add/adjust a unit test for the 0-items case.
```

Mind map: how to decide what to paste



Common pitfalls (and how to avoid them)

- **Missing the “why”**: If you only paste code, ask the assistant to infer intent and you’ll get guesses. Add one sentence about expected behavior.
- **Conflicting context**: If logs say one thing and tests say another, call it out. The assistant can’t reconcile contradictions without guidance.
- **Unstated constraints**: If you care about error codes, performance, or compatibility, state it explicitly. Otherwise, the assistant may “fix” the symptom in a way you can’t accept.
- **No verification plan**: Always ask for a targeted test or a small verification step. Context without a check is just a story.

Quick checklist before you hit “send”

- Did I paste the smallest relevant code path?
- Did I include the exact error or observation?
- Did I state constraints that protect interfaces and behavior?
- Did I ask for a minimal patch plus a verification step?

When you do this consistently, the assistant becomes less of a guesser and more of a careful collaborator—one that works from your evidence instead of its own imagination.

2. Prompting for Correct Code Generation

2.1 Defining Inputs, Outputs, and Edge Cases in Prompts

A good prompt is a small specification. When you state inputs, outputs, and edge cases clearly, the assistant can generate code that fits your constraints instead of guessing. Think of it as telling the assistant what "done" means, not just what you want.

Inputs: what the assistant must assume

Inputs are the facts the assistant should treat as true. Include:

- **Data shape**: types, fields, and example values.
- **Constraints**: time limits, memory limits, allowed libraries, style rules.
- **Environment**: language version, runtime, framework, OS assumptions.
- **Known context**: existing functions, interfaces, and invariants.

Example (API handler):

- Input: `userId` is a string, always non-empty.
- Input: request body has `{ email: string }`.
- Constraint: must not query the database more than once.
- Environment: Node.js 20, TypeScript 5.

If you skip the data shape, the assistant may "helpfully" invent fields or accept invalid inputs, and your tests will catch it later.

Outputs: what the assistant must produce

Outputs define the deliverable and its format. Include:

- **Return type / response schema**: exact fields and types.
- **Side effects**: what it may write, log, or call.
- **Error behavior**: status codes, thrown exceptions, and messages.
- **Verification expectations**: tests to add or checks to run.

Example (function contract):

- Output: a function `normalizeEmail(email: string): string`.
- Output: returns lowercase email trimmed of whitespace.
- Error behavior: throws `TypeError` if input is not a string.

When outputs are explicit, the assistant can match your existing patterns instead of choosing its own.

Edge cases: where correctness usually breaks

Edge cases are the inputs that are valid but unusual, or invalid in ways that should be handled gracefully. Include categories, not just a random list.

Common edge-case categories:

- **Empty values**: `""`, empty arrays, missing optional fields.
- **Boundary values**: min/max lengths, numeric extremes.
- **Invalid types**: `null` where a string is expected.
- **Whitespace and formatting**: leading/trailing spaces, case differences.
- **Concurrency and ordering**: duplicate requests, out-of-order events.
- **Resource limits**: large payloads, pagination limits.

A practical approach is to write edge cases as test scenarios. If you can't imagine a test, you probably haven't defined the edge case clearly.

Mind map: prompt specification for code tasks

Prompt Specification Mind Map

Click here to view the mind map: Prompt Specification

Concrete prompt examples

Example 1: Email normalization

Prompt (good specification):

- Task: Write `normalizeEmail`.
- Inputs: `email` is a string that may include leading/trailing whitespace.
- Output: returns a string in lowercase with whitespace trimmed.
- Error behavior: if `email` is `""` after trimming, throw `RangeError`.
- Edge cases:
  - `" Alice@Example.COM "` → `"alice@example.com"`
  - `""` → `RangeError`
  - `" "` → `RangeError`
  - `null` or `123` → `TypeError`
- Verification: include 4 unit tests.

Why it works: the assistant knows the exact transformation and the exact failure modes.

Example 2: Debugging a failing function

Prompt (inputs + outputs + edge cases):

- Task: Fix `parsePrice`.
- Inputs: `parsePrice(input: string): number`.
- Current behavior: returns `NaN` for inputs like `"$1,234.50"`.
- Output: parse currency strings with commas and optional `$`.
- Error behavior: throw `Error("Invalid price")` for invalid formats.
- Edge cases:
  - `"$0"` → `0`
  - `"1,234"` → `1234`
  - `"$1,234.50"` → `1234.5`
  - `"-5"` → throw `Error("Invalid price")`
  - `"abc"` → throw `Error("Invalid price")`
- Verification: update tests and ensure all pass.

Why it works: you're not asking "make it correct"; you're defining correctness with examples.

Example 3: API endpoint contract

Prompt (request/response schema):

- Task: Implement `POST /v1/invitations`.
- Inputs:
  - Request JSON: `{ "email": string, "role": "viewer" | "editor" }`.
  - `email` may contain whitespace.
- Output:
  - Success: HTTP 201 with `{ "id": string, "email": string, "role": string }`.
  - Must return the normalized email.
- Error behavior:
  - 400 for invalid email format.
  - 409 if invitation already exists for the normalized email.
- Edge cases:
  - `role` missing → 400
  - `email` is `" "` → 400
  - `email` differs only by case → treated as same (409)
- Constraints: only one database lookup.

- Verification: provide tests for 201, 400, and 409.

Why it works: the assistant can align with your HTTP semantics and your normalization rules.

A reusable checklist you can paste into prompts

Use this structure when you want consistent results.

Task:

- What to do (implement/debug/refactor)

Inputs:

- Types and shapes:
- Constraints:
- Environment:
- Existing interfaces/invariants:

Outputs:

- Deliverable:
- Return/response schema:
- Side effects:
- Error behavior:

Edge cases (as test scenarios):

- Case 1: input → expected
- Case 2: input → expected
- Case 3: input → expected

Verification:

- Tests to add or update:
- Commands to run:

Common mistakes to avoid

- **Vague outputs**: "Make it work" instead of "Return `{id, email, role}` and throw `Error("Invalid price")` for bad formats."
- **Unspecified error behavior**: if you don't say what happens on failure, the assistant will pick something.
- **Edge cases without expectations**: listing "empty string" is less useful than stating `""` should throw `RangeError`.
- **Missing constraints**: if you care about "one database lookup," say it. Otherwise, the assistant may choose a simpler but slower approach.

When you define inputs, outputs, and edge cases as a compact contract, the assistant's code generation becomes more like implementation and less like guessing. That's the difference between "it compiles" and "it behaves correctly."

2.2 Specifying Language, Style, and Project Conventions

When you ask an AI coding assistant for code, you're not just requesting functionality—you're also requesting agreement on how the code should look, behave, and fit into the existing project. The fastest way to get useful output is to state the language and style rules explicitly, then anchor them with small examples from your codebase.

What to specify (and why it matters)

1. **Language and version**: "Python 3.11" or "TypeScript 5.x" changes standard library availability, typing patterns, and formatting norms.
2. **Style conventions**: naming rules, formatting rules, and error-handling patterns prevent the assistant from producing code that "works" but doesn't match your repo.
3. **Project conventions**: how you structure modules, where you place files, how you name functions, and how you handle configuration.
4. **Output constraints**: what the assistant should include (tests, comments, docstrings) and what it should avoid (extra dependencies, large rewrites).

A good prompt doesn't just say "write clean code." It says what "clean" means in your project.

Mind map: prompt ingredients for language, style, and conventions

Click here to view the mind map: Prompt ingredients: Language, Style, and Conventions

Language specification: make it concrete

Instead of: “Write code in Python.”

Use: “Write Python 3.11 code for a backend service. Use `typing` and `dataclasses` where appropriate. Avoid async unless the surrounding code is async.”

This reduces mismatches like:

- using features not available in your version,
- choosing sync vs async incorrectly,
- importing modules that don’t exist in your runtime.

Style specification: give the assistant a rule set it can follow

Style rules are easiest to apply when they’re phrased as constraints.

Example style constraints you can paste into a prompt

- “Use `snake_case` for functions and variables.”
- “Keep lines under 100 characters.”
- “Prefer early returns over deep nesting.”
- “Raise `ValueError` for invalid inputs; do not return `None`.”
- “Use `logging.getLogger(__name__)` and log with `logger.info` / `logger.exception`.”

If your project uses a formatter or linter, mention it. Even if the assistant can’t run it, the prompt can steer output toward compliance.

Project conventions: anchor with local patterns

Project conventions are often more important than general style. Two teams can both use “camelCase,” yet still differ on how they structure modules.

Good convention details to include

- Where code lives: “Add the function to `src/services/billing.py`.”
- How modules are organized: “Follow the existing `Repository` pattern.”
- How configuration is accessed: “Read settings from the existing `settings` object.”
- How errors are represented: “Use the existing `AppError` type.”

When possible, include a short snippet from the existing code that demonstrates the pattern.

Example: prompt with language + style + conventions

```
You are editing a Python 3.11 backend.
Follow these conventions:
- snake_case names
- lines <= 100 chars
- early returns
- raise ValueError for invalid inputs
- logging via logger = logging.getLogger(__name__)

Project conventions:
- Put code in src/services/inventory.py
- Use the existing InventoryRepository class
- Do not add new dependencies

Task:
Implement function reserve_items(order_id: str, sku_quantities: dict[str,int]) -> None.
It should validate quantities > 0, call repository.reserve(...), and log success.
Include a small unit test in tests/services/test_inventory.py.
```

Mind map: style rules you can turn into prompt constraints

Click here to view the mind map: Style rules to convert into prompt constraints

Examples of “style mismatch” and how to prevent them

1. **Naming mismatch**
   - Problem: assistant uses `reserveItems` in a snake_case repo.
   - Fix: explicitly state naming rules and include one example identifier from the repo.
2. **Error-handling mismatch**
   - Problem: assistant returns `False` instead of raising.
   - Fix: specify the exact exception type and when it should be thrown.
3. **Logging mismatch**
   - Problem: assistant uses `print()` or logs too much.
   - Fix: require `logger` usage and specify log levels.
4. **Dependency mismatch**
   - Problem: assistant adds a new library for a simple task.
   - Fix: “Do not add new dependencies; use existing utilities.”

A practical checklist you can reuse

Use this checklist as a final pass before sending the prompt.

- [ ] Language and version stated
- [ ] Target environment stated (server/CLI/etc.)
- [ ] Naming rules stated
- [ ] Formatting rules stated (line length, blank lines)
- [ ] Error-handling rules stated (exceptions vs returns)
- [ ] Logging rules stated (logger name and levels)
- [ ] Project file paths and module structure stated
- [ ] Existing pattern to follow included (snippet or description)
- [ ] Output requirements stated (diff vs full file, tests included)

Example: a compact “style block” you can paste

```
Style & conventions:
- Language: TypeScript 5.x
- Formatting: 2-space indent, single quotes
- Naming: camelCase for vars/functions, PascalCase for types
- Errors: throw Error for invalid inputs
- Docs: JSDoc for exported functions
- No new dependencies
Project:
- Edit src/utils/date.ts
- Follow existing helper patterns and export style
Output:
- Provide a unified diff
- Include/update unit tests if they exist for this module
```

When you specify these details, the assistant has fewer degrees of freedom. That usually means fewer revisions, cleaner diffs, and code that looks like it belongs to your repository rather than a generic template.

2.3 Requesting Tests Alongside Implementation

When you ask for code, you’re really asking for behavior. Tests are the part that makes that behavior measurable. The trick is to request tests at the same time as the implementation, so the assistant can align the code with the expected outcomes instead of guessing what you meant.

What to request (and why it works)

Ask for three layers of tests:

1. **A small "happy path" test** that proves the core behavior works.
2. **Edge-case tests** that define what happens at boundaries (empty input, missing fields, limits).
3. **A failure-mode test** that checks how the code responds to invalid inputs or errors.

This structure prevents a common failure mode: code that passes a single example but behaves incorrectly in real usage.

A mind map for test-aligned prompting

Test-aligned prompting mind map

Click here to view the mind map: Goal: tests that match the implementation

Prompt pattern you can reuse

Use a prompt that forces the assistant to treat tests as requirements.

Template (copy and adapt):

- Implement function/module: **[describe behavior precisely]**
- Language/tooling: **[language, test framework]**
- Constraints: **[performance, style, no external calls, etc.]**
- Provide tests:
  - At least **1 happy path**
  - At least **2 edge cases**
  - At least **1 failure mode**
- Tests must include:
  - Clear assertion messages
  - Deterministic inputs
  - Coverage of boundary conditions
- Output format:
  - **Tests first**, then implementation (or tests and implementation together)

The "Tests first" option is especially effective when the behavior is subtle. The assistant can't accidentally implement a different interpretation if the tests are already written.

Concrete example: a small function with clear boundaries

Suppose you need a function that normalizes a user-provided username.

Behavior requirements:

- Trim whitespace.
- Convert to lowercase.
- Remove internal spaces (turn `"a b"` into `"ab"`).
- If the result is empty, raise `ValueError`.

Prompt you can use:

"Write unit tests and implementation in Python for `normalize_username(name: str) -> str`. Requirements: trim, lowercase, remove internal spaces. If the normalized result is empty, raise ValueError. Include: 1 happy path, 2 edge cases, 1 failure mode. Use `pytest`. Write tests first, then implement to satisfy them."

Example tests (what you should expect to receive):

```python
import pytest
from your_module import normalize_username

def test_happy_path():
    assert normalize_username("  Alice Smith  ") == "alicesmith"

def test_edge_case_only_spaces():
    with pytest.raises(ValueError, match="empty"):
        normalize_username("   ")

def test_edge_case_mixed_whitespace_and_case():
    assert normalize_username("  BoB\t  b  ") == "bobb"

def test_failure_mode_none_input():
    with pytest.raises(TypeError):
        normalize_username(None)
```

Notice the tests do more than check outputs. They also define how invalid input should behave. If you want `None` to be treated differently (e.g., return empty string instead of raising), you should say so in the prompt.

Implementation that matches those tests:

```python
def normalize_username(name: str) -> str:
    if name is None:
        raise TypeError("name must be a string")
    normalized = "".join(name.strip().lower().split())
    if not normalized:
        raise ValueError("empty username")
    return normalized
```

Concrete example: requesting tests for a function with side effects

If your code writes files, updates a database, or calls external services, tests need to control those side effects.

Behavior requirements:

- `save_report(report_id, content)` writes `content` to `reports/{report_id}.txt`.
- It should create the directory if missing.

Prompt:

"Implement `save_report(report_id: str, content: str, base_dir: Path) -> Path`. Write tests using `pytest`. Constraints: no real filesystem outside `tmp_path`. Provide: 1 happy path, 2 edge cases (empty content, unusual report_id), 1 failure mode (base_dir not writable). Include assertions that verify the file path and file contents."

Example tests (using `tmp_path`):

```python
from pathlib import Path
import pytest
from your_module import save_report

def test_happy_path(tmp_path):
    p = save_report("r1", "hello", tmp_path)
    assert p == tmp_path / "reports" / "r1.txt"
    assert p.read_text() == "hello"

def test_edge_empty_content(tmp_path):
    p = save_report("r2", "", tmp_path)
    assert p.read_text() == ""

def test_edge_unusual_report_id(tmp_path):
    p = save_report("a-b_c", "x", tmp_path)
    assert p.name == "a-b_c.txt"

def test_failure_mode_unwritable_base_dir(tmp_path):
    unwritable = tmp_path / "nope"
    unwritable.mkdir()
    unwritable.chmod(0o400)
    with pytest.raises(PermissionError):
        save_report("r3", "y", unwritable)
```

The key detail is the prompt's constraint: "no real filesystem outside `tmp_path`." That forces the assistant to write tests that are safe and repeatable.

How to keep tests from becoming vague

When you ask for tests, also ask for **specific assertions**. Vague tests are worse than no tests because they give false confidence.

Add lines like:

- "Assert exact output values, not just truthiness."
- "Assert exception type and message (or at least type)."
- "Assert side effects (file contents, DB row count), not only return values."

A quick checklist for your prompt

Before you hit generate, confirm:

- You stated the behavior rules in plain language.
- You specified the test framework.
- You required at least 1 happy, 2 edge, 1 failure test.
- You asked for deterministic tests and controlled side effects.

If you do that, the assistant's implementation and tests will usually agree on what "correct" means. And that agreement is the whole point.

2.4 Handling Partial Implementations and Incremental Builds

AI coding assistants are great at producing a first draft quickly, but "done" is rarely the first draft. This section teaches you how to keep partial implementations from turning into a tangled mess, and how to build incrementally so every step is testable.

The core idea: make progress measurable

A partial implementation is useful when you can answer three questions:

1. **What is implemented?** (and what is not)
2. **How do we know it works?** (tests, checks, or observable behavior)
3. **What is the next smallest step?** (a change that reduces uncertainty)

When you prompt with these questions in mind, the assistant tends to produce code that can be integrated safely rather than "almost correct" in a way that breaks later.

A simple workflow for incremental builds

Use this loop for each feature or refactor:

1. **Define a thin vertical slice**: one path from input to output that exercises the feature end-to-end.
2. **Implement only the slice**: avoid filling in every edge case at once.
3. **Add a failing test first** (or a check that currently fails): this anchors the work.
4. **Make the smallest change that makes the test pass**.
5. **Repeat**: expand coverage and handle remaining cases.

This keeps the codebase in a state where you can run it, test it, and reason about it.

Mind map: partial implementations and incremental builds

Click here to view the mind map: Partial Implementations & Incremental Builds

How to prompt for partial implementations (without losing control)

A good prompt doesn't ask for "the full solution." It asks for a **bounded step**.

Use a structure like:

- **Current state**: what exists and what doesn't.
- **Target slice**: the smallest end-to-end behavior.
- **Constraints**: keep interfaces stable; don't change unrelated modules.
- **Definition of done for this step**: tests/checks that must pass.
- **Allowed placeholders**: what the assistant may stub.

Example prompt (feature slice)

You're adding a function `parse_user_id`.

Prompt:

- Implement only parsing for numeric IDs like `"user:123"`.
- If the format doesn't match, raise `ValueError`.
- Do not support UUIDs yet.
- Add unit tests for: `"user:123"` and `"user:abc"`.
- Keep the public function signature unchanged.

This forces the assistant to produce a partial implementation that is still correct for the slice.

Handling "not implemented yet" safely

Partial code often fails later because placeholders are ambiguous. The fix is to make placeholders **loud and local**.

Prefer explicit errors over silent placeholders

If a function is incomplete, return a value that looks valid and you'll get confusing downstream failures. Instead, raise an error that points to the missing behavior.

Bad placeholder (silent):

- Returning `None` where callers expect a string.
- Returning an empty list that looks like "no results."

Better placeholder (explicit):

- Raise `NotImplementedError` with a message that names the missing case.
- Or raise `ValueError` for invalid inputs, if that matches the contract.

Example: incremental parsing with explicit stubs (Python)

```python
import re

_USER_RE = re.compile(r"^user:(\d+)$")

def parse_user_id(s: str) -> str:
    """Return numeric user id as a string."""
    m = _USER_RE.match(s)
    if not m:
        raise ValueError("Unsupported user id format")
    return m.group(1)
```

When you later add UUID support, you'll replace the regex logic and expand tests. Until then, the behavior is consistent and failures are meaningful.

Incremental builds: keep the system runnable

Incremental builds are easiest when you avoid large integration jumps.

Strategy: integrate behind stable interfaces

If you're adding a new module, keep the old behavior as the default and route only one code path to the new implementation.

Example scenario: You're optimizing a function but want to keep the old version available.

- Step 1: Introduce a wrapper `compute_total()` that calls the old function.
- Step 2: Add a new implementation `compute_total_fast()`.
- Step 3: Switch the wrapper to call the fast version only after tests pass.

This reduces the blast radius of each change.

Tests as the "integration contract"

When the assistant provides partial code, tests tell you whether the partial behavior matches expectations.

Use contract-style tests for each slice

A contract test checks:

- input shape
- output shape
- error type and message (when relevant)

Example: contract tests for a parser (JavaScript/Jest)

```javascript
function parseUserId(s) {
  const m = /^user:(\d+)$/.exec(s);
  if (!m) throw new Error('Unsupported user id format');
  return m[1];
}

test('parses numeric user id', () => {
  expect(parseUserId('user:123')).toBe('123');
});

test('rejects non-numeric format', () => {
  expect(() => parseUserId('user:abc'))
    .toThrow('Unsupported user id format');
});
```

Even if you later expand formats, these tests ensure you don't break the original contract.

A practical "incremental build" checklist

Before you accept the assistant's partial implementation, verify:

- **Interfaces match**: function signatures and return types align with callers.
- **Errors are intentional**: incomplete cases fail with clear exceptions.
- **One slice works**: the feature path is end-to-end for at least one scenario.
- **Tests cover the slice**: at least one test fails before and passes after.
- **No hidden coupling**: changes don't require unrelated modules to be updated.

Common failure modes (and how to prevent them)

1. **Assistant implements too much at once**
   - Fix: ask for a single slice and a short definition of done.
2. **Placeholder behavior leaks into production paths**
   - Fix: raise explicit errors for unimplemented cases; keep placeholders unreachable except in tests.
3. **Incremental changes mix refactors and features**
   - Fix: separate commits so you can pinpoint what caused a new failure.
4. **Tests are missing for the integration boundary**
   - Fix: add a contract test at the boundary where the assistant's code meets existing code.

Example: turning a messy partial into a clean incremental step

Suppose the assistant generates a function `format_order(order)` but includes multiple formatting rules you didn't ask for.

What to do:

- Ask it to implement only the simplest rule: e.g., "include order id and total."
- Require tests for that rule.
- Ask it to leave other rules as explicit `NotImplementedError` or to omit them entirely.

This converts a broad, risky change into a controlled slice.

Summary

Partial implementations are not inherently bad; they're bad when they're ambiguous, untested, or integrated too broadly. Incremental builds solve this by making each step measurable: define a slice, implement it, add a contract test, and only then expand.

2.5 Verifying Generated Code With Simple Checks

When an assistant proposes code, your job is not to "trust it." Your job is to run a few checks that quickly answer: **does it compile, does it behave correctly on small inputs, and does it fail safely**. The trick is to keep checks small enough that you can do them every time, even when you're tired and the code is new.

A quick verification mindset

Start with a three-step ladder:

1. **Surface correctness**: Does it run and match the expected interface?
2. **Behavior correctness**: Does it produce correct outputs on representative inputs?
3. **Safety correctness**: Does it handle bad inputs and edge cases without surprises?

If a check fails, you don't need a full rewrite. You need a targeted fix and a repeat of the smallest relevant check.

Mind map: verification checklist

Mind map: Simple checks for generated code

Click here to view the mind map: Verify generated code

Surface checks (fast and boring, which is good)

1) Compile or type-check

If the code doesn't compile, everything else is wasted effort. Even in dynamic languages, a quick run of the smallest entry point is worth it.

Example (Python):

```
# generated: parse_user_id.py
def parse_user_id(s):
    return int(s)
```

A simple smoke check catches interface mismatches:

```
# smoke_test.py
from parse_user_id import parse_user_id
assert parse_user_id("42") == 42
```

Run it immediately. If it fails, fix the mismatch before writing more tests.

2) Lint for obvious issues

Linting won't prove correctness, but it catches common mistakes like unused variables, unreachable code, or suspicious comparisons.

Example (JavaScript):

```
// generated: normalizeEmail.js
export function normalizeEmail(email) {
  return email.trim().toLowerCase();
}
```

A linter might warn if `email` is sometimes undefined. That's not a proof, but it's a useful nudge to add a safety check.

3) Smoke test the smallest path

A smoke test should exercise the main function with one "normal" input. Keep it minimal so it runs in seconds.

Example (Go):

```
// generated: clamp.go
func Clamp(x, min, max int) int {
        if x < min { return min }
        if x > max { return max }
        return x
}
```

Smoke test:

```
package main

import "testing"

func TestClampSmoke(t *testing.T) {
        if Clamp(5, 1, 10) != 5 { t.Fatal("unexpected") }
}
```

Behavior checks (prove the logic, not the vibes)

1) Unit tests for core logic

Pick a few inputs that cover:

- typical cases
- boundaries
- one tricky case

Example (Python): Suppose the assistant generated a function to compute a discounted total.

```
def discounted_total(prices, discount_rate):
    return sum(p * (1 - discount_rate) for p in prices)
```

Unit tests:

```
def test_discounted_total_typical():
    assert discounted_total([10, 20], 0.1) == 27.0

def test_discounted_total_empty():
    assert discounted_total([], 0.2) == 0

def test_discounted_total_boundary_zero_discount():
    assert discounted_total([5], 0.0) == 5
```

These tests don't cover every possibility, but they cover the function's shape.

2) Invariants and property-style checks

Instead of enumerating every case, check rules that must always hold.

Example (Java): If you generated a function that returns a sorted list, you can check invariants.

- length is preserved
- output is non-decreasing
- output contains the same multiset of elements

Example (Python property-style without extra libraries):

```
def is_sorted(a):
    return all(a[i] <= a[i+1] for i in range(len(a)-1))

def test_sort_invariants(sort_fn):
    inp = [3, 1, 2, 2]
    out = sort_fn(inp)
    assert len(out) == len(inp)
    assert is_sorted(out)
    assert sorted(out) == sorted(inp)
```

This is still "simple checks," just with a different emphasis: rules over examples.

3) Golden tests for known inputs

When you have stable expected outputs (like formatting, parsing, or deterministic transformations), golden tests are efficient.

Example (formatting):

```
def format_money(amount_cents):
    return f"${amount_cents/100:.2f}"

def test_format_money_golden():
    assert format_money(0) == "$0.00"
    assert format_money(105) == "$1.05"
    assert format_money(-50) == "-$0.50"
```

Golden tests are especially good when the assistant might "almost" match the expected format.

Safety checks (edge cases that save you later)

1) Validate inputs explicitly

If the assistant generated a function that assumes valid inputs, add tests for invalid inputs and decide the expected behavior.

Example (Python):

```
def parse_user_id(s):
    return int(s)
```

Add safety expectations:

```
import pytest

def test_parse_user_id_rejects_non_numeric():
    with pytest.raises(ValueError):
        parse_user_id("abc")

def test_parse_user_id_rejects_none():
    with pytest.raises(TypeError):
        parse_user_id(None)
```

If you prefer a custom error, update the function and keep the tests as the contract.

2) Boundary conditions

Boundaries are where off-by-one bugs hide.

Example (indexing): If you generated a function that returns the first element greater than a threshold:

- threshold below min
- threshold equal to an element
- threshold above max

Even without property testing, three boundary tests often catch the majority of mistakes.

3) Error handling paths

If the code includes error handling, test at least one failure path.

Example (TypeScript):

```
export function divide(a: number, b: number) {
  if (b === 0) throw new Error("division by zero");
  return a / b;
}
```

Test the failure path:

```
test("divide by zero throws", () => {
  expect(() => divide(1, 0)).toThrow("division by zero");
});
```

A practical “verification script” you can reuse

Use a consistent sequence so you don’t reinvent the process.

Click here to view the mind map: Verification script (repeatable)

Mind map: what to check first when time is short

Mind map: triage when verification time is limited

Click here to view the mind map: Time is short

Example workflow: from generated code to confidence

1. Assistant generates `normalizeEmail(email)`.
2. You run a smoke test: `" A@B.COM " -> "a@b.com"`.
3. You add boundary tests: empty string, already-normalized string.
4. You add a safety test: `null` or `undefined` should either throw a clear error or return a defined fallback.
5. You re-run tests after any change.

That's it. You're not proving everything. You're proving enough to move forward without stepping on the same rake twice.

3. Debugging With AI Assistants Using Real Errors

3.1 Turning Stack Traces Into Actionable Debug Steps

A stack trace is a breadcrumb trail with timestamps missing. Your job is to turn it into a sequence of checks that either confirm the cause or narrow the search fast. The trick is to treat the trace as structured data: where the error happened, what call chain led there, and which inputs were likely involved.

Step 1: Identify the first "real" failure

Many logs include wrapper frames from frameworks, test runners, or middleware. Start by locating the line that actually throws the exception (or the first line that mentions the error type). If you see multiple exceptions, begin with the earliest one in time or the one that is not "caused by" another.

Example (Python):

- Error: `TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'`
- Trace points to: `total = price + tax`

Actionable takeaway: the value of `tax` is `None` at the moment of the addition. That's not a "mystery"; it's a specific contract violation.

Step 2: Extract the three key facts

Write these down as you read:

1. **Error type** (what category of failure): `TypeError`, `IndexError`, `KeyError`, `ValueError`, etc.
2. **Location** (file + line): where the failing statement lives.
3. **Call chain** (who called whom): the path that produced the inputs.

This turns reading into a checklist. You're not trying to memorize the trace; you're trying to reproduce the conditions.

Example (JavaScript):

- `TypeError: Cannot read properties of undefined (reading 'id')`
- Location: `user.id` in `buildPayload(user)`
- Call chain: `handleRequest -> buildPayload -> ...`

Actionable takeaway: `user` is undefined (or missing `id`) when `buildPayload` runs. The call chain tells you where `user` should have been created.

Step 3: Map frames to hypotheses

Each frame suggests a hypothesis about what went wrong: wrong argument, wrong state, wrong ordering, or wrong assumption.

Use a simple rule:

- If the failing line uses a variable, the hypothesis is about how that variable was computed.

- If the failing line indexes into a collection, the hypothesis is about bounds or missing keys.
- If the failing line parses or converts, the hypothesis is about input format.

Mind map: Stack trace to debug steps

Click here to view the mind map: Stack Trace → Actionable Debug Steps

Step 4: Add evidence at the boundary, not everywhere

A common mistake is to sprinkle logs across the whole call chain. Instead, place one or two checks where the trace implies the contract breaks.

Example (TypeScript): Failing line: `const id = user.id;`

Add a check right before the failing function is called:

- If `handleRequest` is the caller, validate `user` there.
- If `fetchUser` is responsible for building `user`, validate there.

You want the earliest point where the value can still be corrected.

Concrete pattern:

- Replace "hope it's not null" with an explicit guard.
- Use an assertion that includes the relevant identifiers.

Step 5: Turn the call chain into a minimal repro

If you can't reproduce the failure with a small input, you're debugging the environment rather than the code. Use the call chain to identify the smallest set of inputs that reach the failing line.

Example (C#):

- `NullReferenceException` at `order.Customer.Name`
- Call chain: `GetOrder -> MapOrder -> ...`

Minimal repro approach:

1. Create a sample `order` object that reaches `MapOrder`.
2. Ensure `Customer` is null to see if the same line fails.
3. If it only fails with a specific mapping condition, focus on that mapping step.

Even if you don't know the exact input yet, you can build a harness that calls the same function with controlled values.

Step 6: Use invariants to narrow the search

Invariants are statements that should always be true at a given point. The stack trace tells you where an invariant was violated.

Example (Python): Failing line: `items[i]` with `IndexError: list index out of range`

Invariant candidates:

- `0 <= i < len(items)`
- `items` is non-empty

Add an assertion near where `i` is computed, not inside the loop body. If `i` is derived from user input, validate it at the boundary.

Mind map: Evidence placement

Click here to view the mind map: Evidence Placement

Step 7: Handle common trace patterns

Pattern A: "Cannot read property of undefined"

This usually means a missing object, not a missing property. The fix is to find where the object should have been created.

Action steps:

- Identify the variable being dereferenced.
- Trace backward to the assignment or return value.
- Add a guard at the point of return (or before dereference).

Pattern B: “KeyError” / “undefined index”

This often means the code assumed a key exists. The call chain tells you which input source should provide it.

Action steps:

- Check whether the key is optional.
- If optional, handle the absence explicitly.
- If required, validate earlier and fail with a clearer message.

Pattern C: Async stack traces that look scrambled

Async frameworks can show the call chain across scheduling boundaries. Treat the trace as “where it blew up” plus “which async task led there.”

Action steps:

- Identify the async boundary (await, promise, task spawn).
- Confirm that the awaited value is what you think it is.
- Add a check right after the await, before using the result.

Step 8: Convert the fix into a testable contract

Once you know the cause, write a small test that locks in the behavior. The goal is not to test the stack trace; it’s to test the contract that the trace revealed.

Example: If the failure was caused by `tax` being `None`, decide the contract:

- Either `tax` must never be null (then validate and throw a clear error),
- or `tax` may be null (then define a default like `0`).

Then add a test for that contract. The next time the trace appears, it should be because the contract was intentionally violated.

Quick checklist you can reuse

- [ ] Find the first meaningful exception line.
- [ ] Note error type, file:line, and the failing statement.
- [ ] Identify the variables used on that line.
- [ ] Map each call-chain frame to a hypothesis about those variables.
- [ ] Add one or two checks at the earliest correction point.
- [ ] Build a minimal repro harness or test.
- [ ] Fix the contract and lock it in with a test.

When you follow these steps, the stack trace stops being a wall of text and becomes a structured path: failure point → violated assumption → evidence → corrected contract.

3.2 Diagnosing Common Runtime Errors With Examples

Runtime errors are the ones that show up after the code has started running: exceptions, crashes, timeouts, and “it works on my machine” surprises. The goal here is not to guess the fix, but to narrow the cause quickly using evidence you already have: the error message, the stack trace, and a minimal reproduction.

A practical diagnosis workflow

1. **Confirm the failure mode**: Is it a crash (exception), a hang (no response), or incorrect output (wrong results)?
2. **Read the first “useful” line**: In many stack traces, the top frames are framework noise. The first line that mentions your module, function, or file is usually the best starting point.
3. **Map the error to a category**: Type error, null/undefined, index out of range, permission, network, database, or concurrency.
4. **Check inputs at the boundary**: Most runtime errors come from unexpected data entering a function.

5. **Reduce to a minimal example**: If you can reproduce with a tiny input, you can reason about it.

Mind map: common runtime error categories

Click here to view the mind map: Runtime Errors

Example 1: TypeError from unexpected input shape

Symptom

You see something like:

- `TypeError: unsupported operand type(s) for +: 'int' and 'str'`

Likely cause

A value that should be numeric arrives as a string, often from JSON, environment variables, or form fields.

Example code (Python)

```
def total_price(items):
    return sum(item["price"] + item["tax"] for item in items)

items = [
    {"price": 10, "tax": "2"},
]
print(total_price(items))
```

Diagnosis steps

- The error says `int` plus `str`, so at least one `tax` is a string.
- Print or log the types at the boundary:

```
for item in items:
    print(type(item["price"]), type(item["tax"]), item)
```

Fix pattern

Convert and validate at the boundary, not deep inside the loop.

```
def total_price(items):
    total = 0
    for item in items:
        price = float(item["price"])
        tax = float(item["tax"])
        total += price + tax
    return total
```

Why this works

The conversion makes the function's expectations explicit. If conversion fails, you get a clear error at the point of bad data.

Example 2: KeyError / missing field in dictionaries

Symptom

- `KeyError: 'user_id'`

Likely cause

A request payload or database row is missing a field you assumed existed.

Example code (Python)

```
def build_event(payload):
    return {
        "user_id": payload["user_id"],
        "action": payload["action"],
    }

payload = {"action": "login"}
print(build_event(payload))
```

Diagnosis steps

- The stack trace points to `payload["user_id"]`.
- The payload shown in logs or the debugger confirms the missing key.

Fix pattern

Use explicit validation with a helpful message.

```
def build_event(payload):
    if "user_id" not in payload:
        raise ValueError(f"Missing required field: user_id. Got keys: {list(payload.keys())}")
    if "action" not in payload:
        raise ValueError("Missing required field: action")
    return {"user_id": payload["user_id"], "action": payload["action"]}
```

Why this works

You replace a cryptic runtime exception with a targeted one that tells you what to fix: the input.

Example 3: IndexError from empty lists

Symptom

- `IndexError: list index out of range`

Likely cause

You access `arr[0]` or `arr[-1]` without checking whether the list has elements.

Example code (JavaScript)

```
function firstTag(tags) {
  return tags[0].toLowerCase();
}

console.log(firstTag([]));
```

Diagnosis steps

- `tags[0]` is `undefined` when the array is empty.
- Calling `.toLowerCase()` on `undefined` causes the crash.

Fix pattern

Handle the empty case explicitly.

```
function firstTag(tags) {
  if (!Array.isArray(tags) || tags.length === 0) return "";
  return String(tags[0]).toLowerCase();
}
```

Why this works

You define what “no tags” means. Returning an empty string is a choice; the important part is that the behavior is deliberate.

Example 4: ValueError from parsing formats

Symptom

- `ValueError: time data '2026/03/24' does not match format '%Y-%m-%d'`

Likely cause

A date string uses a different separator or ordering than your parser expects.

Example code (Python)

```
from datetime import datetime

s = "2026/03/24"
dt = datetime.strptime(s, "%Y-%m-%d")
print(dt)
```

Diagnosis steps

- The error message includes the exact input string and the expected format.
- Compare the two: `2026/03/24` uses `/`, while the format expects `-`.

Fix pattern

Either normalize the string or accept multiple formats.

```
from datetime import datetime

def parse_date(s):
    s = s.strip().replace("/", "-")
    return datetime.strptime(s, "%Y-%m-%d")
```

Why this works

Normalization makes the parser’s job consistent.

Example 5: Network timeouts and “it hangs”

Symptom

- Requests stall until a timeout.
- Logs show no progress past a network call.

Likely cause

DNS issues, unreachable hosts, slow servers, or missing timeouts.

Example code (Python)

```
import requests

r = requests.get("https://example.com/api")
print(r.status_code)
```

Diagnosis steps

- If there’s no timeout, the program may wait a long time.
- Add a timeout and log the target URL and elapsed time.

Fix pattern

```python
import requests
import time

start = time.time()
try:
    r = requests.get("https://example.com/api", timeout=5)
    print("status", r.status_code, "elapsed", time.time() - start)
except requests.exceptions.Timeout:
    print("timeout after", time.time() - start)
```

Why this works

You turn a hang into a bounded failure with a clear category.

Mind map: turning evidence into a fix

Click here to view the mind map: Diagnose Runtime Error

Quick checklist you can apply immediately

- Does the error message mention a specific type, key, index, or format? Use that exact detail.
- Does the stack trace point to your code or only to framework internals? Start with your first frame.
- Are inputs coming from JSON, env vars, or user forms? Treat them as untrusted until validated.
- If the program hangs, do you have timeouts around external calls?
- After the fix, can you run a targeted test or a small reproduction to confirm the behavior changed?

When you follow this pattern, “mystery errors” become predictable: each one maps to a small set of causes, and each cause has a straightforward fix strategy.

3.3 Debugging Logic Bugs With Minimal Reproducible Cases

Logic bugs are the ones that compile, run, and still do the wrong thing. The fastest way to debug them is to shrink the problem until you can explain the failure with a tiny input and a single expected-vs-actual mismatch. This section shows how to build a minimal reproducible case (MRC), then use it to reason about the code path without guessing.

What “minimal” means in practice

A minimal reproducible case has three properties:

- **Reproducible:** the bug happens every time.
- **Minimal:** remove everything that doesn’t affect the wrong behavior.
- **Specific:** the case targets one behavior (one function, one rule, one branch).

A good MRC often fits on one screen: a small snippet, a single test, or a short script that prints one wrong value.

Mind map: building and using an MRC

Mind map: Minimal Reproducible Cases for Logic Bugs

Click here to view the mind map: Goal: explain wrong behavior with smallest input

A concrete workflow (with examples)

1) Start from the failing symptom

Suppose you have a function that calculates a discounted total. Users report that totals are too low when the quantity is exactly 5.

You might currently have a large integration test. Instead, write down the smallest failing input you can find:

- Input: `price = 10`, `qty = 5`
- Actual: `40`

- Expected: `45`

That “exactly 5” detail is already a clue: it’s likely a boundary condition like `>` vs `>=`.

2) Reduce to a minimal test

Here’s a minimal reproduction in JavaScript. The bug is intentionally included so you can see the shape of the MRC.

```
function totalWithDiscount(price, qty) {
  let total = price * qty;
  // Bug: discount should apply at qty >= 5, not > 5
  if (qty > 5) {
    total = total * 0.9;
  }
  return total;
}

// Minimal reproducible case
console.log(totalWithDiscount(10, 5)); // actual: 50
console.log(totalWithDiscount(10, 6)); // actual: 54
```

Now convert it into an assertion-style test so it fails loudly.

```
function assertEqual(actual, expected, label) {
  if (actual !== expected) {
    throw new Error(`${label}: expected ${expected}, got ${actual}`);
  }
}

assertEqual(totalWithDiscount(10, 5), 45, 'qty=5 should discount');
```

This is minimal because it removes everything except the decision rule.

3) Reason about the branch condition

With the MRC in place, you can inspect the condition directly. The expected behavior says discount applies at 5. The code uses `qty > 5`, so the boundary case fails.

Fixing it is straightforward:

```
function totalWithDiscount(price, qty) {
  let total = price * qty;
  if (qty >= 5) {
    total = total * 0.9;
  }
  return total;
}
```

The important part is not the fix itself; it’s that the MRC makes the reasoning local. You don’t need to reproduce the whole app to confirm the logic.

Another example: wrong indexing in a loop

Consider a function that finds the first pair of numbers that sums to a target. It returns a pair, but sometimes it skips valid answers.

A minimal reproduction should isolate the loop and the data structure.

```
def find_pair(nums, target):
    for i in range(len(nums)):
        for j in range(i+1, len(nums)-1):
            if nums[i] + nums[j] == target:
                return (nums[i], nums[j])
    return None

print(find_pair([1, 2, 3, 4], 7))  # actual: (None)
```

The bug is subtle: `range(i+1, len(nums)-1)` stops one element early. The minimal case `[1,2,3,4]` with `target=7` should find `(3,4)`, but the loop never checks `j=3` when `i=2`.

Turn it into a test:

```
def test_find_pair():
    assert find_pair([1, 2, 3, 4], 7) == (3, 4)

test_find_pair()
```

Fix the loop bound:

```
def find_pair(nums, target):
    for i in range(len(nums)):
        for j in range(i+1, len(nums)):
            if nums[i] + nums[j] == target:
                return (nums[i], nums[j])
    return None
```

Again, the MRC keeps the reasoning focused on the loop bounds, not on how the function is called.

How to shrink without breaking the bug

When you remove code, you risk removing the very condition that triggers the logic error. Use these guardrails:

- **Keep the same inputs and types.** If the bug depends on `0` vs `"0"`, don't "simplify" by converting types.
- **Preserve boundary values.** If the failure happens at `qty=5`, keep that exact value.
- **Keep ordering.** Some logic bugs depend on iteration order or sorting.
- **Replace dependencies with deterministic stubs.** If a function depends on a service, stub it with a fixed return that matches the failing scenario.

A practical trick: while reducing, keep the failing assertion and run it after each removal. If the test stops failing, you removed something essential; undo the last change.

Using invariants to confirm your theory

Once you have an MRC, you can add small checks that describe what must be true at each step. For example, if a function builds a list of selected items, an invariant might be "the list length never exceeds the requested count."

Invariants don't replace debugging, but they help you locate where the logic first becomes inconsistent.

Example invariant checks:

- Before a discount is applied: `total == price * qty`
- After applying a 10% discount: `total == price * qty * 0.9`

If the invariant fails, you know the bug is earlier than your current suspicion.

Checklist for a strong MRC

- [ ] One failing input that demonstrates the wrong behavior
- [ ] Expected value written explicitly

- [ ] Minimal code path: no unrelated I/O
- [ ] A repeatable test or assertion
- [ ] A clear hypothesis tied to a specific line or condition

A minimal reproducible case turns logic debugging from “searching for what might be wrong” into “verifying which assumption is false.”

3.4 Using AI to Suggest Instrumentation and Logging

Good instrumentation is boring in the best way: it answers specific questions with minimal noise. When you ask an AI assistant for logging suggestions, you get the most value by steering it toward *observability goals* (what you need to know) rather than toward “add logs everywhere.”

What to ask for (and what to avoid)

Ask for:

- **A short list of questions** your logs should answer (e.g., “Why did this request fail?” “Which dependency timed out?”).
- **Log points** tied to those questions (entry, exit, error boundaries, and key state changes).
- **A logging level plan** (INFO for high-signal events, DEBUG for details you can turn on, WARN for recoverable issues, ERROR for failures).
- **Structured fields** (IDs, durations, counts, and error codes) so logs are searchable.

Avoid:

- “Add logging to this function.” Instead, specify *what you want to learn*.
- “Make it verbose.” Instead, specify *what would be actionable*.

Mind map: instrumentation strategy

Instrumentation and Logging Mind Map

Click here to view the mind map: Instrumentation and Logging

A practical prompt template

Use a prompt like this, adapted to your stack:

> You are reviewing my code for instrumentation. My goal is to answer: (1) why requests fail, (2) where time is spent, and (3) whether processing is complete. Here is the code and the expected behavior. Suggest: log points, log levels, and structured fields. For each suggestion, explain what question it answers and show a small code example.

If you include a sample request and a sample failure, the assistant can propose fields that actually match your reality.

Example: instrumenting an API handler

Suppose you have a handler that validates input, calls a service, and returns a response. Without instrumentation, failures look like “500 Internal Server Error.” With targeted logs, you can answer: which stage failed and how long each stage took.

Before (simplified):

- Parse JSON
- Validate
- Call `createOrder`
- Return result

After (structured logging idea):

- Log at the handler boundary with a correlation ID.
- Measure durations for parse/validate/service call.
- Log validation failures as WARN with field-level error counts.
- Log service failures as ERROR with an error code and dependency name.

Here’s a compact example in JavaScript/TypeScript style:

```
const start = Date.now();
logger.info({ event: "order.create.start", requestId }, "start");

try {
  const payload = req.body;
  logger.debug({ event: "order.create.parse", requestId }, "parsed body");

  const validation = validate(payload);
  if (!validation.ok) {
    logger.warn({
      event: "order.create.validation_failed",
      requestId,
      errorCount: validation.errors.length,
    }, "validation failed");
    return res.status(400).json({ errors: validation.errors });
  }

  const t0 = Date.now();
  const result = await createOrder(payload);
  logger.info({
    event: "order.create.service_ok",
    requestId,
    serviceMs: Date.now() - t0,
    orderId: result.id,
  }, "service succeeded");

  return res.status(201).json(result);
} catch (err: any) {
  logger.error({
    event: "order.create.error",
    requestId,
    errorCode: err.code,
    message: err.message,
    totalMs: Date.now() - start,
  }, "handler failed");
  return res.status(500).json({ error: "internal_error" });
}
```

Notice what's *not* logged:

- No full request body (it may contain sensitive data).
- No per-item logs inside loops (not shown here, but the same rule applies).
- No duplicate "start/end" messages at every layer; the handler boundary already provides correlation.

Example: instrumenting a background job

Background jobs often fail silently because there's no request context. The logging goal becomes: "Which job instance failed, and what step did it fail at?"

A good pattern is:

- Log job start with `jobId` and `attempt`.
- Log each major step with a step name and duration.
- Log a single summary at the end (success or failure) with counts.

```
start = time.time()
logger.info({"event": "job.start", "jobId": job_id, "attempt": attempt})

try:
    t0 = time.time()
    items = fetch_items()
    logger.info({"event": "job.step", "jobId": job_id, "step": "fetch_items", "ms": int((time.time()-t0)*1000), "count": len(items

    t1 = time.time()
    processed = process_items(items)
    logger.info({"event": "job.step", "jobId": job_id, "step": "process_items", "ms": int((time.time()-t1)*1000), "processed": pro

    logger.info({"event": "job.end", "jobId": job_id, "status": "ok", "totalMs": int((time.time()-start)*1000)})
except Exception as e:
    logger.error({"event": "job.end", "jobId": job_id, "status": "error", "error": str(e), "totalMs": int((time.time()-start)*1000
    raise
```

This gives you a timeline without drowning you in detail.

Using AI to propose log fields (with guardrails)

When you ask for fields, require the assistant to justify each one:

- **Correlation fields:** `requestId`, `jobId`, `traceId`.
- **Timing fields:** `ms`, `durationMs`.
- **Outcome fields:** `status`, `errorCode`.
- **Counts:** `errorCount`, `processed`, `itemsFetched`.

Add explicit constraints in your prompt:

- "Do not include raw payloads."
- "Cap string lengths to 200 characters."
- "Never log secrets or authorization headers."

A helpful follow-up prompt is:

> Propose a redaction rule for any field you want to log. Show the rule and where it's applied.

Turning logs into actionable debugging

Instrumentation should reduce time-to-answer. A good AI suggestion includes a "what you'll do with it" note, such as:

- If you see `serviceMs` spikes, you know where to look.
- If `validation_failed` includes `errorCount`, you can distinguish user mistakes from system bugs.
- If `job.end status=error` includes `errorCode`, you can group failures.

Mind map: log quality checklist

Click here to view the mind map: Log Quality Checklist

A short workflow you can reuse

1. Write the questions you want answered.
2. Provide the code and a sample success/failure.
3. Ask the assistant for log points, levels, and structured fields.
4. Add guardrails (redaction, caps, no hot-path spam).
5. Review the suggestions by checking whether each log line earns its place.

When you do this, the assistant becomes a fast reviewer for instrumentation decisions, not a generator of random log lines.

3.5 Confirming Fixes With Targeted Test Runs

A fix is only "confirmed" when you can point to evidence: tests that fail before the change and pass after it, ideally with minimal collateral changes. Targeted test runs keep feedback fast and focused, which matters because debugging often involves several iterations.

The goal: prove the fix, not just silence failures

When you run tests broadly, you might pass by accident while missing a regression. Targeted runs aim to answer two questions:

1. **Did the failing behavior stop happening?**
2. **Did the fix avoid breaking nearby behavior?**

A good workflow is: reproduce → implement fix → run the smallest set of tests that cover the failing path → expand only if needed.

Mind map: what to run and why

Mind map: Confirming fixes with targeted test runs

Click here to view the mind map: Confirm the fix

Step 1: Identify the smallest test set that should change

Start from the failure you saw earlier. Use the stack trace to find the module or function involved, then locate tests that mention that module.

Example (Python/pytest):

- You previously saw a failure in `billing/discounts.py:apply_discount()`.
- You search for tests referencing `apply_discount` or `discounts`.

You might find:

- `tests/unit/test_discounts.py::test_apply_discount_percentage`
- `tests/unit/test_discounts.py::test_apply_discount_minimum_price`

Run only that file first:

```
pytest -q tests/unit/test_discounts.py
```

If you want even tighter feedback, run just the failing test:

```
pytest -q tests/unit/test_discounts.py::test_apply_discount_percentage
```

This is the "confirm the fix" moment: the specific test that failed should now pass.

Step 2: Use a "before/after" mindset

If you can, capture the failing test output before changing code. Then after the fix, rerun the same command.

Why this matters: sometimes a change accidentally alters the test setup or expectation. If the test output changes drastically, you want to confirm you're still testing the same behavior.

Example:

- Before fix: `assert expected_total == actual_total` fails with a mismatch.
- After fix: the test passes, but another assertion inside the same test now fails in a different way.

That second failure indicates you may have fixed one symptom while exposing another issue.

Step 3: Confirm the fix with targeted boundary tests

A fix often changes how edge cases are handled. After the failing test passes, run additional tests that cover the boundaries around the failing input.

Example scenario: `apply_discount()` had a bug when the discount produced a negative intermediate value.

You might add or run tests for:

- zero discount
- maximum discount
- discount that would exceed the subtotal
- rounding behavior (e.g., cents)

Even if you don't add new tests yet, running existing boundary tests helps ensure the fix didn't just "fit" the one case.

Example (pytest):

```
pytest -q tests/unit/test_discounts.py -k "apply_discount and (zero or max or rounding)"
```

The `-k` filter keeps the run focused while still checking nearby logic.

Step 4: Expand scope only when the failure mode suggests it

Not every fix needs integration coverage. Expand when the failing behavior crosses boundaries.

Use this simple rule:

- **Pure logic fix** (string formatting, arithmetic, parsing): start with unit tests.
- **Behavioral fix** (HTTP handler, database query, message processing): include integration tests.

Example:

- If the bug was in a controller that builds a response, unit tests might pass but the integration test could still fail due to serialization or schema mismatch.

So after unit tests pass, run the relevant integration test file:

```
pytest -q tests/integration/test_billing_api.py
```

Step 5: Interpret failures in a way that guides the next change

When a targeted run fails, treat the output as a map, not a verdict.

Common patterns and what they mean:

1. **Same test fails, same assertion, different line numbers**
    - The fix didn't reach the intended code path.
    - Check whether the input shape matches the original failing case.
2. **Same test fails, different assertion message**
    - You may have changed behavior in the right area, but not fully.
    - Compare expected vs actual values to see the direction of error.
3. **Different test fails in the same file**
    - Your fix likely affected shared logic.
    - Re-run the whole file to see the full set of impacted cases.
4. **Flaky failure**
    - Rerun the same targeted command.
    - If it passes on rerun, log the conditions (time, ordering, random seeds) and ensure your fix doesn't depend on them.

Step 6: Keep test runs deterministic and fast

Targeted runs should be quick enough that you can repeat them often. If a focused test still takes minutes, it's not targeted anymore.

Example improvements:

- Reduce reliance on external services by using mocks for unit tests.
- Use fixed timestamps and seeded randomness.
- Avoid tests that depend on global state without cleanup.

Even without changing the test suite, you can run with options that reduce noise and speed feedback.

Example (pytest):

```
pytest -q --maxfail=1 tests/unit/test_discounts.py::test_apply_discount_percentage
```

`--maxfail=1` stops after the first failure, which is useful when you're iterating on a single bug.

A concrete mini-workflow you can reuse

1. Run the single failing test.
2. Apply the fix.
3. Re-run the same single test.
4. Run the full test file for the affected module.
5. Run boundary-related tests (filters or explicit cases).
6. If the fix touches boundaries (API/DB/serialization), run the relevant integration test file.

Example command sequence (pytest):

```
pytest -q tests/unit/test_discounts.py::test_apply_discount_percentage
pytest -q tests/unit/test_discounts.py
pytest -q tests/unit/test_discounts.py -k "apply_discount and (zero or max or rounding)"
pytest -q tests/integration/test_billing_api.py
```

This sequence confirms the fix with increasing confidence while keeping each step small enough to be actionable.

Quick checklist for "confirmed"

- The original failing test now passes.
- At least one nearby boundary test passes.
- No new failures appear in the affected unit test file.
- If the fix crosses boundaries, the relevant integration test(s) pass too.

When these conditions hold, you can move on with confidence that the fix solved the problem you actually observed, not just the one test you happened to run.

4. Code Review and Refactoring With AI

4.1 Requesting Reviews Focused on Readability and Maintainability

A good readability-focused review answers two questions: "Can a teammate understand this quickly?" and "Can they safely change it later?" When you ask for that kind of review, you get better feedback than generic "looks fine" comments.

What to ask for (so reviewers know what "good" means)

Use a short checklist in your review request. Include the scope, the target audience, and the acceptance criteria.

Example review request (copy/paste):

> Please review this change for readability and maintainability. Assume the reader is a new teammate who knows the domain but not this module.
>
> Focus on:
>
> 1. Naming clarity (variables, functions, types)
> 2. Control flow (early returns vs deep nesting)
> 3. Duplication (can any logic be centralized safely?)
> 4. Boundaries (what belongs in this function vs a helper?)
> 5. Error handling consistency (messages, types, and where errors are created)
> 6. Test readability (are tests easy to understand and extend?)
>
> If you suggest changes, prefer small, behavior-preserving edits.
>
> Reply with: (a) top 3 issues, (b) suggested rewrite for one key section, (c) any "future bug magnets" you notice.

That structure nudges the reviewer toward actionable feedback instead of opinions.

Mind map: what readability and maintainability reviews actually check

Click here to view the mind map: Readability & Maintainability Review

How to frame the code context (so feedback is specific)

Reviewers need enough context to judge readability choices. Provide:

- **The "why"**: one sentence about the goal of the change.
- **The constraints**: performance limits, API contracts, or style rules.
- **The intended usage**: who calls this code and how.
- **The risk tolerance**: whether refactoring is allowed or only small edits.

Example context block:

> Goal: make `parseOrder()` handle missing optional fields without changing existing outputs. Constraints: keep the public signature unchanged; behavior must match current tests. Risk tolerance: small refactors are fine if tests pass.

When you include constraints, reviewers stop proposing changes that violate your boundaries.

Concrete examples of readability issues (and what to ask for)

1) Naming: "what does this represent?"

Before (hard to scan):

```
function calc(a: number, b: number) {
  const x = a / b;
  const y = x * 100;
  return y;
}
```

After (clear intent):

```
function calcPercent(numerator: number, denominator: number) {
  const ratio = numerator / denominator;
  const percent = ratio * 100;
  return percent;
}
```

In your request, ask the reviewer to check whether names encode meaning: units, formats, and whether values are raw vs derived.

2) Control flow: reduce nesting and make the happy path obvious

Before (deep nesting):

```
def handle(req):
    if req.user:
        if req.user.is_admin:
            return do_admin(req)
        else:
            return do_user(req)
    else:
        return unauthorized()
```

After (early returns):

```
def handle(req):
    if not req.user:
        return unauthorized()
    if req.user.is_admin:
        return do_admin(req)
    return do_user(req)
```

Ask reviewers to point out where early returns would make the logic easier to follow.

3) Duplication: repeated predicates and “almost the same” blocks

Before (two similar branches):

```
if (status == Status.ACTIVE) {
  if (role == Role.ADMIN) {
    return true;
  }
  return false;
}
if (status == Status.ACTIVE) {
  return role == Role.MANAGER;
}
```

After (centralize the shared condition):

```
if (status != Status.ACTIVE) return false;
return role == Role.ADMIN || role == Role.MANAGER;
```

In your request, explicitly ask: “Where do you see repeated conditions or repeated transformations that could be centralized without changing behavior?”

4) Boundaries: keep functions small and outcomes named

Before (one function does everything):

```
func BuildReport(input Input) Report {
  // validate
  // transform
  // compute totals
  // format strings
  // attach metadata
}
```

After (separate steps by outcome):

```
func BuildReport(input Input) Report {
  validated := validateInput(input)
  shaped := shapeData(validated)
  totals := computeTotals(shaped)
  formatted := formatReport(totals)
  return attachMetadata(formatted, validated)
}
```

Ask reviewers to identify where a function is doing multiple jobs, and suggest a split that preserves behavior.

5) Error handling: consistent types and helpful context

Before (inconsistent errors):

```
if (!id) throw new Error('missing id');
if (id < 0) throw 'bad id';
```

After (consistent, structured errors):

```
if (!id) throw new ValidationError('missing id');
if (id < 0) throw new ValidationError('bad id', { id });
```

Ask reviewers to check whether errors are created in one place, whether messages are consistent, and whether context is included where it helps debugging.

A simple scoring rubric you can include in the request

Ask for a quick score with short justification. This makes feedback comparable across reviewers.

- **Naming clarity (0–2):** 0 unclear, 1 mostly clear, 2 consistently descriptive.
- **Control flow (0–2):** 0 tangled, 1 readable, 2 straightforward.
- **Duplication (0–2):** 0 repeated logic, 1 minor repetition, 2 minimal duplication.
- **Boundaries (0–2):** 0 mixed responsibilities, 1 mostly cohesive, 2 clean separation.
- **Error handling (0–2):** 0 inconsistent, 1 mostly consistent, 2 uniform and contextual.
- **Test readability (0–2):** 0 hard to read, 1 readable, 2 clear and extendable.

What to do with the feedback (so it turns into better code)

When you receive comments, respond with:

1. **Agreement or disagreement** with a brief reason.
2. **A plan**: “I’ll change X” or “I’ll keep X because Y.”
3. **A verification step**: “I’ll run the existing tests” or “I’ll add one test that covers the refactor.”

This keeps the review loop grounded in outcomes rather than taste.

Mini template: your “readability review” comment format

```
Top issues:
1) <issue> — why it matters for future changes
2) <issue> — where it shows up in the code
3) <issue> — what a safer alternative could look like

Suggested rewrite:
- <paste small revised section>

Questions:
- <one question that clarifies intent or constraints>
```

Using this format makes it easier to apply feedback quickly, especially when multiple reviewers comment on the same change.

4.2 Refactoring for Clarity Without Changing Behavior

Refactoring for clarity means improving how code reads and how easy it is to reason about, while keeping behavior identical. The trick is to separate "what the code does" from "how it looks," then change only the second part. If you can't confidently state the current behavior, you're not refactoring—you're rewriting.

The safety checklist (before you touch anything)

1. **Lock in behavior with tests**: Run the existing test suite first. If coverage is thin, add a few focused tests around the exact behavior you'll touch.
2. **Record the current contract**: Note inputs, outputs, side effects, and error cases. Even a short bullet list helps prevent accidental "helpful" changes.
3. **Use small steps**: Each step should be easy to review and easy to revert. If you change five things at once, you won't know which one mattered.
4. **Prefer mechanical transformations**: Rename variables, extract functions, reorder code for readability—these are usually behavior-preserving.
5. **Verify after each step**: Run tests (or targeted tests) after each meaningful change.

Mind map: clarity refactoring targets

Click here to view the mind map: Refactoring for clarity (no behavior change)

A practical workflow that stays behavior-safe

Step 1: Identify the "reading path"

Ask: if a teammate reads this function top-to-bottom, what do they learn first? A clarity refactor often starts by making the reading path match the mental model.

Example (before):

```
def process(order):
    total = 0
    for item in order['items']:
        if item['type'] == 'physical':
            total += item['price'] * item['qty']
        else:
            total += item['price']
    if order['coupon']:
        total -= total * order['coupon']['discount']
    if total < 0:
        total = 0
    return total
```

Issues are subtle: the "physical vs other" rule is embedded in the loop, coupon logic is mixed with totals, and the clamp-to-zero rule is tacked on at the end. None of that is wrong, but it's harder to verify.

Step 2: Extract intent into small functions

Extraction is usually behavior-preserving if you keep the same inputs and outputs and don't reorder side effects.

Example (after):

```
def item_total(item):
    if item['type'] == 'physical':
        return item['price'] * item['qty']
    return item['price']


def apply_coupon(total, coupon):
    if not coupon:
        return total
    return total - total * coupon['discount']


def clamp_non_negative(value):
    return max(0, value)


def process(order):
    total = sum(item_total(item) for item in order['items'])
    total = apply_coupon(total, order.get('coupon'))
    return clamp_non_negative(total)
```

Why this is safe: each extracted function is a direct translation of a single rule. The order of side effects is unchanged because the original code had no side effects inside the loop besides arithmetic, and the coupon and clamp happen after the loop in both versions.

Naming: clarity without changing meaning

Renaming is the lowest-risk refactor, but it can still break behavior if you accidentally change what a name refers to (for example, mixing up “discount” vs “discount_rate”). Use renames to make the “contract” visible.

Before:

```
def calc(a, b):
    c = a - b
    if c < 0:
        c = 0
    return c
```

After:

```
def non_negative_difference(minuend, subtrahend):
    difference = minuend - subtrahend
    if difference < 0:
        difference = 0
    return difference
```

The behavior is identical, but the name now tells you what the function guarantees: the result is never negative.

Reducing nesting: make the “happy path” obvious

Deep nesting often hides the real logic. Early returns can improve readability, but you must preserve side effects and exception behavior.

Before:

```
def authorize(user, action):
    if user is None:
        return False
    if user.role == 'admin':
        if action == 'delete':
            return True
        return False
    if action == 'read':
        return True
    return False
```

After:

```
def authorize(user, action):
    if user is None:
        return False
    if user.role == 'admin':
        return action == 'delete'
    return action == 'read'
```

This works because the original logic is a set of mutually exclusive conditions. The refactor collapses them into direct boolean expressions without changing which branch wins.

Control flow and side effects: the part people accidentally break

Clarity refactors often move code around. If code has side effects (logging, metrics, database calls, mutation), you must preserve their order.

Before:

```
def update_profile(user, payload):
    user.name = payload.get('name', user.name)
    if payload.get('email'):
        user.email = payload['email']
    log_change(user)
```

A “cleaner” version that breaks behavior might reorder `log_change` before updates. A safe clarity refactor keeps side effects at the same logical point.

After (safe):

```
def update_profile(user, payload):
    new_name = payload.get('name', user.name)
    new_email = payload.get('email', user.email)

    user.name = new_name
    if payload.get('email'):
        user.email = new_email

    log_change(user)
```

The key nuance: the conditional assignment to `user.email` is preserved. If you always assign `user.email = new_email`, you might overwrite with `None` when `email` is missing.

Duplication: extract shared logic, keep interfaces stable

When two blocks compute the same thing, extraction improves clarity and reduces the chance of inconsistent fixes later.

Before:

```
def shipping_cost(cart):
    subtotal = 0
    for item in cart['items']:
        subtotal += item['price'] * item['qty']
    if subtotal > 100:
        return 0
    return 10


def tax_amount(cart):
    subtotal = 0
    for item in cart['items']:
        subtotal += item['price'] * item['qty']
    return subtotal * 0.08
```

After:

```
def cart_subtotal(cart):
    return sum(item['price'] * item['qty'] for item in cart['items'])


def shipping_cost(cart):
    subtotal = cart_subtotal(cart)
    return 0 if subtotal > 100 else 10


def tax_amount(cart):
    return cart_subtotal(cart) * 0.08
```

Behavior is preserved because both original functions used the same subtotal computation and then applied their own rules.

How to use an AI assistant for this section (without losing control)

When asking for refactors, specify constraints: “Do not change outputs, error behavior, or side effects. Keep function signatures. Extract helpers only if they are pure.” Then compare diffs and run tests.

A good prompt pattern:

- Provide the function.
- State the contract in one or two sentences.
- Ask for a refactor plan first, then code.
- Require a checklist of what stayed the same.

Example prompt (you can adapt):

```
Refactor the function for clarity without changing behavior.
Constraints:
- Keep the same signature and return values.
- Preserve side effects and their order.
- Extract helpers only if they are pure.
- Provide a short list of behavior-preserving steps.
Then show the updated code.
```

Mind map: behavior-preserving guardrails

Click here to view the mind map: Behavior preservation

A quick self-review before you merge

- Can you point to the exact rule each extracted helper implements?
- Did you accidentally change default handling (missing keys, `None`, empty lists)?

- Are side effects still in the same place relative to computations?
- Does the refactor reduce cognitive load for the next reader, not just for you?

Clarity refactoring is successful when the code becomes easier to verify. If you can explain the function in fewer sentences after the change—and those sentences match the code—you've improved clarity without changing behavior.

4.3 Improving Naming, Structure, and Function Boundaries

Good naming and clean boundaries reduce the amount of guessing your future self has to do. When you ask an AI coding assistant to help, you'll get better results if you steer it toward intent, invariants, and "what changes together."

Naming that communicates intent

Use names that answer three questions:

1. *What is it?* (entity type)
2. *What does it do or represent?* (role)
3. *What are the constraints?* (units, ownership, lifecycle)

Examples (before → after):

- `data` → `userProfiles` (what data?)
- `temp` → `cachedToken` (what kind of temporary?)
- `handle()` → `handleCheckoutRequest()` (what request?)
- `count` → `activeUserCount` (which count?)

A practical trick: if you can't write a one-sentence comment explaining the name, the name is probably too vague.

Units and formats belong in the name.

- `timeoutMs` instead of `timeout`
- `createdAtIso` instead of `createdAt`

Avoid misleading specificity.

- `parseUserId()` is fine if it truly parses; if it also validates, consider `parseAndValidateUserId()`.

AI prompting that works: Ask the assistant to propose names *and* justify them with the three questions above. Then require it to keep names consistent across the file.

> Prompt example: "Rename variables and functions to reflect intent. For each rename, state what the name communicates (type, role, constraints). Keep naming consistent with existing conventions."

Structure: group by responsibility, not by convenience

Structure is how you decide what lives together. A common failure mode is "everything related to the feature is in one big file," which makes changes riskier.

Use these grouping rules:

- **Group by responsibility:** request handling, domain logic, persistence, and formatting should not be mixed.
- **Keep data transformations near their source:** if a function converts units, it should be close to the conversion.
- **Prefer small modules with clear entry points:** a reader should know where to start.

Example: separating layers in a small service (TypeScript-like pseudocode):

```
// Before: controller + business logic + persistence mixed
export async function checkout(req: Request) {
  const user = await db.users.find(req.userId);
  const cart = await db.carts.find(req.cartId);
  const total = cart.items.reduce((s, i) => s + i.price, 0);
  if (total <= 0) throw new Error('Invalid total');
  await db.orders.create({ userId: user.id, total });
  return { ok: true };
}
```

```
// After: boundaries make each part easier to test
export async function checkoutHandler(req: Request) {
  const user = await userRepository.getById(req.userId);
  const cart = await cartRepository.getById(req.cartId);
  const result = checkoutService.checkout(user, cart);
  await orderRepository.create(result.order);
  return { ok: true };
}

export const checkoutService = {
  checkout(user: User, cart: Cart) {
    const total = cart.items.reduce((s, i) => s + i.price, 0);
    if (total <= 0) throw new Error('Invalid total');
    return { order: { userId: user.id, total } };
  }
};
```

The “after” version makes it obvious which part can be tested without a database: `checkoutService.checkout`.

Function boundaries: define inputs, outputs, and invariants

A function boundary is a contract. If you make the contract explicit, the assistant can refactor safely.

A good function boundary has:

- **Single responsibility:** one main job.
- **Clear inputs:** types and meaning are unambiguous.
- **Clear outputs:** return values describe outcomes, not internal steps.
- **Defined invariants:** what must be true before and after.

Example: boundary cleanup in Python-like code:

```
# Before: does too much and hides assumptions
def process_order(order, db):
    if not order['items']:
        return None
    total = 0
    for item in order['items']:
        total += item['price']
    if total == 0:
        return None
    db.insert('orders', {'total': total, 'items': order['items']})
    return {'total': total}
```

```
# After: compute, validate, and persist are separate
def compute_order_total(items):
    return sum(item['price'] for item in items)

def validate_order(items, total):
    if not items:
        raise ValueError('No items')
    if total <= 0:
        raise ValueError('Total must be positive')

def persist_order(db, items, total):
    db.insert('orders', {'total': total, 'items': items})

def process_order(order, db):
    items = order['items']
    total = compute_order_total(items)
    validate_order(items, total)
    persist_order(db, items, total)
    return {'total': total}
```

Now each function has a smaller surface area, and the invariants are visible: `validate_order` enforces “items exist” and “total is positive.”

Mind maps: naming, structure, boundaries

Mind Map: Naming

Click here to view the mind map: Naming

Mind Map: Structure

Click here to view the mind map: Structure

Mind Map: Function Boundaries

Click here to view the mind map: Function Boundaries

Using an AI assistant to refactor safely

When you refactor with an assistant, you want it to preserve behavior while improving structure. The best workflow is to constrain the task and ask for a checklist.

Refactor checklist you can request:

- Identify the current function's responsibilities.
- Propose a new name for each extracted function.
- State invariants for each extracted function.
- Ensure call sites still pass the same data.
- Keep side effects in one place.

> Prompt example: "Refactor this function to improve naming and boundaries. Extract pure computation and validation into separate functions. Keep behavior identical. For each new function, list its inputs, outputs, and invariants. Then show the updated call site."

Common mistake to watch for: the assistant may extract helpers but keep them too "smart," such as a function that both validates and persists. If you see mixed responsibilities, ask it to split again.

Quick self-review questions

Before accepting a refactor, check:

- Can I explain each function in one sentence?
- Do names include the important constraints (units, format, ownership)?
- Are side effects isolated to a small number of functions?
- If I remove persistence, does the core logic still make sense?

These questions keep the refactor grounded in readability and correctness, not just rearranging code for its own sake.

4.4 Removing Duplication With Safe Abstractions

Duplication is a code smell with a practical side effect: every repeated change becomes a chance to miss one place. The goal of this section is to remove duplication without accidentally changing behavior. "Safe abstraction" means you extract structure while keeping inputs, outputs, and edge cases intact.

Recognize duplication types (and don't abstract the wrong thing)

Not all repetition deserves extraction. A quick way to sort it:

- **Exact duplication**: same logic copied with minor edits. This is usually a good candidate.
- **Near duplication**: same algorithm, different variable names or small conditional differences. Often extractable, but you must parameterize the differences.
- **Intentional duplication**: repeated code that is clearer when kept local (e.g., two small functions with different domain meaning). Keep it if abstraction would make reading harder.

A safe rule: if the duplicated blocks are both small and the differences are meaningful, prefer clarity over extraction. If the blocks are medium-sized and the differences are mechanical (e.g., different field names), abstraction is usually worth it.

Mind map: where duplication hides and how to remove it safely

Mind Map: Removing Duplication With Safe Abstractions

Click here to view the mind map: Removing Duplication With Safe Abstractions

Step-by-step: extract without breaking behavior

1. **Lock in behavior with tests** If there are no tests, add at least one that covers the duplicated path, including one edge case. You're not trying to test everything; you're trying to prevent silent behavior drift.
2. **Find the common "shape"** Write down what stays the same: the order of operations, the conditions, and the output format. Then list what varies: fields, thresholds, or which callback to call.
3. **Extract the smallest unit that removes the most duplication** Don't jump straight to a grand abstraction. Start with a helper that captures the shared core and accepts parameters for the differences.
4. **Preserve side effects exactly** If one copy logs and the other doesn't, don't "fix" that during extraction unless you're sure it's a bug. Side effects are where refactors go to hide.
5. **Use names that reflect intent** A safe abstraction is also a readable one. If the helper's name forces readers to mentally translate it back to the domain, you extracted too much or chose the wrong boundary.

Example 1: Near-duplicate request validation

Suppose you have two endpoints that validate the same fields but return different error messages.

```
def create_user(payload):
    if 'email' not in payload:
        return {'error': 'email required'}, 400
    if '@' not in payload['email']:
        return {'error': 'invalid email'}, 400
    if 'role' not in payload:
        return {'error': 'role required'}, 400
    return {'ok': True}, 201

def update_user(payload):
    if 'email' not in payload:
        return {'error': 'email missing'}, 400
    if '@' not in payload['email']:
        return {'error': 'email format'}, 400
    if 'role' not in payload:
        return {'error': 'role missing'}, 400
    return {'ok': True}, 200
```

The duplication is the validation flow; the variation is the exact error text and the success status. A safe abstraction extracts the shared checks and parameterizes the messages.

```
def validate_email_and_role(payload, msg):
    if 'email' not in payload:
        return {'error': msg['email_missing']}, 400
    if '@' not in payload['email']:
        return {'error': msg['email_invalid']}, 400
    if 'role' not in payload:
        return {'error': msg['role_missing']}, 400
    return None

def create_user(payload):
    err = validate_email_and_role(payload, {
        'email_missing': 'email required',
        'email_invalid': 'invalid email',
        'role_missing': 'role required'
    })
    if err: return err
    return {'ok': True}, 201

def update_user(payload):
    err = validate_email_and_role(payload, {
        'email_missing': 'email missing',
        'email_invalid': 'email format',
        'role_missing': 'role missing'
    })
    if err: return err
    return {'ok': True}, 200
```

Notice what didn't change: the checks are identical and run in the same order. The abstraction removes repetition while keeping the response shape and status codes consistent.

Example 2: Duplicate error handling around a database call

Two functions might catch the same exception and map it to the same response.

```
def get_order(order_id):
    try:
        return db.orders.get(order_id)
    except KeyError:
        return {'error': 'not found'}, 404

def delete_order(order_id):
    try:
        db.orders.delete(order_id)
        return {'ok': True}, 200
    except KeyError:
        return {'error': 'not found'}, 404
```

A safe abstraction here is a wrapper that centralizes the mapping, while leaving the success behavior to the caller.

```
def not_found_wrapper(fn):
    try:
        return fn()
    except KeyError:
        return {'error': 'not found'}, 404

def get_order(order_id):
    return not_found_wrapper(lambda: db.orders.get(order_id))

def delete_order(order_id):
    def do_delete():
        db.orders.delete(order_id)
        return {'ok': True}, 200
    return not_found_wrapper(do_delete)
```

This is safe because the only behavior being centralized is the exception-to-response mapping. The success paths remain separate and explicit.

Parameterize differences, don't branch inside the abstraction

A common mistake is to create a helper with a boolean flag like `is_create` that changes behavior in multiple places. That tends to reintroduce duplication as conditional complexity.

Instead, pass in the variation as a function or value. In the validation example, error messages were data, not a branching flag. In the database example, the success behavior was a callback.

Quick checklist for "safe" extraction

- **Same order of operations** in the extracted core.
- **Same error types and messages** (or intentionally changed with tests).
- **Same side effects** (logging, metrics, IO) or explicitly moved with care.
- **No hidden coupling**: the helper should not rely on external mutable state unless the original code already did.
- **Small boundary**: the helper should be easy to read in one screen.

A final practical test: can you explain the refactor in one sentence?

If you can't, the abstraction likely mixed concerns. A good one-sentence explanation looks like: "Both endpoints validate the same fields; we moved the shared checks into a helper and parameterized the error text." That sentence is boring, which is exactly the point.

4.5 Applying Style Guides and Linting Rules Consistently

Style guides and linting rules are how teams turn "I think this looks better" into "this is the same everywhere." The trick is to make them practical: enforce the rules that prevent real bugs, and keep the rest readable enough that people don't fight the tooling.

Why consistency matters (beyond aesthetics)

When code is consistent, reviews get faster because the reader spends less time decoding formatting choices. Consistency also reduces accidental behavior changes during edits. For example, a linter that enforces explicit boolean comparisons can prevent subtle truthiness bugs, while a formatter that standardizes indentation makes diffs smaller and easier to scan.

A simple workflow that actually sticks

Use a three-step loop:

1. **Format automatically** (so humans don't argue about whitespace).
2. **Lint with clear severity levels** (so warnings don't become background noise).
3. **Review exceptions explicitly** (so "we'll fix it later" doesn't become a policy).

Mind map: Style + lint consistency

Click here to view the mind map: Applying Style Guides and Linting Rules Consistently

Choose one source of truth

Pick a single style guide configuration per language and keep it in the repo. If different developers use different local configs, you'll see "works on my machine" formatting and lint results.

A good baseline includes:

- **Formatter rules** for whitespace, wrapping, and ordering where possible.
- **Lint rules** for correctness and maintainability.
- **Type checking** (if your stack supports it) to catch whole classes of issues that formatting can't.

Make local and CI behavior match

Consistency fails when CI rejects code that local tools accept. Ensure developers run the same commands locally as CI uses.

A practical approach:

- Add a single "check" command that runs formatter verification + lint.
- Add a "fix" command that runs formatter + auto-fixes for safe lint rules.

Example: a minimal “check” and “fix” setup

```
# check
npm run format:check
npm run lint

# fix
npm run format
npm run lint:fix
```

If your project uses a different ecosystem, the idea stays the same: one command for verification, one for safe correction.

Use severity levels so the signal stays readable

Not every rule should block merges. A common failure mode is treating every lint warning as urgent, which trains people to ignore the output.

A reasonable policy:

- **Errors**: correctness or high-risk maintainability issues (e.g., unused variables that hide mistakes, forbidden patterns).
- **Warnings**: style or complexity concerns that are worth addressing but not always urgent.
- **Info**: suggestions that help but shouldn’t interrupt flow.

Example: turning a “warning flood” into actionable output

Suppose your linter flags 200 line-length issues. If you set line-length as an error immediately, you’ll get merge friction without improving correctness. Instead:

- Start with line-length as a warning.
- Fix it gradually in touched files.
- Promote it to an error only after the baseline is clean.

Prefer rules that prevent real bugs

Some style rules are mostly about taste. Others prevent mistakes.

High-value lint rules often include:

- **No unused variables** (unused imports and variables can mask wrong logic).
- **No implicit fallthrough** (in switch-like constructs).
- **No shadowing of important names** (reduces confusion and accidental misuse).
- **Require explicit equality checks** (prevents truthiness surprises).
- **Disallow unsafe operations** (e.g., unsafe string formatting patterns).

Example: unused variables that hide a logic error

```
function total(items) {
  let sum = 0;
  for (const item of items) {
    const price = item.price;
    // Oops: price is never used; sum never changes.
  }
  return sum;
}
```

A linter rule like “no-unused-vars” would flag `price`. That’s not just cleanliness—it points to a missing update.

Keep naming and structure rules simple

Naming conventions should be enforceable without being pedantic. The goal is predictability.

Good targets:

- Consistent casing for variables and functions.

- Clear prefixes/suffixes for types or interfaces (if your language uses them).
- Avoiding ambiguous abbreviations in public APIs.

Example: consistent naming reduces review friction

```
def get_user_id(user):
    return user['id']

def get_user_email(user):
    return user['email']
```

When names follow a pattern, reviewers can scan quickly and focus on logic rather than guessing what each function returns.

Import ordering and grouping

Import rules prevent “random” diffs and help readers locate dependencies quickly.

A common, practical grouping:

- Standard library imports
- Third-party imports
- Local project imports

Example: import grouping that stays stable

```
import fs from 'node:fs';
import path from 'node:path';

import express from 'express';

import { loadConfig } from './config';
import { logger } from './logger';
```

If your formatter doesn’t handle import ordering, use a linter rule that does, so developers don’t reorder imports manually.

Line length and wrapping: choose a policy

Line length rules are useful when they support readability, not when they create constant churn.

Two workable policies:

- Enforce a moderate limit (e.g., 100–120) as a warning.
- Allow exceptions for URLs, generated code, or long literals.

Example: wrapping without changing meaning

```
result = some_function(
    first_argument,
    second_argument,
    third_argument,
)
```

This keeps diffs small and avoids awkward horizontal scrolling.

Complexity and readability rules: use them carefully

Rules like “max cyclomatic complexity” can be helpful, but they can also punish legitimate logic. A consistent approach:

- Start with warnings.
- Pair the rule with a refactoring guideline (split into helpers, reduce branching, extract predicates).

Example: reducing complexity with a helper

```
func isEligible(user User) bool {
    if !user.Active {
        return false
    }
    if user.Role == "admin" {
        return true
    }
    return user.Age >= 18 && user.Verified
}
```

If complexity becomes an issue, extract the branching into named helpers so the rule and the code both communicate intent.

Handle exceptions explicitly

When you must disable a rule, do it narrowly and with a reason. Broad disables turn into a hiding place.

A good exception includes:

- The exact rule name.
- A short justification tied to the code.
- The smallest scope possible (one line or one block).

Example: narrow disable with justification

```
// eslint-disable-next-line no-constant-condition
while (true) {
  // Intentionally infinite loop for a streaming consumer.
  break;
}
```

Even if the loop is later changed, the comment explains why the rule would otherwise be correct to enforce.

Make the rules teach through examples

When rules are consistent, developers learn them by seeing them applied. Include a short “style and lint” section in your repo’s contribution guide that explains:

- How to run format and lint locally.
- Which rules are errors vs warnings.
- How to request exceptions.

Mind map: exception handling

Click here to view the mind map: Exceptions

Quick checklist for consistency

- Formatter runs automatically and is verified in CI.
- Linter runs in CI and locally with the same config.
- Errors block merges; warnings are visible but not overwhelming.
- Rules focus on correctness first, readability second.
- Exceptions are rare, narrow, and explained.

When these pieces line up, style and lint stop being a chore and start being a shared language. The code becomes easier to read, and the team spends more time on behavior than on formatting.

5. Writing Tests and Using AI for Test Coverage

5.1 Choosing Test Types: Unit, Integration, and End to End

When you're using AI coding assistants, tests are your "ground truth." The trick is choosing the right test type so you catch the right kind of mistake without paying for slow, flaky runs. A good test suite is a set of different lenses: each lens has a job, and none of them should be forced to do the work of the others.

Unit tests: fast checks for small behavior

Unit tests validate a single function, method, or small class in isolation. They run quickly, so you can execute them frequently during development.

What unit tests are best at

- Validating pure logic: parsing, formatting, calculations, branching rules.
- Enforcing edge cases: empty inputs, boundary values, invalid states.
- Verifying interactions with collaborators via mocks (only when needed).

What unit tests are not best at

- Proving that your database query is correct.
- Proving that your HTTP endpoint works end-to-end.
- Catching integration issues like authentication wiring, schema mismatches, or network timeouts.

Example: unit test for business logic Suppose you have a function that calculates a discounted price.

- Input: `price=100`, `coupon=10%`, `minPrice=50`
- Expected: `90`, but never below `minPrice`

A unit test should focus on the calculation, not on HTTP, not on a database, and not on coupon storage.

```
def discounted_price(price, percent_off, min_price):
    discounted = price * (1 - percent_off / 100)
    return max(discounted, min_price)

def test_discounted_price_respects_min_price():
    assert discounted_price(100, 10, 95) == 95
```

How AI can help here (without making it messy) Ask the assistant to generate tests for edge cases you specify: "Add tests for percent_off=0, percent_off=100, negative percent_off should raise, and min_price greater than price." Then review the generated assertions for correctness and clarity.

Integration tests: verifying components work together

Integration tests check that multiple parts of the system cooperate correctly. They usually involve real infrastructure components (or realistic substitutes) such as a database, message broker, or filesystem.

What integration tests are best at

- Confirming that your persistence layer matches your expectations.
- Ensuring that serialization/deserialization works with real schemas.
- Validating that your service calls the right repository methods and handles returned data correctly.

What integration tests are not best at

- Validating UI behavior.
- Validating full request routing, middleware chains, and external third-party services all at once.

Example: integration test for repository + service Imagine a service method `createOrder` that writes to a database and returns the created order with an assigned ID.

An integration test should:

- Use a test database (often a disposable one).
- Run the service method.
- Query the database to verify the record.
- Optionally verify constraints like unique fields or required columns.

```
def test_create_order_persists_and_returns_id(db, order_service):
    order = order_service.create_order(customer_id=1, total=42.50)
    assert order.id is not None

    row = db.query_one("SELECT total FROM orders WHERE id = %s", (order.id,))
    assert row["total"] == 42.50
```

Common integration test pitfalls

- Over-mocking: if you mock the repository, you've turned it back into a unit test.
- Testing too much at once: if you also call external payment providers, you'll get brittle tests.
- Missing cleanup: leftover data can make tests pass or fail depending on order.

End-to-end (E2E) tests: validating user-visible flows

E2E tests run the application in a near-real setup and validate a complete workflow from the outside. They often involve the HTTP layer, authentication, and multiple internal services.

What E2E tests are best at

- Proving that routing, middleware, and request/response contracts work.
- Validating that the system behaves correctly as a whole.
- Catching "it works on my machine" wiring issues.

What E2E tests are not best at

- Covering every edge case in business logic.
- Acting as your only safety net for calculations and validations.

Example: E2E test for an HTTP endpoint If you have an endpoint `POST /orders`, an E2E test should:

- Send an HTTP request.
- Authenticate as a test user.
- Assert on the HTTP status and response body.
- Optionally verify database state (but keep it focused on the flow).

```
def test_post_orders_returns_201_and_order_id(http_client):
    resp = http_client.post(
        "/orders",
        json={"customer_id": 1, "total": 42.50},
        headers={"Authorization": "Bearer test-token"}
    )
    assert resp.status_code == 201
    assert "id" in resp.json()
```

Why E2E tests should be fewer They're slower and more sensitive to environment differences. A good approach is to cover the most important flows and keep the rest to unit and integration tests.

Mind map: choosing the right test type

Test Types Mind Map

Click here to view the mind map: Test Types

A practical decision checklist

Use this sequence when deciding what to test.

1. **What is the smallest thing that could be wrong?**
    - If the bug would live inside a calculation or validation rule, unit tests are the right first stop.
2. **Does the bug require a boundary to reproduce?**
    - If it depends on SQL, ORM mapping, or serialization, choose an integration test.
3. **Would a user notice the problem as a complete workflow failure?**
    - If yes, add an E2E test for that workflow.
4. **How expensive is the test to run and debug?**
    - If it's slow, keep it narrow and rely on unit/integration tests for detailed coverage.
5. **Can you diagnose failures quickly?**
    - If an E2E failure doesn't tell you where the problem is, add a unit or integration test that isolates the suspected layer.

Example test pyramid for a small feature

Consider a feature: "Create an order."

- **Unit tests** cover:
    - total validation rules
    - discount application
    - error handling for invalid inputs
- **Integration tests** cover:
    - service writes correct rows
    - constraints behave as expected
    - returned ID and fields match the database
- **E2E tests** cover:
    - `POST /orders` returns the right status and response shape
    - authentication and routing are wired correctly

This structure keeps the suite fast enough to be useful while still catching the kinds of failures that only appear when components meet.

How to phrase prompts for test generation

When using an AI assistant, be explicit about the test type and scope.

- For **unit tests**: "Generate unit tests for `discounted_price` covering boundary values and invalid inputs. No database or HTTP."
- For **integration tests**: "Generate an integration test that uses the test database to verify `create_order` persists the order and returns the ID."
- For **E2E tests**: "Generate an E2E test that sends an authenticated HTTP request to `POST /orders` and asserts on status code and response body."

Clear scope prevents the assistant from producing a slow test that pretends to be thorough, but isn't actually targeted.

5.2 Prompting for High Value Unit Tests With Examples

High-value unit tests do two things well: they pin down behavior precisely, and they fail in a way that tells you what to fix. When you ask an AI coding assistant to generate tests, you'll get better results if you describe (1) the contract, (2) the inputs that matter, and (3) what "correct" means in observable terms.

What to ask for (a practical checklist)

Use this structure in your prompt:

1. **Target function and contract**: "Given inputs X, it returns Y" or "It throws error Z."

2. **Test boundaries**: include at least one “happy path,” one “edge case,” and one “invalid input” case.
3. **Determinism requirements**: if time, randomness, or I/O is involved, specify how to control it (e.g., inject a clock, mock a dependency).
4. **Assertions that matter**: specify what to assert (return value, thrown exception type/message, side effects, calls to collaborators).
5. **Test style constraints**: prefer table-driven tests, avoid snapshot tests unless the output is stable and small.

A good prompt doesn’t ask for “more tests.” It asks for **specific behaviors**.

Mind map: turning a function into test cases

Click here to view the mind map: Unit test prompt recipe

Example 1: Pure function with edge cases (JavaScript)

Function under test: `normalizeEmail(email)` returns a normalized email string.

Contract (you provide):

- Trim whitespace.
- Lowercase the domain part only.
- Reject missing `@` by throwing `TypeError`.

Prompt you can use:

> Write unit tests for `normalizeEmail(email)`. Contract: trim whitespace; lowercase only the domain; keep local part as-is. Throw `TypeError` if `email` has no `@`. Include: one happy path, one whitespace edge case, one missing-@ invalid case. Use Jest. Assert exact returned strings and exact error type.

Example tests (what you should expect to see):

```
// normalizeEmail.test.js
const { normalizeEmail } = require('./normalizeEmail');

describe('normalizeEmail', () => {
  test('trims whitespace and lowercases domain only', () => {
    const input = '  Alice@EXAMPLE.com  ';
    const out = normalizeEmail(input);
    expect(out).toBe('Alice@example.com');
  });

  test('keeps local part casing unchanged', () => {
    const input = 'Bob+Tag@Example.org';
    const out = normalizeEmail(input);
    expect(out).toBe('Bob+Tag@example.org');
  });

  test('throws TypeError when missing @', () => {
    expect(() => normalizeEmail('not-an-email'))
      .toThrow(TypeError);
  });
});
```

Notice what’s missing: no test that just repeats the implementation. Each test targets a rule from the contract.

Example 2: Function with collaborators (Python)

Function under test: `sendInvoiceEmail(invoice_id, email_client)`.

Contract:

- Fetch invoice data via `getInvoice(invoice_id)`.
- Call `email_client.send(to, subject, body)` exactly once.
- If invoice is not found, raise `ValueError` and do not call `send`.

Prompt you can use:

> Create unit tests for `sendInvoiceEmail(invoice_id, email_client)`. Behavior: it calls `getInvoice(invoice_id)`. If invoice exists, it sends one email using `email_client.send` with correct arguments. If invoice does not exist, raise ValueError and do not call `email_client.send`. Use pytest. Mock `getInvoice` and `email_client`. Include: success case and not-found case.

Example tests:

```
# test_send_invoice_email.py
import pytest
from unittest.mock import Mock

import mymodule
from mymodule import sendInvoiceEmail


def test_send_invoice_email_success(monkeypatch):
    fake_invoice = {
        'id': 'inv-1',
        'customerEmail': 'buyer@example.com',
        'total': 42.50,
    }
    monkeypatch.setattr(mymodule, 'getInvoice', lambda _: fake_invoice)

    email_client = Mock()
    sendInvoiceEmail('inv-1', email_client)

    email_client.send.assert_called_once()
    args, kwargs = email_client.send.call_args
    assert args[0] == 'buyer@example.com'
    assert 'Invoice' in args[1]
    assert '42.50' in args[2]


def test_send_invoice_email_not_found(monkeypatch):
    monkeypatch.setattr(mymodule, 'getInvoice', lambda _: None)

    email_client = Mock()
    with pytest.raises(ValueError):
        sendInvoiceEmail('missing', email_client)

    email_client.send.assert_not_called()
```

The assertions focus on observable behavior: call count and argument content. That's how tests stay useful when the internal formatting changes slightly.

Example 3: Table-driven tests for parameterized logic (Go)

Function under test: `isValidZip(zip string) bool`.

Contract:

- Accepts 5 digits.
- Accepts 5 digits + hyphen + 4 digits.
- Rejects anything else.

Prompt you can use:

> Write table-driven unit tests for `isValidZip(zip)` in Go. It returns true for: "12345" and "12345-6789". It returns false for: empty string, letters, wrong digit counts, and wrong separators. Use `testing` package and `t.Run`.

Example tests:

```go
// zip_test.go
package mypkg

import "testing"

func TestIsValidZip(t *testing.T) {
  cases := []struct {
    name string
    zip  string
    want bool
  }{
    {"five_digits", "12345", true},
    {"plus_four", "12345-6789", true},
    {"empty", "", false},
    {"letters", "12A45", false},
    {"too_short", "1234", false},
    {"wrong_sep", "12345_6789", false},
  }

  for _, tc := range cases {
    t.Run(tc.name, func(t *testing.T) {
      got := isValidZip(tc.zip)
      if got != tc.want {
        t.Fatalf("isValidZip(%q)=%v, want %v", tc.zip, got, tc.want)
      }
    })
  }
}
```

Table-driven tests are especially good when the contract is a set of accepted patterns.

Prompting tactics that improve test quality

1. **Ask for "one test per rule."** If the contract has five rules, request five tests (or five groups) rather than one big test.
2. **Require explicit invalid inputs.** Many assistants generate only happy paths unless you name invalid cases.
3. **Specify assertion granularity.** If you only care about the exception type, say so. If you care about the message, request it.
4. **Control time and randomness.** If a function uses `now()` or `uuid()`, ask the assistant to inject a clock/ID generator or mock them.
5. **Prevent "test mirroring."** Tell the assistant not to re-implement the production logic inside the test. For example: "Do not compute expected output by calling `normalizeEmail` again."

A compact prompt template you can reuse

```
Write unit tests for <function/module>.
Contract:
- <rule 1>
- <rule 2>
- <rule 3>
Include:
- 1 happy path
- 1 edge case
- 1 invalid input
Constraints:
- Use <framework>
- Mock <dependencies>
- Assertions: <what to assert>
Output: <test file only>
```

When you provide the contract and the boundaries, the assistant can generate tests that are both correct and maintainable. The goal isn't maximum coverage; it's coverage of the behaviors you actually rely on.

5.3 Generating Test Data and Fixtures Reliably

Reliable tests start with reliable inputs. When test data is inconsistent, failures become hard to reproduce and fixes turn into guesswork. This section focuses on generating test data and fixtures in a way that is deterministic, readable, and easy to adjust when requirements change.

Why “reliable” test data matters

A fixture is only as good as its ability to produce the same conditions every time. Reliability usually means three things:

- **Determinism:** the same test run produces the same data.
- **Isolation:** tests don’t accidentally depend on each other’s side effects.
- **Traceability:** when a test fails, you can tell which data caused it.

A practical rule: if you can’t explain how the test data was produced in one minute, the fixture is probably too clever.

Mind map: fixture reliability checklist

Click here to view the mind map: Generating test data reliably

Deterministic data: controlling randomness and time

If you use randomness to create variety, make it repeatable.

Example (Python-style pseudocode): deterministic generator

```
import random

def make_user(seed, idx):
    rng = random.Random(seed)
    return {
        "id": f"u-{idx}",
        "email": f"user{idx}+{rng.randint(1000,9999)}@example.com",
        "role": rng.choice(["admin", "member"]),
    }
```

In this pattern, the generator’s output depends only on `seed` and `idx`. If a test fails, you can regenerate the exact same user objects.

Time is another common source of non-determinism. If your code behaves differently based on timestamps, your tests should provide explicit timestamps.

Example: explicit timestamps instead of “now”

- Bad: `created_at = datetime.utcnow()` inside the fixture.
- Better: `created_at = datetime(2026, 1, 15, 12, 0, 0)` or a fixed offset from a known base.

Isolation: keep tests from stepping on each other

Isolation is often where “reliable” breaks down. Two tests can pass alone and fail together if they share state.

Common pitfalls

- Reusing the same database rows across tests.
- Using a global list that accumulates objects.
- Relying on auto-increment IDs without resetting the database.

Example: unique identifiers per test

```
import uuid

def user_payload():
    return {
        "id": str(uuid.uuid4()),
        "email": f"test-{uuid.uuid4()}@example.com",
    }
```

Even if the database isn’t reset between tests, unique IDs prevent collisions. Still, you should prefer a clean state strategy (transaction rollback, database reset, or per-test schema) because uniqueness doesn’t protect against logic that queries by non-unique fields.

Traceability: make it obvious what data a test uses

When a test fails, you want to answer: “Which fixture values led to this?” That means your fixture should be easy to inspect.

Good fixture traits

- Critical fields are set explicitly.
- Defaults are sensible and documented in code.
- Overrides are supported without rewriting the whole fixture.

Example: composable factory with overrides

```
def user_fixture(role="member", email=None):
    payload = {
        "id": "u-1",
        "role": role,
        "email": email or "member@example.com",
    }
    return payload

# In a test:
admin = user_fixture(role="admin", email="admin@example.com")
```

This is intentionally boring: explicit parameters make failures easier to interpret.

Building fixtures: defaults plus targeted overrides

A useful approach is to create small builders that cover common cases, then override only what a test needs.

Mind map: fixture design approach

Click here to view the mind map: Fixture design

Example: related objects without hidden coupling Suppose you need an order and its line items.

- Create an `order_fixture()` that sets order-level fields.
- Create a `line_item_fixture()` that sets item-level fields.
- In the test, assemble them explicitly.

This avoids a single factory that silently creates ten different things with unclear relationships.

Handling edge cases without random chaos

Edge cases are easier to test when you can describe them precisely.

Example: boundary values as explicit fixtures

- Empty list
- Single element
- Maximum length
- Invalid format

Instead of generating random strings until one fails validation, define the failing input directly.

Example: invalid email fixture

```
def invalid_email_fixture():
    return {
        "email": "not-an-email",
        "role": "member",
    }
```

This makes the test’s intent obvious and prevents “flaky by luck” outcomes.

Data volume: keep it small, but representative

Large datasets slow tests and make failures harder to interpret. A fixture should be just big enough to exercise the behavior.

Rule of thumb

- If the code path depends on ordering, include at least the minimum number of items that demonstrate ordering.
- If the code path depends on aggregation, include enough items to cover the aggregation logic.
- If the code path depends on filtering, include items that should match and items that should not.

Example: filtering test with minimal coverage

- Create 3 records: 1 matching, 1 non-matching by type, 1 non-matching by status.
- Assert that only the matching record appears.

You get coverage without turning the test into a mini production dataset.

Practical fixture patterns for common stacks

Even without tying to a specific framework, these patterns show up everywhere.

1. **Factory functions** for payloads (pure data).
2. **Repository/DAO helpers** for persistence (side effects).
3. **Test setup hooks** that create a clean baseline.

Keep payload generation separate from database writes. That way, you can reuse payload fixtures for both unit tests (no database) and integration tests (with database).

Using AI assistants to generate fixtures safely

When you ask an assistant to generate test data, require it to:

- Use explicit values for critical fields.
- Avoid `now` and uncontrolled randomness.
- Provide a clear override mechanism.
- Keep the fixture small and focused.

Example prompt you can adapt

- "Create a deterministic fixture for an order with 2 line items. Use fixed timestamps and stable IDs. Provide an override for quantity on the first line item."

The assistant's job is to produce structured, readable fixtures—not to invent new test scenarios on your behalf.

Quick checklist before you trust a fixture

- Can I reproduce the same data every run?
- Does the test clean up after itself or run in isolation?
- Are the key inputs explicit and easy to read?
- Is the dataset minimal while still covering the behavior?
- Can I override one field without breaking the rest?

When these answers are "yes," your tests stop being fragile and start being informative.

5.4 Debugging Failing Tests With AI Explanations

When a test fails, you want two things: a precise location of the problem and a short path to a fix. AI explanations can help you get there faster, but only if you treat them like hypotheses. The goal of this section is to turn a failing test into a small, test-driven investigation.

A practical workflow (use it every time)

1. **Read the failure like a log, not a story.** Capture the test name, assertion message, and the first stack frame that points to your code.
2. **Reproduce locally with the narrowest command.** Run only the failing test so you can iterate quickly.
3. **Classify the failure type.** Most failures fall into: wrong output, wrong side effect, wrong exception, timing/flakiness, or environment mismatch.

4. **Ask for an explanation tied to the exact failure.** Provide the test code, the relevant production code, and the full error output.
5. **Convert the explanation into a concrete next action.** For example: "Check the boundary condition at index 0," or "Verify the mocked dependency is actually called."
6. **Patch minimally and re-run the same test.** Avoid "fixing" multiple things at once.

Mind map: from failing test to fix

Click here to view the mind map: Failing test

Example 1: Wrong output (off-by-one)

Failing test (Python):

```
def test_slices_include_end():
    data = [10, 20, 30, 40]
    assert slice_items(data, 1, 3) == [20, 30]
```

Failure message (typical):

- `AssertionError: assert [20, 30, 40] == [20, 30]`

What to notice: the actual result includes one extra element. That usually points to an end index handling issue.

Production code (buggy):

```
def slice_items(items, start, end):
    return items[start:end+1]
```

AI prompt that works (include evidence):

- "Test `test_slices_include_end` expects `[20, 30]` but got `[20, 30, 40]`. Explain the most likely cause in `slice_items` and propose the smallest code change. Then list one extra test case to confirm the boundary behavior."

AI explanation you should look for:

- It should connect the extra element to `end+1`.
- It should mention that Python slicing already treats `end` as exclusive.
- It should propose removing `+1`.

Minimal patch:

```
def slice_items(items, start, end):
    return items[start:end]
```

Verification: re-run only `test_slices_include_end`, then add a boundary test like `slice_items(data, 0, 0) == []`.

Example 2: Wrong side effect (mock not called)

Failing test (JavaScript/Jest):

```
test('sends email on signup', async () => {
  const sendEmail = jest.fn();
  await signup({ email: 'a@b.com' }, { sendEmail });
  expect(sendEmail).toHaveBeenCalledWith('a@b.com');
});
```

Failure message:

- `Expected number of calls: >= 1, Received: 0`

What to notice: the function under test ran, but the dependency wasn't invoked with the expected argument.

Production code (buggy):

```
async function signup(user, deps) {
  if (!user.email) return;
  await deps.sendEmail(user.username);
}
```

AI prompt that works:

- "The test expects `sendEmail('a@b.com')` but the mock was never called. Explain why `deps.sendEmail` might not be called or might be called with a different argument. Use the provided `signup` code and propose a minimal fix."

AI explanation you should look for:

- It should point out `user.username` is undefined in this test input.
- It should note that the mock is called with `undefined` only if the code reaches `sendEmail`; if the mock shows zero calls, it may also be due to the early return `if (!user.email) return;` (but here `user.email` exists, so the more likely issue is argument mismatch).

Minimal patch:

```
async function signup(user, deps) {
  if (!user.email) return;
  await deps.sendEmail(user.email);
}
```

Verification: re-run the test; if it still fails, ask the AI to suggest a targeted assertion like `expect(sendEmail).toHaveBeenCalled()` to separate "not called" from "called with wrong args."

Example 3: Wrong exception (error type mismatch)

Failing test (Java):

```
@Test
void rejectsNegativeAmount() {
  assertThrows(IllegalArgumentException.class,
    () -> process(-5));
}
```

Failure message:

- `Unexpected exception type thrown; expected IllegalArgumentException but got RuntimeException`

Production code (buggy):

```
void process(int amount) {
  if (amount < 0) throw new RuntimeException("negative");
}
```

AI prompt that works:

- "The test expects `IllegalArgumentException` for negative amounts, but `RuntimeException` is thrown. Explain the mismatch and propose the smallest change. Also suggest one test for amount = 0."

AI explanation you should look for:

- It should map the thrown exception class to the test expectation.
- It should recommend changing the exception type, not rewriting the logic.

Minimal patch:

```
void process(int amount) {
  if (amount < 0) throw new IllegalArgumentException("negative");
}
```

How to ask for “useful explanations” (and avoid vague ones)

Use prompts that force the assistant to reference the specific failure.

- **Include the exact assertion and diff.** “Expected X, got Y.”
- **Provide the smallest relevant code slice.** Test + the function it calls.
- **Ask for verification steps, not just guesses.** “What line would you inspect first, and what would you expect to see?”
- **Request a minimal patch.** “Change as little as possible.”

Mind map: prompt structure for debugging

Click here to view the mind map: prompt structure for debugging

Common pitfalls when using AI explanations

- **Treating the explanation as truth.** Always confirm with a targeted rerun or a small additional assertion.
- **Fixing the test instead of the code.** Only adjust the test if the spec is wrong; otherwise you’re teaching the test to accept the bug.
- **Ignoring boundary conditions.** Many failures come from index handling, empty inputs, or null/undefined values.

A closing checklist for this subsection

- I can point to the first relevant line in my code from the stack trace.
- I know whether the failure is value, behavior, exception, or environment.
- My AI prompt included the exact expected/actual evidence.
- The proposed fix is minimal and I re-run only the failing test.
- I added (or planned) one boundary test to prevent the same class of bug from returning.

5.5 Measuring Coverage and Filling Gaps Pragmatically

Coverage is a blunt instrument: it tells you which lines (or branches) ran, not whether the code behaved correctly. Still, it’s useful when you treat it like a map of where your tests have already walked, and where they haven’t.

What to measure (and what to ignore)

Start by deciding which coverage signal matches your risk.

- **Line coverage** answers: “Did this line execute?” It’s helpful for catching totally untested paths, but it can look good even when assertions are weak.
- **Branch coverage** answers: “Did both sides of a decision run?” It’s better for logic-heavy code like conditionals and error handling.
- **Path coverage** is usually impractical. You can’t realistically test every combination of inputs, but you can test the important ones.

A practical rule: use coverage to find **missing behavior**, then use tests to prove **correct behavior**.

Mind map: coverage to action

Coverage → Action Mind Map

Click here to view the mind map: Coverage → Action

Step-by-step: a pragmatic workflow

1. **Generate a coverage report locally** and sort by the biggest gaps.
   - Don’t start with the smallest percentages; start with files that contain critical logic.
2. **Open the report and read the uncovered code**.

- Coverage tools show line numbers; your job is to interpret what behavior those lines represent.

3. **Classify the gap**.
   - Is it unreachable code? Is it missing input coverage? Is it an error path you never triggered?
4. **Write the smallest test that proves the intended behavior**.
   - If the code is supposed to reject invalid input, the test should fail on invalid input and pass on valid input.
5. **Re-run tests and confirm the gap closes**.
   - If coverage improves but the test doesn't add confidence, you likely wrote a "runs without failing" test.

Example: line coverage looks fine, but branch coverage is missing

Imagine a function that validates a coupon code.

```
def apply_coupon(code: str, total: float) -> float:
    if not code:
        raise ValueError("Coupon code required")
    if code == "SAVE10":
        return total * 0.9
    if code == "FREESHIP":
        return total
    raise ValueError("Unknown coupon")
```

You might have tests for `SAVE10` and an unknown code, but forget the empty string case. Line coverage could still be high if most lines execute in other tests.

A branch-focused test set:

```
import pytest

def test_apply_coupon_save10():
    assert apply_coupon("SAVE10", 100.0) == 90.0

def test_apply_coupon_freeship():
    assert apply_coupon("FREESHIP", 100.0) == 100.0

def test_apply_coupon_empty_raises():
    with pytest.raises(ValueError, match="required"):
        apply_coupon("", 100.0)

def test_apply_coupon_unknown_raises():
    with pytest.raises(ValueError, match="Unknown coupon"):
        apply_coupon("NOPE", 100.0)
```

Now you're not just executing lines; you're proving that each decision point behaves correctly.

Example: uncovered lines are sometimes unreachable

Suppose you see uncovered code in an exception handler:

```
def parse_int(s: str) -> int:
    try:
        return int(s)
    except ValueError:
        return 0
    except TypeError:
        # Should never happen if s is always a string
        return 0
```

If your type hints and callers guarantee `s` is a string, the `TypeError` branch may be unreachable. Filling that gap by forcing a `TypeError` might create a test that doesn't match real usage.

A better approach is to:

- confirm the call sites (or add a runtime check if the guarantee is weak), and

- decide whether the branch is truly dead code or just untested.

If it's dead, you can accept the uncovered line and focus on behavior that matters.

Example: "executed but unasserted" tests

Coverage can increase while confidence stays flat. Consider this test:

```
def test_apply_coupon_runs():
    apply_coupon("SAVE10", 100.0)
```

It executes the code but doesn't verify the result. Coverage tools will happily count it as a hit.

A better test asserts outcomes:

```
def test_apply_coupon_asserts_result():
    assert apply_coupon("SAVE10", 100.0) == 90.0
```

When you see a coverage gap that's stubborn, check whether existing tests are asserting anything meaningful.

Filling gaps by risk, not by percentage

A low-coverage module that only formats strings might be less urgent than a high-coverage module that handles money, permissions, or state transitions.

Use a simple triage matrix:

- **High risk + low coverage**: write tests first.
- **High risk + high coverage**: review assertions and branch coverage.
- **Low risk + low coverage**: consider whether tests are worth the effort.
- **Low risk + high coverage**: ensure tests aren't just "running."

Turning uncovered lines into test ideas

When you open uncovered code, convert it into a testable statement.

- If the code checks an input condition, write tests for **valid** and **invalid** inputs.
- If the code handles an error case, write tests that trigger that error deterministically.
- If the code transforms data, write tests that validate the transformation with representative inputs.

A quick checklist for each uncovered block:

- What inputs lead to this code?
- What output or side effect should happen?
- What should happen when inputs are wrong?
- Are there invariants (like "never returns negative totals") that should be enforced?

Practical guardrails to keep coverage work sane

- **Don't chase coverage in generated or vendor code**. Focus on your code paths.
- **Avoid testing implementation details**. Assert behavior, not internal calls.
- **Prefer small, deterministic tests**. If a test depends on timing or randomness, coverage improvements may be flaky.
- **Re-check after refactors**. A refactor can move code around and change coverage without changing behavior.

A final sanity check

After you add tests to close a coverage gap, ask one question: "If this test fails, would I know what behavior broke?" If the answer is no, improve the assertions or the test inputs. Coverage is the starting point; good tests are the destination.

6. Performance Optimization Fundamentals

6.1 Identifying Bottlenecks With Profiling and Metrics

When code feels slow, the hardest part is proving *where* the time goes. Profiling and metrics answer that question with evidence, not vibes. The goal is simple: find the smallest set of components that account for most of the latency or resource use, then measure again after changes.

Start with the question you're actually answering

Before running tools, decide what "bottleneck" means for your case.

- **Latency bottleneck:** Which request phase (CPU work, waiting on I/O, serialization) dominates end-to-end time?
- **Throughput bottleneck:** Which stage limits requests per second under load?
- **Resource bottleneck:** Which resource is saturated (CPU, memory, disk, network, database connections)?

A quick sanity check prevents wasted effort. If CPU is near 100% during slow periods, you likely have compute-heavy work. If CPU is low but requests wait, you likely have I/O waits, locks, or external dependencies.

Build a measurement baseline

A baseline is the "before" state you can compare against.

1. **Pick a metric:**
   - p50/p95/p99 latency for user requests
   - error rate (to avoid optimizing the wrong thing)
   - CPU time, heap usage, GC time, queue depth
2. **Pick a scope:**
   - local dev run for unit-level issues
   - staging environment for realistic behavior
   - production sampling for real traffic patterns
3. **Pick a time window:**
   - enough duration to smooth out noise (for example, 5–15 minutes)

If you only measure averages, you'll miss tail latency. p95 and p99 often reveal lock contention, slow queries, or occasional retries.

Use a mind map to choose the right tool

Bottleneck Identification Mind Map

Click here to view the mind map: Goal: Find the biggest contributor to time or resource use

Profiling: match the profiler to the symptom

Different profilers answer different questions.

CPU profiling (where the CPU time goes)

Use CPU profiling when you suspect compute-heavy work: parsing, hashing, serialization, tight loops, or expensive algorithms.

Example scenario: A web endpoint slows down after adding a new feature.

- Metrics show p95 latency increased.
- CPU usage on app servers rises during the slowdown.
- CPU profiler highlights a hot function: `renderTemplate()`.

Now you know where to look, not just that "the server is slow."

Heap and allocation profiling (why memory pressure rises)

Use heap/alloc profiling when you see increased GC time, memory growth, or frequent pauses.

Example scenario: Latency spikes correlate with GC.

- Metrics show heap usage climbing and GC time increasing.
- Allocation profiler shows large temporary buffers created per request.

The bottleneck isn't "GC is bad." It's "we allocate too much per request," which forces GC to work harder.

Tracing (what happens across services)

Use distributed tracing when latency includes waits across network calls.

Example scenario: The endpoint calls three services.

- End-to-end p95 is high.
- Traces show the app spends 70% of time waiting on `billing-service`.

Now you can separate "our code is slow" from "an upstream dependency is slow," and you can measure the upstream contribution precisely.

Thread and lock profiling (why work is blocked)

Use lock contention and thread state analysis when CPU is low but latency is high.

Example scenario: CPU is 20%, but p99 latency is terrible.

- Thread dump shows many threads waiting on a single mutex.
- Profiling shows a shared cache lock around a slow operation.

This is a classic "everything is blocked behind one door" situation.

Metrics: instrument the phases, not just the endpoint

Endpoint timing alone is often too coarse. Add phase-level measurements so you can attribute time.

A practical approach is to record durations for:

- request parsing / validation
- business logic CPU time
- database calls (including time waiting for results)
- external HTTP calls
- serialization / response writing

Example (pseudo-instrumentation):

- `phase_db_ms`
- `phase_http_ms`
- `phase_compute_ms`
- `phase_total_ms`

Then you can answer: "Which phase grew?"

A concrete workflow that doesn't waste time

1. **Observe:** Identify which metric is bad (p95 latency, GC time, queue depth).
2. **Correlate:** Check resource saturation (CPU, heap, disk, network).
3. **Localize:** Use profiling to find hot spots in the suspected category.
4. **Confirm:** Verify with phase metrics or traces.
5. **Fix and re-measure:** Apply one change, then compare the same metrics.

This loop prevents the common failure mode: optimizing a function that isn't actually responsible for the slowdown.

Example: from "slow" to a specific bottleneck

Suppose a checkout endpoint slows down.

- Baseline: p95 latency rises from 220ms to 480ms.
- Resource check: CPU stays flat; GC time increases slightly.
- Phase metrics: `phase_db_ms` jumps from 40ms to 220ms.

- Database profiling: query plan shows a missing index on `orders(user_id, status)`.
- After adding the index: `phase_db_ms` drops to 55ms and p95 latency returns to ~240ms.

Notice the logic: the fix matches the measured contributor. No guesswork required.

Example: from "high CPU" to a targeted change

A background job processes events and starts falling behind.

- Throughput metric: events processed per minute drops.
- CPU profiling: 60% of CPU time is in `json_encode()`.
- Allocation profiling: large temporary strings created during encoding.
- Change: switch to a streaming encoder or reuse buffers.
- Re-measure: CPU drops and throughput recovers.

Again, the profiler tells you what to optimize; the metrics confirm the impact.

Avoid the common traps

- **Optimizing the wrong layer:** If traces show most time is waiting on the database, rewriting a loop won't help.
- **Measuring in a different environment:** Local tests can hide contention and network waits.
- **Changing multiple variables at once:** If you refactor and add caching simultaneously, you won't know what caused the improvement.
- **Ignoring tail behavior:** p50 can look fine while p99 is broken.

What "good" looks like after identification

After you identify a bottleneck, you should be able to state it precisely, for example:

- "p95 latency increased because `phase_db_ms` increased by 180ms, driven by query X."
- "GC time increased because we allocate ~N MB per request in function Y."
- "Threads are blocked on lock Z, causing queue growth and p99 latency spikes."

That level of specificity is what makes the next step—debugging and optimization—efficient and reliable.

6.2 Prompting AI to Propose Measurable Improvements

Measurable improvements start with measurable baselines. When you ask an AI assistant to optimize code, you want it to propose changes that can be verified with numbers, not vibes. The trick is to give it (1) a target metric, (2) a current measurement, (3) constraints, and (4) a way to validate.

What to include in your prompt

Use this checklist as a mental model. Each item reduces guesswork.

- **Target metric:** Pick one primary metric and one secondary metric. Example: "p95 latency" (primary) and "error rate" (secondary).
- **Current baseline:** Provide the current values and the test method. Example: "p95=420ms from 10k requests, load test profile X."
- **Scope:** State what parts are in play. Example: "optimize only the request handler and its helper functions."
- **Constraints:** Include correctness, style, and resource limits. Example: "no behavior change, keep memory under 200MB, no new dependencies."
- **Validation plan:** Ask for a concrete test/benchmark command or procedure. Example: "propose how to rerun the same load test and how to compare results."
- **Evidence style:** Request reasoning tied to code structure. Example: "point to the exact loop, allocation, or query pattern causing the cost."

A practical prompt template

You can copy this structure and fill in the blanks.

> You are helping me optimize a code path. Primary metric: ____. Current baseline: ____ (how measured: ____). Secondary metric: ____.
> Scope: ____.
> Constraints: ____.
> Read the code below and propose 3–5 changes ranked by expected impact.
> For each change, include:
>
> 1. what to change (specific location),
> 2. why it should improve the metric (mechanism),
> 3. how to measure before/after (exact test/benchmark steps),
> 4. risks or tradeoffs (what might worsen and how we'll detect it). Keep behavior identical; if you need clarification, ask targeted questions.

Mind map: turning "optimize" into measurable proposals

Mind map: Measurable optimization prompts

[Click here to view the mind map: Goal: measurable improvements]

Example 1: Latency optimization in a request handler

Scenario: A web endpoint is slow. You have a baseline.

- Primary metric: p95 latency
- Baseline: p95=420ms, p50=95ms
- Secondary metric: error rate < 0.1%
- Scope: function `handle_request()` and helpers it calls
- Constraints: no behavior change, keep response schema identical

Prompt you send (shortened for readability):

> Optimize `handle_request()` for p95 latency. Baseline: p95=420ms from 10k requests using load profile "A", p50=95ms. Error rate currently 0.03%. Scope: only code under `handle_request()` and its direct helpers. Constraints: no behavior change, no new dependencies, memory must not increase by more than 10MB. Propose 3–5 ranked changes. For each: specify the exact code location, explain the mechanism (e.g., repeated parsing, extra allocations, blocking I/O), and give a before/after measurement plan using the same load profile.

What a good assistant response should contain (example content):

1. **Cache repeated parsing within the request**
    - **Change:** Move JSON schema parsing out of a per-item loop; parse once per request.
    - **Mechanism:** Re-parsing schema for each item multiplies CPU cost; parsing is deterministic per request.
    - **Measure:** Run the same load profile "A" and compare p95 and CPU time per request. Also add a micro-benchmark for the parsing function.
    - **Risk/tradeoff:** If schema depends on request fields, caching must be keyed; detect by adding a test that varies those fields.
2. **Reduce blocking calls by batching**
    - **Change:** Replace N sequential calls with a single batched call or parallel calls with a bounded concurrency limit.
    - **Mechanism:** Sequential I/O adds waiting time; batching reduces round trips.
    - **Measure:** Compare p95 latency and the number of outbound calls per request (log counters). Confirm error rate stays below threshold.
    - **Risk/tradeoff:** Batching may increase payload size; detect by monitoring response size and timeouts.
3. **Avoid unnecessary allocations in hot loops**
    - **Change:** Preallocate lists/arrays when size is known; reuse buffers where safe.
    - **Mechanism:** Allocation pressure increases GC/allocator overhead, especially on the p95 tail.
    - **Measure:** Use a profiler to compare allocation counts and GC time before/after; rerun load test.
    - **Risk/tradeoff:** Reuse must not leak state across requests; detect with concurrency tests.

Notice what's missing: no "should be faster" statements without a measurement plan.

Example 2: CPU optimization with a micro-benchmark

Scenario: A function `score(items)` burns CPU. You can benchmark it directly.

- Primary metric: runtime of `score(items)` for a fixed input size
- Baseline: 18.4ms per call (median over 1,000 runs)
- Secondary metric: peak memory
- Constraints: keep output identical

Prompt:

> I need measurable CPU improvements for `score(items)`. Baseline: median 18.4ms per call on input size N=10,000, measured with benchmark harness ____. Secondary: peak memory must not increase. Scope: only `score()` and its helper functions. Propose 3–5 changes. For each, include a micro-benchmark plan and an explanation tied to the code's operations (e.g., nested loops, repeated hashing, string conversions).

Assistant response that earns trust:

- **Change:** Replace repeated `hash(key)` inside an inner loop with a precomputed map.
- **Mechanism:** Hashing is O(1) but expensive; doing it N times inside another loop multiplies cost.
- **Measure:** Add a benchmark variant that isolates hashing cost; compare overall runtime and allocation counts.
- **Risk:** If keys are mutated or depend on iteration order, precomputation must preserve semantics; add a correctness test.

Example 3: Query optimization with explainable metrics

Scenario: A database query is slow.

- Primary metric: query execution time and endpoint p95
- Baseline: query takes 220ms; endpoint p95=500ms
- Constraints: same results, same filters

Prompt:

> Optimize the SQL used by `get_orders(user_id, status)`. Baseline: query execution time 220ms (measured via DB logs) and endpoint p95=500ms. Scope: only the query and its indexes (if needed). Constraints: same results, no schema redesign. Propose 3–5 changes. For each, include the exact query rewrite or index suggestion, what it changes in the execution plan, and how to verify using EXPLAIN and a before/after benchmark.

What to ask for explicitly:

- "Show the expected execution plan differences (e.g., fewer rows scanned, index usage)."
- "Provide an acceptance threshold, like p95 improvement of at least 10%."

A simple scoring rubric for assistant suggestions

When you receive proposals, rank them using a rubric you control. This prevents you from accepting "clever" changes that don't move metrics.

- **Expected impact on primary metric** (high/medium/low)
- **Measurability** (clear before/after plan)
- **Risk level** (how likely to break correctness or increase tail latency)
- **Effort** (how many lines and how localized)

You can even ask the assistant to output a table with these fields, but the key is that the assistant must tie each change to a measurable verification step.

Common prompt mistakes (and how to fix them)

- **Mistake:** "Optimize this code."
  - **Fix:** Add baseline numbers and a primary metric.
- **Mistake:** "Make it faster."
  - **Fix:** Ask for ranked changes with mechanisms and measurement steps.
- **Mistake:** No constraints.
  - **Fix:** State correctness and resource limits so the assistant doesn't trade one problem for another.
- **Mistake:** No validation plan.
  - **Fix:** Require exact benchmark/load-test procedures and acceptance thresholds.

When your prompt forces the assistant to propose changes that can be tested, you get suggestions that are both practical and falsifiable. That's the whole point: optimization that survives contact with a stopwatch.

6.3 Optimizing Algorithms and Data Structures With Examples

When people say “optimize,” they often mean “make it faster,” but the fastest path depends on *where* time is spent. Algorithms control how much work you do; data structures control how you find and update things. The trick is to change the right lever, not just rewrite code.

Mind map: where speed comes from

Click here to view the mind map: Optimization levers

Start with a quick diagnosis

Before changing code, answer three questions:

1. **What operation dominates?** (e.g., searching, sorting, hashing, copying.)
2. **How often does it happen?** (per request, per loop iteration, per item.)
3. **What’s the input shape?** (many duplicates, mostly sorted, small N, huge N.)

A common pattern: you have a correct solution that repeatedly performs an expensive operation inside a loop. Moving that expensive work outside the loop or replacing it with a better data structure often yields the biggest gains.

Example 1: Replace repeated scans with a hash map

Problem: You need to count how many times each value appears.

Naive approach: for each element, scan the array to count matches.

```
def counts_naive(nums):
    out = {}
    for x in nums:
        c = 0
        for y in nums:
            if y == x:
                c += 1
        out[x] = c
    return out
```

This does about n^2 comparisons. A hash map lets you update counts in one pass.

```
from collections import defaultdict

def counts_fast(nums):
    out = defaultdict(int)
    for x in nums:
        out[x] += 1
    return dict(out)
```

Why it works: each update is expected $O(1)$, so the loop is $O(n)$. The data structure choice (hash table) removes the repeated scan.

Practical note: if keys are complex objects, ensure they’re hashable and have stable equality semantics.

Example 2: Use the right ordering structure for “top-k”

Problem: Find the largest k numbers in a stream.

Sorting the entire list is $O(n \log n)$. If k is small, a min-heap keeps only the best candidates.

```python
import heapq

def top_k(nums, k):
    heap = []
    for x in nums:
        if len(heap) < k:
            heapq.heappush(heap, x)
        elif x > heap[0]:
            heapq.heapreplace(heap, x)
    return sorted(heap, reverse=True)
```

Complexity: heap operations are $O(\log k)$, so total time is $O(n \log k)$. Memory is $O(k)$.

When to use: when you only need a small subset, not a full ordering.

Example 3: Avoid quadratic behavior in string building

Problem: Build a long string by repeatedly appending.

In Python, repeated `+=` on strings can create many intermediate strings.

```python
def build_slow(parts):
    s = ""
    for p in parts:
        s += p
    return s
```

Better: collect pieces and join once.

```python
def build_fast(parts):
    return "".join(parts)
```

Why it works: `join` computes the final size and writes once, avoiding repeated copying.

Algorithm angle: the work becomes linear in total characters rather than repeatedly re-copying partial results.

Example 4: Choose between set and list for membership

Problem: Check whether each query value exists in a collection.

Using a list:

```python
def exists_list(items, queries):
    out = []
    for q in queries:
        out.append(q in items)
    return out
```

Membership in a list is $O(n)$, so total is $O(nm)$ for n items and m queries.

Convert once to a set:

```python
def exists_set(items, queries):
    s = set(items)
    return [q in s for q in queries]
```

Now membership is expected $O(1)$, so total is $O(n + m)$.

Tradeoff: sets use more memory, and they require elements to be hashable.

Example 5: Data structure choice for “range queries”

Problem: You need to answer many queries of the form “sum from index l to r.”

Naive approach: sum each range.

```
def range_sum_naive(arr, queries):
    out = []
    for l, r in queries:
        s = 0
        for i in range(l, r + 1):
            s += arr[i]
        out.append(s)
    return out
```

If each query spans many elements, this can be $O(nm)$. Prefix sums make each query constant time.

```
def range_sum_fast(arr, queries):
    pref = [0]
    for x in arr:
        pref.append(pref[-1] + x)
    out = []
    for l, r in queries:
        out.append(pref[r + 1] - pref[l])
    return out
```

Complexity: building prefix sums is $O(n)$; each query is $O(1)$. Total is $O(n + m)$.

Correctness detail: prefix sums rely on consistent indexing; off-by-one errors are the usual bug.

Mind map: common upgrades and what they replace

Click here to view the mind map: common upgrades and what they replace

How to ask an AI assistant for algorithm/data-structure improvements

A useful request includes: the goal (faster), constraints (memory/time), input size, and what you already tried. For example:

- “Given this function and typical input sizes (n≈50k, queries≈200k), propose an algorithmic change and show complexity before/after.”
- “Identify any $O(n^2)$ patterns and rewrite using a map/set/heap/prefix sums where appropriate.”

Then verify the result with tests that cover edge cases like empty inputs, duplicates, and boundary indices.

A small checklist for safe optimization

- **Preserve behavior:** confirm outputs match for random small inputs.
- **Watch memory:** a faster structure can still be a net loss if it grows too large.
- **Check assumptions:** heap methods depend on comparisons; hash maps depend on correct hashing/equality.
- **Measure the right thing:** if you optimize the wrong part, the code may get more complex without speed gains.

Algorithm and data structure optimization is mostly about removing repeated expensive operations and choosing structures that match the access pattern. Once you can point to the dominant operation, the “best” change becomes much easier to justify.

6.4 Reducing Unnecessary Work in Loops and Queries

When code is slow, it’s often doing “extra” work: repeating the same computation, fetching more data than needed, or running queries inside loops. The goal here is simple: move work out of hot paths, reduce the amount of work per iteration, and make the database do less guessing.

The core idea: count the work you repeat

A loop multiplies costs. If an operation inside the loop is (O(1)) but you do it 1,000,000 times, it’s suddenly (O(1,000,000)) in practice. Similarly, a query that takes 20 ms inside a loop of 200 items becomes 4 seconds, even if each query is “fast enough” alone.

A useful mental checklist for each loop:

- What is recomputed every iteration but could be computed once?
- What is fetched every iteration but could be fetched in one batch?
- What is filtered in memory but could be filtered in the query?
- What is checked repeatedly but could be short-circuited?

Mind map: where unnecessary work hides

Mind map: Reducing Unnecessary Work in Loops and Queries

Click here to view the mind map: Unnecessary work

1) Move invariant computation out of loops

If a value doesn't change across iterations, compute it once.

Before (recomputes a regex and a derived value):

```
import re

def normalize_emails(emails):
    pattern = re.compile(r"\s+")
    out = []
    for e in emails:
        out.append(pattern.sub("", e.lower()))
    return out
```

After (compile once, keep loop focused):

```
import re

pattern = re.compile(r"\s+")

def normalize_emails(emails):
    out = []
    for e in emails:
        out.append(pattern.sub("", e.lower()))
    return out
```

This is small, but it's the kind of small that becomes large when loops are hot.

2) Replace "query per item" with batching

The classic performance bug is the N+1 query pattern: one query to get items, then one query per item to fetch related data.

Before (N+1):

```
# orders: list of order objects
for order in orders:
    items = db.query(
        "SELECT * FROM order_items WHERE order_id = %s",
        (order.id,)
    )
    process(order, items)
```

After (batch once, then group in memory):

```
order_ids = [o.id for o in orders]
items = db.query(
    "SELECT * FROM order_items WHERE order_id = ANY(%s)",
    (order_ids,)
)

items_by_order = {}
for it in items:
    items_by_order.setdefault(it.order_id, []).append(it)

for order in orders:
    process(order, items_by_order.get(order.id, []))
```

The second version does more work in memory, but it replaces many round trips with one. That trade is usually worth it when latency dominates.

3) Fetch less: select only what you need

Over-fetching is a silent tax. If you only need IDs, don't pull full rows.

Before (fetches full rows):

```
SELECT *
FROM users
WHERE status = 'active';
```

After (fetches only what's used):

```
SELECT id
FROM users
WHERE status = 'active';
```

In application code, this also reduces serialization/deserialization costs.

4) Filter in the database, not after the fact

If you fetch a broad set and then filter in memory, you pay for transferring and parsing data you'll discard.

Before (broad fetch, then filter):

```
rows = db.query("SELECT * FROM events WHERE created_at >= %s", (since,))
filtered = [r for r in rows if r.type == "purchase" and r.amount > 0]
```

After (filter in SQL):

```
filtered = db.query(
    "SELECT * FROM events WHERE created_at >= %s AND type = %s AND amount > 0",
    (since, "purchase")
)
```

This tends to be faster and simpler to reason about because the "rules" live in one place.

5) Use the right in-memory data structures

Sometimes the database isn't the problem; the loop is doing linear searches.

Before (membership check is (O(n)) each time):

```
allowed = ["read", "write", "delete"]

for action in actions:
    if action in allowed:  # linear scan each time
        handle(action)
```

After (membership check is (O(1)) average):

```
allowed = {"read", "write", "delete"}

for action in actions:
    if action in allowed:
        handle(action)
```

Similarly, if you repeatedly look up objects by ID, build a map once.

6) Short-circuit when you can

Loops often keep going after the answer is known.

Before (keeps scanning):

```
found = False
for x in xs:
    if x == target:
        found = True
# loop ends, but we already knew
```

After (stop early):

```
for x in xs:
    if x == target:
        return True
return False
```

This is especially effective when the target is usually near the beginning.

7) Avoid rebuilding the same map repeatedly

If you build a dictionary inside a loop, you're paying that cost every iteration.

Before (rebuilds map each time):

```
for group in groups:
    by_id = {u.id: u for u in users}  # rebuilt every group
    process(group, by_id)
```

After (build once):

```
by_id = {u.id: u for u in users}
for group in groups:
    process(group, by_id)
```

8) Make query patterns predictable

Even without changing SQL, you can reduce unnecessary work by making queries consistent.

- Prefer stable ordering only when you need it.
- Avoid calling the same query repeatedly with different parameters when a single query can cover all parameters.
- Use pagination carefully: fetching 10 pages of 100 rows is 1000 rows total, but fetching only the page you need is 100.

Practical mini-workflow: optimize with evidence

1. Identify the hot loop (where time is spent).
2. Count repeated operations (per-iteration work and per-item queries).
3. Apply one change at a time: batch queries, move invariants, reduce selected columns, or fix data structures.
4. Re-run the same test or benchmark to confirm the improvement.

A good rule: if you can't point to the repeated work, you're guessing. If you can, you can usually remove it.

Quick examples to practice

- **Batching:** Replace per-order item queries with one query using `WHERE order_id IN (...)`.
- **Filtering:** Move `type == 'purchase'` and `amount > 0` into SQL.
- **Data structures:** Convert `allowed` from list to set for fast membership checks.
- **Short-circuiting:** Use `break` / `return` once the condition is satisfied.

Reducing unnecessary work is mostly about discipline: keep loops small, keep queries few, and keep data movement proportional to what you actually use.

6.5 Avoiding Performance Regressions With Benchmarks

Performance work is only "done" when you can prove it didn't get worse elsewhere. Benchmarks help you catch regressions early, but only if they're designed to be trustworthy and repeatable. The goal is simple: measure the same thing, the same way, every time, and compare results using consistent rules.

Why regressions happen (and why benchmarks catch them)

A regression often comes from a change that improves one path while quietly harming another. Examples include:

- A refactor that adds allocations in a hot loop.
- A "cleaner" query that removes an index-friendly filter.
- A new logging statement that formats strings even when logs are disabled.
- A concurrency change that increases contention under load.

Benchmarks catch these because they create a controlled comparison between "before" and "after." Without them, you're left with vibes and production surprises.

Mind map: benchmark strategy

Click here to view the mind map: Benchmarking to prevent regressions

Step 1: Pick benchmark targets that represent real cost

Start with a short list of "critical paths," not everything you can measure. Good targets are:

- Called frequently (hot path)
- Expensive per call (heavy computation)
- User-visible (request latency)
- Operationally risky (memory growth, GC pressure)

Example (micro + service):

- Micro: `parseLine()` in a log ingestion pipeline.
- Service: "ingest 10k lines" end-to-end, including parsing and writing.

If you only benchmark the micro function, you might miss that the change increases database time. If you only benchmark end-to-end, you might not know whether the slowdown is parsing, serialization, or I/O.

Step 2: Make the benchmark deterministic enough to compare

Benchmarks fail when inputs vary wildly between runs. Use fixed datasets and stable parameters.

Example: stable input generation

- Pre-generate a list of 100,000 representative inputs.
- Reuse it for both baseline and candidate.
- Avoid random generation inside the benchmark loop.

Also keep configuration constant:

- Same feature flags.
- Same thread count.
- Same cache state policy (either warm caches for both, or cold caches for both).

Step 3: Warm up and measure steady-state

Many runtimes behave differently at startup due to JIT compilation, caching, and allocator behavior. A common mistake is measuring the first few iterations and treating them as "the truth."

Practical rule:

- Run a warmup phase until results stabilize.
- Measure a fixed number of iterations after warmup.
- Use the same warmup and measurement settings for baseline and candidate.

Step 4: Control noise and repeat runs

Noise comes from scheduling, background processes, and thermal/power effects. You can't eliminate it entirely, but you can reduce it.

Example checklist:

- Run on a quiet machine or dedicated CI runner.
- Pin thread counts to a known value.
- Repeat the benchmark multiple times and compare distributions.

If your benchmark shows a 2% swing between runs, don't set a 1% regression threshold. Your threshold must be larger than your measurement noise.

Step 5: Compare results with clear acceptance rules

A benchmark report is not automatically a decision. Define rules that are easy to apply.

Example acceptance policy (simple and effective):

- Fail if median latency increases by more than 5%.
- Fail if allocations increase by more than 10%.
- Allow small changes if both CPU time and allocations improve.

You can also separate metrics:

- CPU time regression might be acceptable if memory drops significantly and vice versa, but only if you've decided that tradeoff.

Step 6: Use "before vs after" baselines correctly

Benchmarks are most useful when you compare against a known baseline.

Example workflow:

1. Record baseline results for the current main branch.
2. Run the same benchmark suite for the PR.
3. Compute deltas for each metric.
4. Gate merges based on acceptance rules.

Avoid comparing two unrelated commits. If the baseline is stale, you may miss a regression that already exists.

Step 7: Benchmark the change, not the benchmark

Sometimes the benchmark accidentally measures the benchmark harness.

Example pitfall:

- Creating large objects inside the timed section.
- Logging inside the timed section.
- Using a data structure that grows during the benchmark.

Fix:

- Move setup outside the timed region.
- Pre-allocate buffers.
- Disable debug logging or ensure it doesn't format strings when disabled.

Step 8: Concrete example—catching an allocation regression

Suppose a developer "simplifies" code by returning a new string each time.

Before (conceptual):

- Reuse a buffer and write into it.
- Return a view or copy only when needed.

After (conceptual):

- Build a new string every call.

A benchmark should include an allocation metric, not just time.

Example benchmark design:

- Input: fixed set of 10,000 lines.
- Measure: time per parse + allocations per parse.
- Run: baseline and candidate under the same settings.

Expected outcome:

- Time might look similar for small runs.
- Allocations will spike, leading to more GC work under realistic load.

This is exactly why you should track allocations or memory-related metrics when optimizing.

Step 9: Concrete example—preventing a query regression

A refactor might change a query shape.

Benchmark approach:

- Use a dataset with realistic distribution (not just a few rows).
- Measure query latency and CPU time.
- Include an "explain-style" check in the benchmark output if your stack supports it.

Example acceptance rule:

- Fail if p95 query latency increases by more than 5%.
- Fail if the query plan stops using the expected index.

Even if total time changes only slightly, a plan change can cause future regressions when data grows.

Mind map: what to record for each benchmark

Click here to view the mind map: Benchmark evidence to store

Step 10: Make it hard to ignore regressions

Benchmarks should be part of the workflow, not a separate ritual.

Practical gating pattern:

- Run a small “regression suite” on every PR.
- Run a larger suite on demand or nightly.
- Fail the PR when acceptance rules are violated.

If a change legitimately slows something down, require an explicit justification and a compensating improvement elsewhere, backed by benchmark evidence.

Summary

Avoiding performance regressions with benchmarks comes down to three habits: measure the right things, make comparisons fair, and enforce clear acceptance rules. When your benchmarks are stable and your thresholds match your measurement noise, you can trust the results enough to prevent slowdowns from sneaking into the codebase.

7. Optimizing Code Paths and Resource Usage

7.1 Minimizing Memory Allocations With Practical Patterns

Memory allocations are expensive not just because they cost time, but because they create churn: more objects means more work for the allocator, the garbage collector, and the CPU caches. The goal here is simple: reduce the number of allocations on hot paths, and when you must allocate, do it predictably and in larger chunks.

Why allocations happen (and where to look)

Allocations usually come from a few common sources:

- **Creating new objects per iteration** (e.g., building temporary strings, lists, or wrapper objects inside loops).
- **Implicit allocations** from language features (e.g., string concatenation, iterator adapters, boxing).
- **Resizing dynamic containers** (e.g., vectors growing repeatedly, hash maps rehashing).
- **Copying buffers** (e.g., converting between string/byte representations, slicing that forces copies).

A practical workflow is to measure first, then fix. Use your profiler to find the allocation hotspots, then confirm by reading the code around those lines. If you can’t profile, add lightweight counters (e.g., track how many times a function is called and how many bytes it returns) to narrow down likely culprits.

Mind map: allocation-minimizing patterns

Click here to view the mind map: Minimize Memory Allocations

Pattern 1: Reuse buffers instead of recreating them

If a function builds a temporary byte array or string repeatedly, reuse a single buffer.

Example (JavaScript/TypeScript):

```
function formatLine(items: string[]): string {
  // Bad: creates many intermediate strings via join + template work
  return `items: ${items.join(', ')}`;
}
```

A better approach is to build into a reusable buffer-like structure. In JS you can’t fully avoid string creation, but you can reduce intermediate work by using a single pass.

```
function formatLine(items: string[]): string {
  let out = 'items: ';
  for (let i = 0; i < items.length; i++) {
    if (i !== 0) out += ', ';
    out += items[i];
  }
  return out;
}
```

This still allocates the final string, but it avoids creating extra arrays from `join`. In performance-critical code, you'd go further by writing into a preallocated `Uint8Array` (when you control encoding) or using a streaming writer.

Reasoning: fewer intermediate allocations means less GC pressure and fewer cache misses. Even when the final output must exist, the "in-between" objects are optional.

Pattern 2: Preallocate capacity for dynamic containers

Repeated resizing causes allocations and copies. Preallocate when you know (or can estimate) the size.

Example (C++):

```
std::vector<int> buildIds(const std::vector<int>& src) {
  std::vector<int> ids;
  for (int x : src) {
    if (x > 0) ids.push_back(x);
  }
  return ids;
}
```

If you expect many positives, reserve capacity.

```
std::vector<int> buildIds(const std::vector<int>& src) {
  std::vector<int> ids;
  ids.reserve(src.size());
  for (int x : src) {
    if (x > 0) ids.push_back(x);
  }
  return ids;
}
```

Reasoning: `reserve` reduces reallocations and avoids copying already-inserted elements. This is one of the highest ROI fixes because it's usually safe and easy.

Pattern 3: Avoid per-iteration string building

String concatenation in loops often creates many temporary strings. Prefer strategies that build once, or write to a buffer.

Example (C#):

```
string BuildLog(List<string> parts) {
  string s = "";
  foreach (var p in parts) s += p + ";";
  return s;
}
```

Use a `StringBuilder` and reuse it when possible.

```
string BuildLog(List<string> parts) {
  var sb = new System.Text.StringBuilder(parts.Count * 8);
  for (int i = 0; i < parts.Count; i++) {
    if (i > 0) sb.Append(';');
    sb.Append(parts[i]);
  }
  return sb.ToString();
}
```

Reasoning: `StringBuilder` grows with amortized cost, and it avoids creating a new string for every `+=`. The initial capacity hint reduces growth reallocations.

Pattern 4: Prefer slices/views over copies

Copying data into new arrays is a common hidden allocation. If your language supports it, pass around views.

Example (Rust):

```
fn count_prefixes(lines: Vec<String>) -> usize {
  let mut c = 0;
  for line in lines {
    let prefix = line[0..3].to_string();
    if prefix == "GET" { c += 1; }
  }
  c
}
```

Avoid creating a new `String` for the prefix.

```
fn count_prefixes(lines: Vec<String>) -> usize {
  let mut c = 0;
  for line in lines {
    let prefix = &line[0..3];
    if prefix == "GET" { c += 1; }
  }
  c
}
```

Reasoning: `&line[0..3]` is a view into existing data. You still do bounds checks, but you avoid allocating a new string each iteration.

Pattern 5: Reduce boxing and temporary wrappers

Boxing turns value types into heap objects. Iterator adapters can also allocate depending on the language and implementation.

Example (Java):

```
int sum(int[] a) {
  return java.util.stream.IntStream.of(a).boxed().mapToInt(x -> x).sum();
}
```

Avoid boxing by staying in primitive streams.

```
int sum(int[] a) {
  return java.util.stream.IntStream.of(a).sum();
}
```

Reasoning: boxing creates objects; even if they're short-lived, they still cost allocations and GC work.

Pattern 6: Pool only when reuse is hard

Pooling can reduce allocations, but it adds complexity and can cause memory to stick around longer than intended. Use it when you repeatedly allocate the same kind of buffer and you can bound the pool size.

Example (C# concept):

- Rent a byte buffer from a pool.
- Use it for the operation.
- Return it in a `finally` block.

This pattern reduces allocations for large temporary buffers while keeping lifetime controlled.

Reasoning: pooling is most effective for large, frequently reused buffers. For small objects, the overhead of pooling can outweigh the benefits.

Pattern 7: Stream output instead of collecting

Collecting results into a list or string often allocates multiple times: the container, then each element, then the final join/serialization.

Example (Python):

```
def render(items):
    lines = []
    for x in items:
        lines.append(f"item={x}")
    return "\n".join(lines)
```

Stream into an output writer (or yield lines to the caller).

```
def render_lines(items):
    for x in items:
        yield f"item={x}"
```

Then the caller can write incrementally.

Reasoning: you avoid storing all intermediate strings at once. Even if each line is still a string, you reduce peak memory and container allocations.

Verification: make allocation reduction measurable

After applying patterns, confirm with evidence:

- Compare allocation counts or allocated bytes before/after.
- Run the same workload that previously triggered allocations.
- Add a regression check for the hot function (e.g., assert allocation count stays under a threshold in a benchmark harness).

A good rule: if you can't point to a specific allocation hotspot and show it moved, you probably fixed something else (or only improved average case).

Quick checklist for hot paths

- No new containers inside tight loops (or preallocate them).
- No per-iteration string concatenation (use a builder or buffer).
- Prefer views/slices over copying.
- Avoid boxing and unnecessary wrapper objects.
- Stream results when you don't need them all at once.
- Pool only for large, repeated buffers, with bounded lifetime.

When these patterns are applied together, allocation reduction usually comes with a second benefit: more predictable CPU behavior and fewer cache misses, because the program spends less time managing memory and more time doing the actual work.

7.2 Improving I/O Efficiency With Batching and Streaming

I/O work is often slow not because the CPU is weak, but because waiting on disks, networks, or syscalls dominates runtime. Two practical levers help immediately: **batching** (fewer, larger I/O operations) and **streaming** (processing data as it arrives instead of storing it all first). Used together, they reduce overhead and keep memory usage predictable.

Why batching helps

Batching reduces per-operation costs: each read/write typically pays a fixed "tax" (system call overhead, protocol framing, lock contention, buffering churn). If you write 10,000 small chunks, you pay that tax 10,000 times. If you write 10 chunks of 1,000 items each, you pay it 10 times.

Rule of thumb: batch size should be large enough to amortize overhead, but small enough to avoid long pauses, excessive memory, or timeouts.

Example: batching writes to a file

Suppose you're logging events. A naive approach writes each event immediately.

- Naive: 1 write per event
- Batched: 1 write per N events

```
# Naive
for event in events:
    f.write(event + "\n")

# Batched
buf = []
for event in events:
    buf.append(event)
    if len(buf) >= 1000:
        f.write("\n".join(buf) + "\n")
        buf.clear()
if buf:
    f.write("\n".join(buf) + "\n")
```

This changes the number of writes dramatically. It also improves throughput when the underlying storage benefits from larger sequential writes.

Mind map: batching and streaming

Click here to view the mind map: I/O Efficiency

Why streaming helps

Streaming avoids the "collect everything, then process" pattern. It reduces peak memory and can overlap work: while you're processing early chunks, the producer can continue sending later chunks.

Streaming is especially valuable when:

- input is large or unbounded (logs, uploads, message streams)
- you only need a transformation or filtering
- you can process each chunk independently or with small state

Example: streaming a large file transform

Imagine converting a CSV file to JSON lines. Loading the entire CSV first wastes memory.

```
import csv, json

with open("input.csv", newline="") as src, open("out.jsonl", "w") as dst:
    reader = csv.DictReader(src)
    for row in reader:
        dst.write(json.dumps(row) + "\n")
```

This reads and processes row-by-row. Peak memory stays low, and the output begins immediately.

Combining batching + streaming: the practical pipeline

The most effective pattern is often:

1. **stream** input in chunks
2. **batch** the expensive output operation
3. **flush** when thresholds are met (count, size, or time)

Example: streaming lines, batching database inserts

Assume you ingest newline-delimited events and insert them into a database. You want to avoid one insert per line, but you also don't want to read the entire file into memory.

```
import time

BATCH_SIZE = 500
MAX_WAIT_S = 2.0

batch = []
last_flush = time.time()

with open("events.log") as f:
    for line in f:
        batch.append(line.rstrip("\n"))
        now = time.time()
        if len(batch) >= BATCH_SIZE or (now - last_flush) >= MAX_WAIT_S:
            db_insert_many(batch)  # one bulk insert
            batch.clear()
            last_flush = now

if batch:
    db_insert_many(batch)
```

This balances throughput and latency. The time-based flush prevents a slow stream from waiting forever to reach the count threshold.

Choosing batch size and chunk size without guesswork

Batching and streaming both require sizes, but you can choose them systematically.

1. **Measure the per-operation overhead**: time a single write/insert and compare it to writing N items.
2. **Find the knee point**: increase batch size until throughput gains flatten.
3. **Respect latency constraints**: if users expect updates within a second, don't batch for ten seconds.
4. **Watch memory**: batch size times average item size should fit comfortably with other allocations.

A simple way to reason about it:

- If each item is small, overhead dominates, so batching helps a lot.
- If each item is large, the bottleneck may shift to bandwidth, so batching still helps but with diminishing returns.

Handling partial batches and correctness

Batching introduces edge cases that are easy to miss:

- the final batch may be smaller than the threshold
- errors may occur mid-batch
- retries can cause duplicates if you don't have idempotency

Practical mitigations:

- always flush the remainder at the end
- wrap bulk operations so you can retry safely
- include a stable event ID and use upsert/unique constraints when possible

Streaming with backpressure: don't outrun your consumer

Streaming doesn't automatically guarantee efficiency. If the consumer is slower than the producer, you can still accumulate memory.

A common fix is to use bounded buffers:

- producer reads/receives
- it appends to a queue with a max size
- when full, the producer waits (or slows down)

This keeps memory bounded and prevents the system from turning "streaming" into "buffering with extra steps."

Quick checklist

- Batch operations that have fixed per-call overhead (writes, inserts, requests).
- Stream large inputs to keep peak memory stable.
- Flush batches on both **size** and **time** when latency matters.
- Ensure correctness for partial batches and retries.
- Use bounded queues to avoid unbounded buffering.

When you apply batching to the expensive boundary and streaming to the data flow, you get the best of both worlds: fewer I/O calls and steady memory usage, with behavior that stays predictable under real workloads.

7.3 Handling Concurrency Correctly With Examples

Concurrency bugs are rarely "mysterious." They're usually the result of a specific mismatch between *what you think happens* and *what the scheduler actually does*. This section focuses on practical patterns that keep shared state consistent, keep work bounded, and make failures predictable.

The core problem: shared state + interleaving

When two tasks run at the same time, their operations can interleave in ways that break assumptions. A classic example is "check then act."

- Task A checks `balance >= amount`.
- Task B withdraws funds.
- Task A proceeds anyway.

Even if each task is correct in isolation, the interleaving is not.

Mind map: concurrency correctness

Click here to view the mind map: Concurrency Correctness

Pattern 1: Prefer isolation (no shared mutable state)

If each task can work on its own data and only combine results at the end, you avoid most race conditions.

Example (Python-like pseudocode):

- Each worker computes a partial sum.
- The main thread combines results.

```
# Each worker uses only local variables.

def partial_sum(nums, start, end):
    s = 0
    for i in range(start, end):
        s += nums[i]
    return s

# Combine after all workers finish.
parts = [executor.submit(partial_sum, nums, a, b) for a,b in ranges]
result = sum(f.result() for f in parts)
```

Reasoning: there is no shared variable that multiple workers update, so there's nothing to synchronize.

Pattern 2: Use a lock to protect an invariant

Locks are for protecting a *rule* that must always be true. For a bank account, the invariant might be: `balance` never goes negative.

Example (thread-safe withdraw):

```
import threading

class Account:
    def __init__(self, balance):
        self.balance = balance
        self.lock = threading.Lock()

    def withdraw(self, amount):
        with self.lock:  # mutual exclusion
            if amount > self.balance:
                raise ValueError("insufficient funds")
            self.balance -= amount
```

Reasoning: the check and the update happen under the same lock, so no other thread can interleave between them.

Pattern 3: Avoid "check-then-act" races

If you split the check and the action without synchronization, you reintroduce the race.

Buggy example:

```
# Not safe: check and update are separate.
if amount <= account.balance:
    account.balance -= amount
```

Fix: put both operations under the same lock (or use an atomic primitive if your language provides one).

Pattern 4: Use message passing for coordination

Instead of sharing state, share *events*. A single consumer owns the state; producers send requests.

Example (single-threaded account owner + worker requests):

```
import queue

requests = queue.Queue()

def account_owner():
    balance = 100
    while True:
        msg = requests.get()
        if msg is None:
            return
        amount, reply_q = msg
        if amount > balance:
            reply_q.put(False)
        else:
            balance -= amount
            reply_q.put(True)
```

Reasoning: only `account_owner` touches `balance`. Producers never race on the invariant.

Pattern 5: Bound concurrency with semaphores

Unbounded concurrency can cause resource exhaustion, which then looks like "random" failures.

Example (limit concurrent HTTP calls):

```
import asyncio

sem = asyncio.Semaphore(10)

async def fetch(url):
    async with sem:
        return await http_get(url)
```

Reasoning: the semaphore enforces a hard cap, so you can reason about memory, file descriptors, and upstream load.

Pattern 6: Make ordering explicit with synchronization

If one task produces data and another consumes it, you need a happens-before relationship.

Example (producer sets result, consumer waits):

```
import threading

ready = threading.Event()
value = None

def producer():
    global value
    value = compute()
    ready.set()  # publish

def consumer():
    ready.wait()  # acquire
    return value
```

Reasoning: `ready.set()` and `ready.wait()` provide the ordering guarantee that the consumer won't read `value` before it's assigned.

Pattern 7: Prevent deadlocks with consistent lock ordering

Deadlocks happen when tasks hold multiple locks in different orders.

Bad pattern:

- Task A locks `lock1`, then `lock2`.
- Task B locks `lock2`, then `lock1`.

Fix: always acquire locks in the same global order.

```
def transfer(a, b, amount):
    first, second = (a, b) if id(a) < id(b) else (b, a)
    with first.lock:
        with second.lock:
            if a.balance < amount:
                raise ValueError("insufficient funds")
            a.balance -= amount
            b.balance += amount
```

Reasoning: consistent ordering removes the circular wait condition.

Pattern 8: Handle cancellation and cleanup safely

If a worker can be cancelled, ensure it releases locks and stops promptly.

Example (lock release via context manager):

```
def update(shared, lock, stop_event):
    with lock:
        while not stop_event.is_set():
            shared['count'] += 1
            break  # keep example short
```

Reasoning: the context manager guarantees lock release even when the surrounding control flow exits early.

Pattern 9: Test concurrency with targeted stress

You don't need chaos testing to find real bugs. You need tests that force interleavings.

Example test idea:

- Run many threads that call `withdraw(1)`.
- Start them at the same time using a barrier.
- Assert the final balance equals `initial - successful_withdrawals`.

```
# Pseudocode for a concurrency test
barrier = Barrier(n)
results = []

def worker():
    barrier.wait()
    try:
        account.withdraw(1)
        results.append(True)
    except ValueError:
        results.append(False)

run_threads(worker)
assert account.balance == initial - sum(results)
```

Reasoning: the barrier increases the chance of overlapping operations, and the final invariant check catches lost updates.

Quick checklist for correct concurrency

- Protect invariants with a single synchronization mechanism.
- Avoid check-then-act without atomicity.
- Prefer isolation or message passing over shared mutable state.
- Bound concurrency with semaphores.
- Use explicit ordering (events/futures/joins) for producer-consumer flows.
- Acquire multiple locks in a consistent order.
- Ensure cleanup happens even on early exit.
- Write tests that force overlap and verify invariants.

Concurrency is manageable when you treat it like engineering: define invariants, choose a synchronization strategy, and verify with tests that actually exercise interleavings.

7.4 Preventing Leaks and Managing Lifecycles

Memory leaks and resource leaks are the kind of bugs that don't always crash your program immediately. Instead, they slowly drain it: memory grows, file handles run out, database connections pile up, and "it worked yesterday" becomes "it fails under load." The fix is rarely one magic line; it's a lifecycle discipline—knowing what owns what, when it starts, and when it ends.

The lifecycle model: who owns the resource?

Start by naming the resource and its owner. A resource might be:

- **Memory** (objects, buffers, caches)
- **Handles** (files, sockets, streams)

- **External capacity** (DB connections, HTTP clients)
- **Background work** (timers, goroutines, tasks)

A simple rule helps: **the owner is responsible for cleanup**. If multiple parts can create the resource, decide which part also performs cleanup. If you can't decide, you'll eventually get either double-cleanup (errors) or no cleanup (leaks).

Example (Node.js):

```
// Bad: stream never closed on early return
function readFirstLine(path) {
  const stream = fs.createReadStream(path);
  let line = null;
  stream.on('data', chunk => {
    line = chunk.toString().split('\n')[0];
    return line; // doesn't stop the stream
  });
  return line; // returns before data arrives
}
```

The lifecycle problem is twofold: the function returns before the stream finishes, and there's no explicit close/destroy path. A better pattern is to use a single "done" path that always closes.

Example (Node.js):

```
async function readFirstLine(path) {
  const stream = fs.createReadStream(path);
  try {
    for await (const chunk of stream) {
      return chunk.toString().split('\n')[0];
    }
    return null;
  } finally {
    stream.destroy();
  }
}
```

Here, cleanup happens even when you return early.

Mind map: leak prevention and lifecycle management

Click here to view the mind map: Leak Prevention & Lifecycle Management

Common leak patterns (and what to do instead)

1) Early returns that skip cleanup

Early returns are fine when cleanup is guaranteed. The fix is to put cleanup in a `finally` block (or equivalent).

Example (Python):

```
def process_file(path):
    f = open(path, 'r')
    try:
        line = f.readline()
        if not line:
            return 0
        return len(line)
    finally:
        f.close()
```

If you're using `with open(...) as f:`, you get the same guarantee with less room for mistakes.

2) Event listeners that accumulate

If you attach listeners repeatedly (for example, per request) and never remove them, you get a leak even when memory "looks stable" at first. The lifecycle is "subscribe now, unsubscribe when done."

Example (JavaScript):

```
function attachOnce(emitter, event, handler) {
  emitter.on(event, handler);
  return () => emitter.off(event, handler);
}

const detach = attachOnce(emitter, 'data', onData);
// later, when request ends
detach();
```

In request-scoped code, make detaching part of the request completion path.

3) Caches without eviction

A cache is a memory lifecycle decision. If you never evict, the cache becomes a memory leak by design. Use bounded caches (size limits) or time-based expiry.

Example (Java):

```
// Bounded cache: prevents unbounded growth
Cache<String, String> cache = Caffeine.newBuilder()
    .maximumSize(10_000)
    .build();
```

Even with eviction, you still need to ensure cached values don't hold onto large object graphs longer than necessary.

4) Connection leaks: "checked out" connections not returned

Connection pools are strict: if you forget to release, the pool eventually empties and requests block or fail. The lifecycle is "acquire, use, release," even on exceptions.

Example (Go):

```
conn, err := pool.Acquire(ctx)
if err != nil { return err }
defer conn.Release()

// use conn
rows, err := conn.Query(ctx, "SELECT ...")
if err != nil { return err }
defer rows.Close()
```

Notice the pattern: each acquired resource has a matching `defer` close/release.

Managing background work: timers, tasks, and cancellation

Background work leaks in two ways: it keeps running after the request ends, and it keeps references alive through closures.

A good lifecycle boundary is "start background work only within a scope that can cancel it." In practice, that means:

- Pass a cancellation token (or equivalent) into the worker.
- Stop the worker when the scope ends.
- Ensure the worker doesn't swallow cancellation.

Example (TypeScript with AbortController):

```
async function startWorker(signal: AbortSignal) {
  while (!signal.aborted) {
    await doUnitOfWork();
  }
}

const controller = new AbortController();
startWorker(controller.signal);
// later, when request ends
controller.abort();
```

If `doUnitOfWork()` can block, make it responsive to the same signal.

Using AI assistants safely for lifecycle fixes

When you ask an assistant to fix leaks, provide the lifecycle context: where the resource is created, how it's used, and what "done" means. Otherwise, it may add cleanup in the wrong place.

A helpful prompt structure is:

- Resource type (stream, handle, connection, listener)
- Creation site and scope
- Failure/early-return paths
- Expected cleanup behavior
- Language and any existing patterns (e.g., `with`, `try/finally`, RAII)

Example prompt (copy/paste style):

> Review this function for resource leaks. The function creates a stream and returns early when it finds a match. Ensure the stream is always closed/destroyed on success and on errors. Show the corrected code.

Verification: prove the lifecycle is correct

Don't rely on "it seems fine." Verification should target the failure modes you fixed.

- Add tests that force early returns and exceptions.
- Run a stress test that repeatedly exercises the lifecycle boundary.
- Check observable signals: open handles, pool usage, heap growth.

Example test idea (language-agnostic):

- Call the function 10,000 times with inputs that trigger early return.
- Assert that the number of open handles stays constant.
- Assert that the pool never hits "exhausted."

Practical checklist

- **Name the resource** and its owner.
- **Pair every acquire with a guaranteed release/close**.
- **Use a single exit path** or a `finally` / `defer` cleanup.
- **Unsubscribe listeners** when the scope ends.
- **Bound caches** and avoid holding large graphs unnecessarily.
- **Pass cancellation** into background work and stop it at scope end.
- **Test the failure paths**, not just the happy path.

Lifecycle management is mostly about being consistent. Once you treat cleanup as part of the design—not an afterthought—the leaks become much easier to prevent and much easier to spot.

7.5 Tuning Caching Strategies With Clear Invalidation Rules

Caching is only "optimization" when it stays correct. The trick is to make invalidation rules explicit, testable, and boring. This section shows how to tune cache behavior by choosing the right cache scope, defining what makes entries stale, and enforcing those rules consistently.

Start with the cache contract: what must be true?

Before changing code, write a one-sentence contract for each cache:

- **Cache contract:** "For input X, return value Y as long as data source Z has not changed since time T."

If you can't name the data source and the change signal, you'll end up with caches that "usually work" and fail at the worst time.

Choose the cache scope that matches the invalidation signal

Common scopes and their typical invalidation triggers:

- **Per-request cache (in-memory):** invalidation is automatic at the end of the request.
- **Per-user cache:** invalidation often ties to user profile updates or permissions changes.
- **Per-tenant / per-organization cache:** invalidation ties to tenant configuration or feature flags.
- **Global cache:** invalidation ties to dataset versioning or schema changes.

A practical rule: if you can't describe when an entry becomes wrong, don't cache it globally.

Use versioning when you can

Versioning turns invalidation into a simple comparison.

- Store a **data version** (e.g., `catalog_version`, `policy_version`, `feature_flags_version`).
- Include that version in the cache key or store it alongside the cached value.

Example (key includes version):

- Key: `product:{productId}:v{catalogVersion}`
- When catalog changes, `catalogVersion` increments, so old keys stop being used.

This approach avoids "guessing" staleness. It's also easy to test: bump the version and verify new results.

Use TTL when you must, but make it measurable

TTL (time-to-live) is a fallback when you can't get a reliable change signal.

- Pick TTL based on acceptable staleness and update frequency.
- Treat TTL as a safety net, not the primary correctness mechanism.

Example (TTL with conservative default):

- Cache "search suggestions" for 60 seconds.
- Cache "pricing rules" for 5 minutes.
- Cache "authorization decisions" for 0 seconds (or use versioning/explicit invalidation).

If correctness matters, TTL alone is rarely enough.

Invalidate on writes: the simplest rule that works

For caches tied to mutable data, invalidate when you write.

- After a successful update, delete affected keys.
- Prefer deleting by **pattern** or by maintaining a **key index**.

Example (delete on update):

- Update `user.email`.
- Immediately delete `user:{id}:profile` and `user:{id}:permissions` if permissions depend on email.

This keeps the cache aligned with the source of truth.

Invalidate by dependency: avoid "random deletes"

When one change affects multiple cached values, define dependencies.

- Track which cache entries depend on which data.

- Invalidate all dependents when the source changes.

Example dependency mapping:

- `policy:{roleId}` depends on `policy_rules_version`.
- `menu:{roleId}` depends on both `policy_rules_version` and `feature_flags_version`.

If you only delete `policy:{roleId}` but not `menu:{roleId}`, you'll serve stale menus until TTL expires.

Handle partial invalidation carefully

Sometimes you can invalidate only part of a cached response.

- Cache at the right granularity: per item, per page, per query signature.
- Avoid caching huge blobs when only one field changes.

Example:

- Instead of caching `GET /orders` as one big object, cache `order:{id}` and assemble the list.
- When one order changes, invalidate only that order entry.

This reduces invalidation blast radius and improves hit rate.

Prevent stampedes: invalidation can cause load spikes

When entries expire or are deleted, many requests may regenerate the same value.

- Use request coalescing (single-flight) or a short "lock" per key.
- Serve stale while revalidating if correctness allows it.

Example (single-flight concept):

- First request after invalidation computes and stores.
- Other requests wait briefly or reuse the in-progress result.

Even with correct invalidation, stampedes can hurt performance.

Concrete tuning workflow

Use this sequence when adjusting caching behavior:

1. **Identify the cached value and its source of truth.**
2. **List invalidation triggers** (write events, version changes, TTL expiry).
3. **Choose the invalidation mechanism** (delete keys, bump version, dependency invalidation).
4. **Define the cache key strategy** (include version, include query params, normalize inputs).
5. **Add instrumentation**: hit rate, stale rate, regeneration count, and error rate.
6. **Test invalidation** with deterministic scenarios.

Example: caching a computed response with explicit invalidation

Suppose you cache `GET /recommendations?userId=...`.

- Recommendations depend on `user_profile` and `behavior_events`.
- You have a `profile_version` and `events_version`.

Mind map (invalidation rules):

Click here to view the mind map: Recommendations cache

Example cache key:

- `reco:{userId}:pv{profileVersion}:ev{eventsVersion}`

Invalidation behavior:

- When profile updates, `profile_version` changes, so the old key is never reused.
- When events arrive, `events_version` changes, so recommendations refresh.

TTL becomes a safety net for cases where version bumps are delayed or missed.

Example: TTL-only cache with explicit “stale is acceptable” rule

Some data can be stale without breaking correctness.

- Example: “UI badge counts” that update frequently.

Mind map (TTL-only with guardrails):

Click here to view the mind map: Badge counts cache

The key is writing the correctness rule in plain language. If stale is not acceptable, TTL-only caching is the wrong tool.

Testing invalidation rules (make them executable)

Write tests that prove invalidation works, not just that caching works.

- **Version bump test:** compute once, bump version, compute again, assert value changes.
- **Write invalidation test:** update underlying data, assert cached value is deleted or bypassed.
- **Dependency test:** change a dependency, assert all dependent caches refresh.

Example test scenario:

- Cache `reco` for `userId=7`.
- Update `user_profile` for user 7.
- Verify the next request uses a new key (or recomputes) and returns updated recommendations.

Common pitfalls to avoid

- **Caching with incomplete keys:** forgetting query params or headers that affect output.
- **Invalidating the wrong layer:** deleting a list cache but not the item caches it contains.
- **Relying on TTL for correctness:** TTL should not be the only invalidation mechanism for security-sensitive data.
- **Unbounded key growth:** versioning without cleanup can accumulate old keys; set retention policies.

Quick reference: pick the invalidation rule that matches the data

- **You have a change signal:** use versioning or delete-on-write.
- **You can’t detect changes reliably:** use TTL, but only when stale is acceptable.
- **Multiple dependencies:** use dependency invalidation or version composition.
- **High concurrency:** add single-flight or revalidation to prevent stampedes.

Clear invalidation rules turn caching from a guessing game into a predictable system. Once you can state “when an entry is wrong,” tuning becomes straightforward: adjust keys, triggers, and regeneration behavior until the system stays correct under load.

8. Database and Query Optimization With AI

8.1 Writing Efficient Queries With Index Awareness

Efficient queries start with a simple question: “Which index can this database use to avoid scanning lots of rows?” Index awareness means you design your query so the database can match your conditions to an index’s structure, order, and data type.

The index basics that actually matter

An index is not magic; it’s a sorted lookup structure. Most relational databases use B-tree indexes for common cases. That implies a few practical rules:

- **Equality filters are friendly.** `WHERE status = 'PAID'` is usually easy to match to an index.
- **Range filters are partially friendly.** `WHERE created_at >= ? AND created_at < ?` can use an index, but only up to the point where ordering matters.
- **Leading wildcards break usefulness.** `LIKE '%abc'` typically can’t use a normal B-tree index.

- **Functions on indexed columns can block index use.** `WHERE DATE(created_at) = '2026-03-01'` often prevents using an index on `created_at`.
- **Column order in composite indexes matters.** For an index on `(a, b)`, conditions on `a` help most; conditions on `b` help best when `a` is constrained.

Mind map: index awareness checklist

Index Awareness Mind Map

Click here to view the mind map: Index Awareness

Concrete example: make a filter indexable

Assume a table:

- `orders(id, customer_id, status, created_at, total_cents)`

You have an index on:

- `CREATE INDEX idx_orders_customer_status_created ON orders(customer_id, status, created_at);`

Inefficient query (often forces extra work)

```
SELECT id, total_cents
FROM orders
WHERE customer_id = 42
  AND DATE(created_at) = '2026-03-01'
  AND status = 'PAID';
```

Reasoning: `DATE(created_at)` applies a function to the indexed column `created_at`. Many databases can't use the index efficiently because the predicate no longer matches the raw indexed values.

Efficient query (range on the raw column)

```
SELECT id, total_cents
FROM orders
WHERE customer_id = 42
  AND status = 'PAID'
  AND created_at >= '2026-03-01'
  AND created_at <  '2026-03-02';
```

Reasoning: this expresses a range directly on `created_at`. With the composite index, the database can use `customer_id` and `status` equality constraints, then narrow by the `created_at` range.

Composite index order: write conditions in the index's language

Suppose you also have:

- `CREATE INDEX idx_orders_status_created ON orders(status, created_at);`

Good match

```
SELECT id
FROM orders
WHERE status = 'PAID'
  AND created_at >= '2026-03-01'
  AND created_at <  '2026-03-02';
```

Reasoning: `status` is the first index column, and `created_at` is the second. Equality on the first column plus a range on the second is a common sweet spot.

Weaker match

```
SELECT id
FROM orders
WHERE created_at >= '2026-03-01'
  AND created_at <  '2026-03-02'
  AND status = 'PAID';
```

Reasoning: the database can still use the index, but depending on the optimizer, it may not be as direct. The query text order doesn't usually matter, but the *effective constraints* do. If the optimizer treats the range as the primary filter, it may reduce how much it can use the `status` portion.

Pattern matching: prefer prefix searches

Assume:

- `users(id, email)`
- index: `CREATE INDEX idx_users_email ON users(email);`

Often index-friendly

```
SELECT id
FROM users
WHERE email LIKE 'alex.%';
```

Reasoning: the pattern has a fixed prefix, so the database can search the index range for values starting with `alex.`.

Usually not index-friendly

```
SELECT id
FROM users
WHERE email LIKE '%@example.com';
```

Reasoning: the leading wildcard means the database can't narrow by a starting point in the sorted index. It typically falls back to scanning.

Avoid implicit casts and type mismatches

Assume `orders.status` is a text/varchar column, but your application sometimes sends numeric codes.

Risky query

```
SELECT id
FROM orders
WHERE status = 1;
```

Reasoning: the database may cast `status` or the literal, which can prevent index usage. Even if it still returns correct results, it may do extra scanning.

Safer query

```
SELECT id
FROM orders
WHERE status = '1';
```

Reasoning: matching the literal type to the column type keeps the predicate aligned with the index.

NULLs: write them explicitly

Assume `customers(last_login_at)` is indexed.

Correct and index-aware

```
SELECT id
FROM customers
WHERE last_login_at IS NULL;
```

Reasoning: `IS NULL` is explicit. Using `= NULL` returns no rows in SQL, and using expressions that hide NULL checks can complicate index usage.

Join efficiency: index the join keys

Assume:

- `orders(customer_id, ...)`
- `customers(id, ...)`

If you join orders to customers:

```
SELECT o.id, c.email
FROM orders o
JOIN customers c ON c.id = o.customer_id
WHERE o.status = 'PAID';
```

Reasoning: you want indexes on `orders.customer_id` and `customers.id` (the latter is often the primary key). Without them, the database may choose a plan that scans one side and probes the other inefficiently.

Verify with EXPLAIN and interpret what you see

A query can look "correct" but still scan. Use `EXPLAIN` (or `EXPLAIN ANALYZE`) to confirm:

- whether an **Index Scan** / **Index Seek** is used
- whether the plan shows **rows removed by filter** is small
- whether the index chosen matches your intended columns

Mind the difference between "an index exists" and "the optimizer used it for this query."

Quick reference: patterns to rewrite

> If you see any of these in a query, consider rewriting them to preserve index usage.

- `WHERE DATE(col) = ...` → `WHERE col >= ... AND col < ...`
- `WHERE col LIKE '%x'` → redesign search or use a different index type
- `WHERE col = CAST(? AS type)` → ensure the parameter type matches the column
- `WHERE func(col) = ...` → move the function to the other side when possible

A practical mini-template

When you have a composite index like `(a, b, c)`, aim for:

- `a = ?`
- `b = ?`
- `c` as either equality or a range

Example:

```sql
SELECT id
FROM orders
WHERE customer_id = 42
  AND status = 'PAID'
  AND created_at >= '2026-03-01'
  AND created_at <  '2026-03-02';
```

This query aligns with the index structure and gives the optimizer clear boundaries to work with.

8.2 Diagnosing Slow Queries Using Explain Plans

When a query is slow, the database is usually telling you why—just in a language that looks like it was designed by accountants. An `EXPLAIN` (or `EXPLAIN ANALYZE`) plan shows how the database intends to execute the query: which tables it reads, which indexes it uses, how it joins rows, and where it spends time. The goal is not to memorize plan formats; it's to connect plan choices to concrete query behavior.

What to run first: `EXPLAIN` vs `EXPLAIN ANALYZE`

- `EXPLAIN` shows the planned strategy without executing. It's useful for quick checks and for catching obvious issues like missing indexes.
- `EXPLAIN ANALYZE` executes the query (or the relevant parts) and reports actual timing and row counts. This is the best option when you can afford the run and want to confirm whether the plan matches reality.

If your query is expensive, start with `EXPLAIN` and then move to `EXPLAIN ANALYZE` after you've narrowed the likely culprit.

Reading an explain plan: the five signals that matter

1. **Access path**: Are you seeing `Index Scan`, `Index Seek`, or `Seq Scan`? A sequential scan isn't always wrong, but it often explains "why it's slow" when the table is large.
2. **Estimated vs actual rows**: Big gaps suggest the optimizer's assumptions are off (stale statistics, skewed data, or predicates that don't match indexes).
3. **Join strategy**: Look for `Nested Loop`, `Hash Join`, or `Merge Join`. The "right" join depends on row counts and available indexes.
4. **Sort and aggregation steps**: `Sort`, `GroupAggregate`, `HashAggregate`, or `Distinct` can dominate runtime if they process too many rows.
5. **Filter placement**: Filters applied early reduce work. Filters applied late can force the engine to process far more rows than necessary.

Mind map: explain plan diagnosis workflow

Click here to view the mind map: Diagnose slow query with EXPLAIN

Example 1: Missing index shows up as a sequential scan

Query (PostgreSQL-style):

```sql
SELECT *
FROM orders
WHERE customer_id = 42
  AND created_at >= '2024-01-01';
```

Plan symptoms (conceptually):

- `Seq Scan on orders`
- Filter on `customer_id` and `created_at`
- Large number of rows removed by filter

Reasoning: If the plan reads most of the table and then filters down, the engine is doing a lot of unnecessary work. The fix is usually to create an index that matches the predicate.

Targeted index:

- If you filter by `customer_id` and a date range, a composite index like `(customer_id, created_at)` often helps.

After fix, you want to see:

- `Index Scan` (or `Index Range Scan`) on `orders`
- Far fewer rows examined before applying the range condition

Example 2: Estimated vs actual rows reveals a statistics or predicate mismatch

Query:

```
SELECT status, COUNT(*)
FROM events
WHERE occurred_at >= NOW() - INTERVAL '7 days'
GROUP BY status;
```

Plan symptoms:

- Estimated rows are small, but actual rows are huge
- `HashAggregate` or `GroupAggregate` processes far more rows than expected

Reasoning: The optimizer may believe the time window is selective when it isn't, or it may not understand the distribution of `occurred_at`. Another common cause is a predicate that doesn't match the index due to a function or cast.

Practical checks:

- Confirm whether there's an index on `occurred_at`.
- Ensure the predicate is sargable (search-argument-able). For example, avoid wrapping the column in a function like `DATE(occurred_at) = ...`.

What to look for after rewriting:

- The scan operator should reduce rows earlier.
- The aggregation operator should receive fewer input rows.

Example 3: Join strategy changes when indexes exist on join keys

Query:

```
SELECT u.id, o.id
FROM users u
JOIN orders o ON o.user_id = u.id
WHERE u.country = 'US';
```

Plan symptoms:

- `Nested Loop` join
- Many iterations of the inner side
- Inner side shows `Seq Scan` or an inefficient index usage

Reasoning: A nested loop join can be fine when the inner lookup is cheap. If the inner lookup scans many rows each time, runtime explodes.

Fix:

- Ensure `orders.user_id` is indexed.
- If the query filters `users` by `country`, consider whether an index on `users(country, id)` (or at least `country`) helps the outer side reduce rows.

After fix, you want:

- Outer side returns fewer `users` rows.
- Inner side uses an index lookup on `orders.user_id`.
- The plan may still use nested loops, but the inner work per iteration should be tiny.

Example 4: ORDER BY + LIMIT without a supporting index

Query:

```sql
SELECT *
FROM logs
WHERE service = 'auth'
ORDER BY created_at DESC
LIMIT 50;
```

Plan symptoms:

- `Sort` on many rows before applying `LIMIT`
- Sort method and memory usage indicate heavy work

Reasoning: Without an index that matches both the filter and the ordering, the database may sort a large candidate set just to return the first 50 rows.

Fix:

- Create an index that supports the filter and the order, such as `(service, created_at DESC)`.

After fix, you want:

- An index scan that produces rows already in the desired order.
- Minimal or no explicit sort step.

Example 5: DISTINCT and GROUP BY can hide expensive row expansion

Query:

```sql
SELECT DISTINCT user_id
FROM sessions
WHERE started_at >= '2024-02-01';
```

Plan symptoms:

- Large scan output
- `HashAggregate` or `Sort` used to implement `DISTINCT`

Reasoning: `DISTINCT` requires deduplication. If the plan deduplicates after reading many rows, it can be costly.

Fix options:

- If there's an index on `(started_at, user_id)` or `(user_id, started_at)`, the engine may reduce work by scanning in an order that makes deduplication cheaper.
- If the business logic allows it, reduce the candidate set earlier with more selective predicates.

After fix, check that the deduplication operator receives fewer rows.

A quick checklist to apply to any plan

- Does the plan use an index for the most selective predicate?
- Are there operators that process "too many" rows (scan, join inner side, sort, aggregate)?
- Are estimated row counts wildly different from actual row counts?
- Are join keys indexed on the side that's being probed?
- Does `ORDER BY ... LIMIT` avoid sorting large sets?

Interpreting one plan line: a small but useful habit

When you spot the slowest operator, note three things: **input rows**, **output rows**, and **time** (from `EXPLAIN ANALYZE`). Then ask what in the query shape could reduce the input rows to that operator. Most fixes—indexes, predicate rewrites, or query restructuring—work by shrinking the input, not by "making the operator faster" in isolation.

Putting it together: a concrete workflow

1. Run `EXPLAIN ANALYZE` for the slow query.

2. Identify the operator with the highest time or the biggest row explosion.
3. Check whether the scan is sequential when you expected an index.
4. Verify join key indexes and whether join strategy matches row counts.
5. Re-run `EXPLAIN ANALYZE` after each change and confirm that the operator's input rows dropped.

That last step matters: a plan that "looks better" but doesn't reduce rows at the bottleneck operator usually means the change didn't address the real cause.

8.3 Refactoring ORM Queries to Reduce N Plus One Problems

N plus one problems happen when an ORM fetches a list in one query, then performs one additional query per item in that list. The result is usually correct data, but slow performance and noisy database logs. The fix is to fetch related data in a single, well-shaped query (or a small, predictable set of queries).

What the problem looks like

Consider an app that shows orders and the customer name for each order.

Bad pattern (typical ORM lazy loading):

- Query all orders.
- For each order, access `order.customer`.
- Each access triggers a separate query.

Example (Django ORM):

```
# BAD: one query for orders, then one query per order's customer
orders = Order.objects.all()
for order in orders:
    print(order.customer.name)
```

If there are 50 orders, you'll likely see 51 queries: 1 for orders plus 50 for customers. The code is simple, which is why it slips into production.

Mind map: where N+1 comes from and how to stop it

N+1 Refactoring Mind Map

Click here to view the mind map: N+1 Refactoring

The core refactor: eager loading

Eager loading tells the ORM to fetch related objects up front. Most ORMs provide two common strategies:

- **Join-based eager loading**: one query with joins. Great when you need a small number of related rows.
- **Prefetch-based eager loading**: one query for parents plus one query for related rows, then the ORM stitches results in memory. Great when relationships are one-to-many.

Django example: `select_related` for many-to-one

If each order has exactly one customer (many-to-one), use `select_related`.

```
# GOOD: orders + customer fetched together via JOIN
orders = Order.objects.select_related('customer').all()
for order in orders:
    print(order.customer.name)
```

Now the ORM can fetch customer data in the same query as orders. Query count stays constant as the number of orders grows.

Django example: `prefetch_related` for one-to-many

If an order has many line items, and each line item has a product, you typically want `prefetch_related`.

```
# GOOD: stable query count for one-to-many relationships
orders = (
    Order.objects
    .prefetch_related('line_items', 'line_items__product')
    .all()
)
for order in orders:
    total = sum(li.quantity * li.product.price for li in order.line_items.all())
    print(total)
```

This avoids one query per line item and one query per product. The ORM performs a predictable number of queries, then assembles the object graph.

Refactor by shaping the data you actually need

Eager loading fixes query count, but you can still waste time by fetching too much.

Select only required fields

If you only need order id and customer name, consider projecting fields instead of loading full objects.

Example (Django):

```
# GOOD: fewer columns, less object construction
rows = (
    Order.objects
    .select_related('customer')
    .values('id', 'customer__name')
)
for row in rows:
    print(row['id'], row['customer__name'])
```

This can reduce memory pressure and speed up serialization.

Refactor by moving work out of loops

A common pattern is to compute something inside a loop that triggers relationship access. Even with eager loading, you might still do extra work.

Example: computing totals

Suppose you compute order totals by iterating line items and multiplying by product price. If product price is already prefetched, the loop is fine. If not, it becomes N+1.

A cleaner approach is to compute totals in the database when possible.

Example (Django ORM with aggregation):

```
# GOOD: compute totals in SQL
from django.db.models import Sum, F

orders = (
    Order.objects
    .annotate(total=Sum(F('line_items__quantity') * F('line_items__product__price')))
)
for order in orders:
    print(order.id, order.total)
```

This reduces both query count and Python-side iteration. Use it when the calculation maps well to SQL.

Mind map: choosing between join and prefetch

Click here to view the mind map: Join vs Prefetch

How to verify the fix

Refactoring is only complete when you can prove the query count and behavior.

Add a query-count assertion in tests

In Django, you can assert the number of queries executed during a view or service call.

Example (conceptual):

```
# Pseudocode-style test idea
# Assert that rendering orders executes a stable number of queries
# e.g., 2 queries: orders + related data
```

Even if you don't lock the exact number, you can still ensure it doesn't scale with the number of orders.

Inspect generated SQL

When results are wrong after refactoring, the issue is usually one of these:

- You used `select_related` for a one-to-many relationship.
- You forgot a nested relation like `line_items__product`.
- You filtered on related fields but changed the join behavior.

Inspecting the ORM's SQL helps pinpoint which relationship fetch strategy is actually being used.

Common pitfalls (and how to avoid them)

1. **Eager loading the wrong path**: If your code accesses `order.line_items.all()` but you prefetched `line_items`, it's fine; if you accessed `order.line_items` without `.all()` in a different way, it might still work, but be consistent.
2. **Accidentally reintroducing lazy loads**: Serializers or template code may access relationships you didn't eager load. If you see query count rise again, check the access points.
3. **Row explosion from joins**: Joining across multiple one-to-many relationships can multiply rows. When that happens, switch to `prefetch_related`.

A practical refactoring checklist

- Identify which relationship attributes are accessed inside loops or during serialization.
- Measure current query count with a representative dataset.
- Replace lazy loading with `select_related` (many-to-one/one-to-one) or `prefetch_related` (one-to-many/many-to-many).
- If you only need a few fields, use projection (`values`) to reduce payload.
- For computed aggregates, consider database-side `annotate` / `aggregate`.
- Add a test that ensures query count stays stable as the number of parent rows increases.

When you apply these steps, the code stays readable, the database does predictable work, and performance stops depending on how many rows you happen to have today.

8.4 Optimizing Pagination and Filtering Patterns

Pagination is where "works on my machine" meets "why is this slow?" The goal is to return the right slice of data with predictable performance, stable ordering, and filters that don't accidentally force the database to scan everything.

1) Start with a stable sort order

Offset-based pagination (using `LIMIT` / `OFFSET`) assumes the result order doesn't change between requests. If rows can be inserted or updated, users may see duplicates or missing items.

Best practice: choose a deterministic sort key.

- For typical feeds: `ORDER BY created_at DESC, id DESC`.
- For user lists: `ORDER BY last_name ASC, id ASC`.

Example (SQL):

```
SELECT id, created_at, title
FROM posts
WHERE status = 'published'
ORDER BY created_at DESC, id DESC
LIMIT 20 OFFSET 40;
```

If you can't guarantee stability, prefer keyset pagination (next section).

2) Use keyset pagination for large or frequently changing datasets

Keyset pagination (a.k.a. "seek method") avoids `OFFSET`, which grows expensive because the database must skip more rows.

Concept: instead of "page 3," ask "items after the last seen key."

Example (SQL, keyset):

```
SELECT id, created_at, title
FROM posts
WHERE status = 'published'
  AND (created_at, id) < (:last_created_at, :last_id)
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

This pattern stays fast because the database can use an index on `(created_at, id)`.

Practical tip: return the "cursor" values (`last_created_at`, `last_id`) with the response so the client can request the next slice.

3) Pick the right pagination style for the job

- **Offset pagination** is fine for small datasets, admin screens, or when users jump to arbitrary pages.
- **Keyset pagination** is better for timelines, search results, and anything where users mostly move forward.

A common compromise: use offset for "page navigation" UI, but switch to keyset for infinite scroll.

4) Make filtering index-friendly

Filtering is where performance wins or losses happen. The database can only use indexes effectively when the query shape matches available indexes.

Avoid wrapping indexed columns in functions

If you do `WHERE DATE(created_at) = '2026-03-01'`, the index on `created_at` may not help.

Better: use a range.

```
WHERE created_at >= '2026-03-01'
  AND created_at <  '2026-03-02'
```

Prefer exact matches and ranges over "contains" when possible

- `status = 'published'` is index-friendly.
- `title LIKE '%foo%'` usually isn't (unless you have specialized indexing).

If you must support partial text matching, keep it separate from the main pagination query so the database doesn't combine expensive text work with every page.

Keep filter logic consistent across pages

If the filter changes between requests (for example, “only show items created after now”), pagination becomes inconsistent. Use a fixed snapshot boundary when correctness matters.

5) Combine filters with pagination without breaking ordering

When you add filters, ensure the `ORDER BY` still matches the pagination method.

Offset pagination with stable order:

```
SELECT id, created_at
FROM posts
WHERE status = 'published'
  AND author_id = :author_id
ORDER BY created_at DESC, id DESC
LIMIT 20 OFFSET :offset;
```

Keyset pagination with the same order:

```
SELECT id, created_at
FROM posts
WHERE status = 'published'
  AND author_id = :author_id
  AND (created_at, id) < (:last_created_at, :last_id)
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

If you change the `ORDER BY` fields, the cursor no longer represents a correct “next page.”

6) Mind the “N+1” effect in paginated endpoints

Pagination often hides a second problem: fetching related data per row.

Example (bad pattern):

- Query page of posts.
- For each post, query comments.

This can multiply database calls by page size. Instead, fetch what you need in one query (joins or batched queries) and keep the result bounded.

7) Mind maps: pagination and filtering patterns

Mind map: Pagination choices

Click here to view the mind map: Pagination

Mind map: Filtering that stays fast

Click here to view the mind map: Filtering

8) A concrete workflow for designing a paginated endpoint

1. **Write the exact ordering** you want users to see.
2. **Decide pagination style** based on whether users need random page jumps.
3. **Add filters** and ensure they don’t require scanning the whole table.
4. **Align cursor fields** with the `ORDER BY` keys.
5. **Check indexes**: the database should be able to use them for both filtering and ordering.
6. **Validate correctness** with a small dataset where inserts happen between requests.

9) Example: timeline endpoint with filters and keyset pagination

Assume a feed of events with optional filters.

Requirements:

- Show newest first.
- Filter by `type` and `user_id`.
- Paginate with a cursor.

SQL pattern:

```
SELECT id, created_at, type, payload
FROM events
WHERE user_id = :user_id
  AND type = :type
  AND (created_at, id) < (:last_created_at, :last_id)
ORDER BY created_at DESC, id DESC
LIMIT 25;
```

Why it works:

- The tuple comparison matches the ordering.
- The filters are simple and indexable.
- The query cost stays stable as the feed grows.

10) Common mistakes to avoid

- **Cursor not matching ordering:** cursor fields must correspond exactly to `ORDER BY`.
- **Changing filters mid-session:** keep filter parameters fixed for a pagination sequence.
- **Using OFFSET for deep pages:** it often becomes the dominant cost.
- **Unbounded joins:** if you join to large child tables, you can inflate rows and break pagination expectations.

When pagination and filtering are designed together—ordering first, then cursor or offset, then index-friendly predicates—your endpoint becomes predictable. Users get consistent slices, and the database gets a query plan it can actually live with.

8.5 Validating Changes With Regression Tests

Regression testing is how you prove that a change fixed the intended problem without breaking the rest of the system. When you use an AI coding assistant, this step matters even more: the assistant can produce plausible code that compiles but still changes behavior in subtle ways. The goal here is not “more tests,” but the right tests that catch the specific ways things can go wrong.

What “validated” means in practice

A change is validated when you can answer three questions:

1. **Did the bug stay fixed?** Run the test that reproduces the bug and confirm it passes.
2. **Did behavior stay consistent elsewhere?** Run a focused suite that covers the surrounding logic and boundaries.
3. **Did performance-critical paths stay acceptable?** If the change touches hot code, run at least one benchmark-style test or a time-based assertion.

A good regression plan is small enough to run often and specific enough to fail when something breaks.

Mind map: regression test validation workflow

Click here to view the mind map: Regression validation (8.5)

Step 1: Start from the smallest failing example

If you already have a failing case, turn it into a regression test immediately. If you don’t, create one by capturing the minimal input that triggers the problem.

Example (Python, bug in date parsing):

Suppose production fails when the input is `"2024-02-29"` (leap day). The fix changes parsing logic.

```
# test_date_parsing.py
import pytest
from myapp.dates import parse_date

def test_parse_date_leap_day():
    assert parse_date("2024-02-29").isoformat() == "2024-02-29"
```

This test is intentionally narrow: it checks the exact behavior that was broken.

Step 2: Add invariants around the fix

A regression test should assert more than "it doesn't crash." It should encode an invariant: something that must always be true.

For date parsing, invariants might include:

- Valid dates parse to the same calendar date.
- Invalid formats raise a specific exception.
- Time zone handling (if any) is consistent.

Example (same module, add error-path coverage):

```
import pytest
from myapp.dates import parse_date

def test_parse_date_rejects_invalid_format():
    with pytest.raises(ValueError):
        parse_date("02/29/2024")
```

This catches a common regression: the fix accidentally broadens accepted formats.

Step 3: Use "change-aware" test selection

Not every test needs to run for every change. Instead, select tests based on what the change could affect.

A practical approach:

- If you changed a **pure function**, unit tests are usually enough.
- If you changed **data flow** (mapping, serialization, validation), add integration tests that cross the boundary.
- If you changed **external behavior** (HTTP status codes, error messages, database queries), add contract-style tests.

Example (JavaScript/TypeScript, refactor in request validation):

If you changed validation rules, you want tests that check HTTP responses, not just the validator function.

```
// validation.test.ts
import request from "supertest";
import { app } from "./app";

test("rejects missing email with 400", async () => {
  const res = await request(app).post("/signup").send({ email: "" });
  expect(res.status).toBe(400);
  expect(res.body.error).toMatch(/email/i);
});
```

This ensures the refactor didn't change the user-visible contract.

Step 4: Make tests deterministic (or they'll waste your time)

Regression tests should fail for real reasons, not timing or randomness.

Common sources of flakiness:

- Tests that depend on the current time.
- Tests that rely on unordered collections.
- Concurrency tests without synchronization.

Example (freeze time):

```
from datetime import datetime, timezone
import pytest
from myapp.dates import parse_date

def test_parse_date_with_fixed_now(monkeypatch):
    monkeypatch.setattr(
        "myapp.dates.utcnow",
        lambda: datetime(2024, 2, 29, tzinfo=timezone.utc)
    )
    assert parse_date("today").isoformat() == "2024-02-29"
```

Even if the bug was about parsing, freezing time prevents unrelated changes from causing intermittent failures.

Step 5: Run the right suite, then interpret failures correctly

A regression run should produce actionable signals.

Recommended workflow:

1. Run the **targeted tests** for the changed module.
2. If they pass, run a **broader smoke suite** (e.g., all tests for the service).
3. If performance is affected, run a **benchmark-style check** or a time-limited test.

When tests fail, triage quickly:

- If the failure is in the test itself (wrong expectation), fix the test only after verifying the intended behavior.
- If the failure is in the code, keep the test and adjust the implementation.

Example (interpreting a mismatch): If a test expects `400` but receives `422`, the regression might be in error mapping, not validation logic.

Step 6: Add a performance guardrail when optimization is involved

If the change is an optimization, correctness tests alone don't guarantee you didn't accidentally slow down a hot path or introduce extra work.

A simple guardrail is a benchmark-style test that asserts an upper bound under controlled conditions.

Example (Python, coarse timing check):

```
import time
from myapp.search import find_matches

def test_find_matches_fast_enough():
    data = ["alpha"] * 10000
    start = time.perf_counter()
    find_matches(data, "alpha")
    elapsed = time.perf_counter() - start
    assert elapsed < 0.05
```

This is not a perfect performance measurement, but it catches obvious regressions introduced by the change.

Step 7: Use AI assistants to generate tests, then verify intent

AI can help draft tests, but you should validate the test's meaning:

- Does it reproduce the original bug?
- Does it assert the invariant you care about?
- Are inputs minimal and deterministic?

Example (prompt pattern for regression test generation):

Ask for:

- A minimal failing test first.
- Then a second test for the adjacent boundary.
- Finally, a test for the error path.

When the assistant proposes tests, review them like you would any code: check assumptions, edge cases, and expected outputs.

Step 8: Lock regression tests into the workflow

Validation is only real if it runs consistently.

Practical checklist:

- Tests are committed with the change.
- CI runs the regression suite on pull requests.
- Flaky tests are fixed immediately (or removed) rather than tolerated.

A regression test that never runs is just documentation with extra steps.

Quick example: end-to-end regression for a bug fix

1. Write a minimal test that fails on the bug.
2. Implement the fix.
3. Add one invariant test (boundary or error path).
4. Run targeted tests for the module.
5. Run the service smoke suite.
6. If performance was touched, add a guardrail test.

That sequence keeps the feedback loop tight and makes it clear whether the change is truly safe.

9. Secure Coding With AI Assistants

9.1 Prompting for Secure Defaults and Input Validation

Secure defaults are what you get when you assume the user is wrong until proven otherwise. Input validation is how you prove it. When you ask an AI coding assistant to help, you want it to generate code that (1) rejects unsafe input early, (2) uses safe parsing and escaping, and (3) fails in predictable ways.

What to ask for (and why it matters)

1. **State the trust boundary.** Tell the assistant which fields come from users (query params, JSON body, headers, form fields). This prevents "helpful" code that validates the wrong thing.
2. **Require explicit validation rules.** Ask for length limits, allowed character sets, numeric ranges, and required/optional fields. Without these, the assistant may only add superficial checks.
3. **Demand safe parsing and error handling.** Request that parsing errors return a consistent client error (e.g., HTTP 400) without leaking stack traces or internal details.
4. **Ask for secure defaults in behavior.** Examples: default to deny on unknown enum values, cap pagination sizes, treat missing required fields as invalid, and avoid "best effort" parsing.
5. **Require tests for validation.** Ask for a small set of tests that cover both valid and invalid inputs. This catches the common failure mode: validation that looks right but doesn't actually run.

Mind map: secure defaults + validation

Click here to view the mind map: Secure Defaults & Input Validation

Example prompt: validating a JSON request

Use a prompt that names the rules and the expected response behavior.

Prompt you can paste:

> You are implementing request validation for `POST /api/users`. The request body is JSON from an untrusted client. Validate these fields:
>
> - `email`: required, max 254 chars, must match a basic email pattern, reject whitespace.
> - `age`: optional, if present must be an integer between 0 and 120.
> - `role`: optional, if present must be one of `"user"`, `"admin"` (deny unknown values).
> - `bio`: optional, max 500 chars. If validation fails, return HTTP 400 with JSON: `{ "error": "invalid_request", "details": [ ... ] }`. Do not throw uncaught exceptions. Do not log request bodies. Provide unit tests for at least 6 invalid cases.

Example implementation (TypeScript/Node)

```ts
type Role = "user" | "admin";

function isEmail(s: string) {
  if (s.length > 254) return false;
  if (/\s/.test(s)) return false;
  return /^[^@\s]+@[^@\s]+\.[^@\s]+$/.test(s);
}

function validateCreateUser(body: any) {
  const details: string[] = [];

  if (!body || typeof body !== "object") details.push("body must be an object");

  if (typeof body?.email !== "string" || !isEmail(body.email)) {
    details.push("email must be a valid email without whitespace");
  }

  if (body?.age !== undefined) {
    const age = body.age;
    if (!Number.isInteger(age) || age < 0 || age > 120) details.push("age must be an integer 0..120");
  }

  if (body?.role !== undefined) {
    const role = body.role;
    if (role !== "user" && role !== "admin") details.push("role must be user or admin");
  }

  if (body?.bio !== undefined) {
    if (typeof body.bio !== "string" || body.bio.length > 500) details.push("bio must be a string up to 500 chars");
  }

  return { ok: details.length === 0, details };
}
```

Example usage in an endpoint

```ts
app.post("/api/users", (req, res) => {
  const { ok, details } = validateCreateUser(req.body);
  if (!ok) {
    return res.status(400).json({ error: "invalid_request", details });
  }
  // proceed with safe, validated values
  res.status(201).json({ status: "created" });
});
```

Example prompt: secure defaults for query parameters

Query parameters are where “helpful” coercion goes to die. Ask for strict parsing and caps.

Prompt you can paste:

Implement validation for `GET /api/items` query params:

- `page`: optional, integer 1..1000, default 1.
- `limit`: optional, integer 1..100, default 20.
- `sort`: optional, allowlist `"name"`, `"createdAt"`, default `"createdAt"`.
- `q`: optional, string max 100 chars, reject control characters. If invalid, return HTTP 400 with `{ "error": "invalid_query", "details": [...] }`. Do not accept floats for integers. Do not accept empty strings for `q`.

Example implementation (Python)

```
import re

def validate_items_query(params):
    details = []

    def parse_int(name, default=None, lo=None, hi=None):
        if name not in params or params[name] is None:
            return default
        raw = params[name]
        if isinstance(raw, bool) or not isinstance(raw, str) or not raw.isdigit():
            details.append(f"{name} must be an integer")
            return None
        val = int(raw)
        if lo is not None and val < lo: details.append(f"{name} must be >= {lo}")
        if hi is not None and val > hi: details.append(f"{name} must be <= {hi}")
        return val

    page = parse_int("page", default=1, lo=1, hi=1000)
    limit = parse_int("limit", default=20, lo=1, hi=100)

    sort = params.get("sort")
    if sort is None:
        sort = "createdAt"
    if sort not in ("name", "createdAt"):
        details.append("sort must be name or createdAt")

    q = params.get("q")
    if q is not None:
        if not isinstance(q, str) or q == "":
            details.append("q must be a non-empty string")
        elif len(q) > 100:
            details.append("q must be at most 100 chars")
        elif re.search(r"[\x00-\x1F\x7F]", q):
            details.append("q must not contain control characters")

    ok = len(details) == 0
    return ok, {"page": page, "limit": limit, "sort": sort, "q": q}, details
```

Mind map: common secure-default traps

Click here to view the mind map: Secure defaults: common traps

Example tests prompt: validation edge cases

When you ask for tests, specify the failure shape and the edge cases.

Prompt you can paste:

Write unit tests for `validateCreateUser`. Include invalid cases for:

1. email with whitespace
2. email too long
3. age as a float (e.g., 12.5)
4. age negative
5. role set to an unknown string
6. bio longer than 500 chars Assert that `ok` is false and that `details` contains the expected message fragments.

Quick checklist to include in your prompts

- “Treat all request fields as untrusted.”
- “Use allowlists for enums; deny unknown values.”
- “Add max lengths and numeric bounds.”
- “Reject control characters and whitespace where relevant.”
- “Return HTTP 400 with a consistent error schema.”
- “Provide unit tests for invalid inputs and edge cases.”

Secure defaults aren’t a separate feature you bolt on later. They’re the default behavior you ask the assistant to implement from the first prompt, so the code starts out safe and stays that way when you iterate.

9.2 Preventing Injection Attacks With Parameterization Examples

Injection attacks happen when untrusted input is treated as code or structure instead of data. Parameterization fixes this by separating “what the user typed” from “what the database or interpreter should execute.” The result is that special characters keep their meaning as plain text, not as instructions.

The core idea (in plain terms)

When you build a query by concatenating strings, the database parses the final string as SQL. If the input contains SQL syntax, it becomes part of the command. With parameterized queries, the database parses the SQL once, then receives parameters as values. Those values are not re-parsed as SQL.

Mind map: where injection comes from and how parameters stop it

Injection Prevention Mind Map

Click here to view the mind map: Injection Prevention

SQL injection: a concrete example

Assume you want to fetch a user by email.

Unsafe approach (string concatenation):

- The query string is assembled with user input.
- A malicious email can inject extra SQL.

```
-- Unsafe: user input is merged into SQL text
SELECT id, email
FROM users
WHERE email = '" + userEmail + "';
```

If `userEmail` is `a' OR '1'='1`, the resulting SQL becomes logically true for many rows.

Safe approach (parameterization):

- The SQL structure is fixed.
- The email is passed as a parameter value.

```
-- Safe: SQL text is fixed, value is a parameter
SELECT id, email
FROM users
WHERE email = ?;
```

The database treats the parameter as a literal string, so `a' OR '1'='1` stays just a string.

Parameterization examples in common languages

Example: Node.js with PostgreSQL (parameterized)

```
// Unsafe pattern to avoid:
// const sql = `SELECT * FROM users WHERE email = '${email}'`;

const sql = 'SELECT id, email FROM users WHERE email = $1';
const result = await db.query(sql, [email]);
return result.rows;
```

Why this works: `$1` is a placeholder. The driver sends the SQL and the parameter separately.

Example: Python with psycopg (parameterized)

```
# Unsafe:
# cur.execute(f"SELECT id, email FROM users WHERE email = '{email}'")

cur.execute(
    "SELECT id, email FROM users WHERE email = %s",
    (email,)
)
rows = cur.fetchall()
```

Why this works: `%s` marks a value slot. The driver binds the value safely.

Example: Java with JDBC (parameterized)

```
String sql = "SELECT id, email FROM users WHERE email = ?";
PreparedStatement ps = conn.prepareStatement(sql);
ps.setString(1, email);
ResultSet rs = ps.executeQuery();
```

Why this works: `PreparedStatement` ensures the value is not interpreted as SQL.

Handling “dynamic” parts safely

A common mistake is to parameterize only the value, then still concatenate identifiers like column names or sort directions.

Safe: parameterize values

- Filter values: parameterize.
- Limits/offsets: parameterize if supported by your driver.

Not safe: parameterize SQL identifiers

Placeholders usually cannot replace table names, column names, or keywords. If you need dynamic identifiers, use a strict allowlist.

Mind map: dynamic SQL without foot-guns

Click here to view the mind map: Dynamic SQL Safely

Example: safe ORDER BY with an allowlist

Suppose you allow sorting by `created_at` or `email`.

```
const allowedFields = {
  created_at: 'created_at',
  email: 'email'
};

const field = allowedFields[sortField];
if (!field) throw new Error('Invalid sort field');

const sql = `SELECT id, email, created_at
             FROM users
             WHERE status = $1
             ORDER BY ${field} ASC
             LIMIT $2`;

const result = await db.query(sql, [status, limit]);
```

Reasoning: the only concatenated part is a value chosen from a fixed allowlist, not raw user input.

Command injection: parameterization by avoiding the shell

SQL has placeholders; command execution has a similar principle: don't let user input be interpreted as part of a command line.

Unsafe approach:

- Passing a single string to a shell.
- Special characters can change meaning.

Safer approach:

- Provide the executable and arguments separately.
- Avoid shell interpretation.

```
// Unsafe:
// exec(`grep ${pattern} ${file}`, ...)

// Safer:
spawn('grep', [pattern, file], { shell: false });
```

Why this works: the runtime treats `pattern` and `file` as arguments, not as shell syntax.

Testing parameterization with malicious inputs

You don't need a full security lab to get value from tests. Use a few targeted cases that would break unsafe code.

Example test cases for SQL filters:

- `email = "a' OR '1'='1"`
- `email = "' ; DROP TABLE users; --"`
- `email = "normal@example.com"`

Expected behavior:

- The query returns either zero rows or the correct matching row.
- It never errors due to SQL syntax.
- It never performs unintended actions.

Practical checklist for parameterization

- Use prepared statements or driver parameter binding for every user-controlled value.
- Keep query structure fixed; parameterize values, not SQL keywords.
- For dynamic identifiers (columns, sort fields), use allowlists and reject everything else.
- For command execution, pass arguments separately and avoid shell parsing.
- Add tests that include characters commonly used in injection payloads.

Summary

Parameterization prevents injection by ensuring the database (or command runner) parses the query structure separately from the user-provided values. When you combine placeholders for values with allowlists for identifiers, you close the most common gaps where “dynamic” code accidentally becomes executable instructions.

9.3 Handling Secrets and Configuration Safely

Secrets are values that grant access: API keys, database passwords, signing keys, and tokens. Configuration is everything else that changes behavior: feature flags, log levels, hostnames, and timeouts. Treating them differently keeps you from accidentally publishing credentials while still letting the app run in multiple environments.

The core rule: separate “where it lives” from “how it’s used”

Your code should never contain secrets in plain text. Instead, it should read them from the environment (or a secrets manager) at startup, validate them, and fail fast if required values are missing. This makes failures obvious and prevents the common “it worked on my machine” problem.

A practical mental model:

- **Build time:** compile code, not secrets.
- **Deploy time:** provide configuration and secrets.
- **Run time:** load, validate, and use.

Mind map: secrets vs configuration

Click here to view the mind map: Secrets and Configuration Safely

Environment variables: simple, effective, and easy to mess up

Environment variables are common because they’re straightforward and work across languages. The safety part is in the details: naming, validation, and logging.

Example (Node.js/TypeScript-style pseudocode):

```
const required = (name: string) => {
  const v = process.env[name];
  if (!v || v.trim() === "") {
    throw new Error(`Missing required env var: ${name}`);
  }
  return v;
};

const config = {
  dbUrl: required("DB_URL"),
  jwtSecret: required("JWT_SECRET"),
  logLevel: process.env.LOG_LEVEL ?? "info",
};
```

Notice what’s missing: no printing of `jwtSecret`, and no fallback to an empty string. If a secret is absent, the app should stop immediately.

Validation: treat secrets like inputs, not decorations

Validation isn’t only for format; it’s also for presence and sanity. For example, a token secret might be required to meet a minimum length, and a database URL should parse as a URL.

Example (Python-style validation):

```python
import os
from urllib.parse import urlparse

def require(name: str) -> str:
    v = os.getenv(name)
    if not v or not v.strip():
        raise RuntimeError(f"Missing required env var: {name}")
    return v

DB_URL = require("DB_URL")
parsed = urlparse(DB_URL)
if not parsed.scheme or not parsed.netloc:
    raise RuntimeError("DB_URL must be a valid URL")
```

This prevents a subtle failure mode: the app starts, but later authentication or database connections fail in confusing ways.

Logging: redact by default, and be careful with error messages

A lot of accidental secret exposure happens through logs. Common culprits include:

- printing environment variables during debugging
- logging request headers that include `Authorization`
- logging exception objects that contain the secret value

Example (redaction helper):

```python
import re

def redact(text: str) -> str:
    # Redact patterns like "Authorization: Bearer <token>"
    return re.sub(r"(Authorization:\s*Bearer\s*)[^\s]+", r"\1[REDACTED]", text)
```

Use redaction on any text that might include headers, tokens, or connection strings. Also ensure your error messages mention what's wrong without echoing the value.

Least privilege: credentials should do one job well

If a service only needs read access to a database, give it a read-only account. If an app only calls one external API, scope the token to that API and restrict permissions. Least privilege reduces the blast radius when something goes wrong.

A concrete checklist:

- separate credentials per environment (dev/staging/prod)
- separate credentials per service (web vs worker)
- separate credentials per function (read vs write)

Configuration files: keep them boring and non-secret

It's tempting to put everything in a `.env` file or a JSON config. That's fine for non-secret configuration, but secrets should not be stored in version-controlled files.

Repository hygiene example:

- Commit `config.example.json` with placeholders.
- Do not commit `config.json` containing real values.
- Add local env files to `.gitignore`.

Example `config.example.json`:

```
{
  "LOG_LEVEL": "info",
  "DB_URL": "postgres://user:REPLACE_ME@host:5432/db",
  "JWT_SECRET": "REPLACE_ME"
}
```

Your runtime config loader can require real values and reject placeholders.

Secrets in code: avoid “temporary” constants

Hard-coding secrets often starts as a quick test and then survives longer than intended. If you must reference a secret-like value for local development, use a placeholder and require the real value at runtime.

Example (reject placeholder):

```
const JWT_SECRET = required("JWT_SECRET");
if (JWT_SECRET === "REPLACE_ME") {
  throw new Error("JWT_SECRET must be set to a real value");
}
```

This makes it harder to accidentally run with insecure defaults.

Handling rotation: design so code doesn’t care

Rotation means replacing a secret without changing application code. The code should always read the current secret from its configured source at startup. If you need to support multiple active secrets (for example, verifying tokens signed with older keys), keep that logic explicit and bounded.

A simple pattern is to allow a list of verification keys while keeping signing keys separate. The important part is that the application’s behavior is driven by configuration, not by hard-coded key material.

Preventing accidental commits: treat the repo like a liability

Even careful teams can leak secrets. Reduce risk with automated checks and clear rules:

- add `.env*` and local config files to `.gitignore`
- use pre-commit hooks or CI checks that scan for common secret patterns
- review diffs that touch configuration loading

If a secret is committed, removing it from the latest commit is not enough; you must treat it as compromised and rotate it. The safest approach is to avoid commits of secrets entirely.

Startup behavior: fail fast, fail safely

When required secrets are missing or invalid, the app should stop with an error that helps operators fix the deployment. The message should name the missing variable or describe the format issue, but never include the secret itself.

Example error style:

- Good: “Missing required env var: JWT_SECRET”
- Risky: “JWT_SECRET=abcd1234 is invalid”

Quick checklist for 9.3

- [ ] Secrets are never hard-coded.
- [ ] Secrets are loaded from environment or a secrets store at startup.
- [ ] Required secrets are validated for presence (and basic format where appropriate).
- [ ] Logs never print secrets or authorization headers.
- [ ] Error messages explain the problem without echoing values.
- [ ] Credentials use least privilege and are separated by environment/service.
- [ ] Local config examples use placeholders; real values stay out of version control.

- [ ] Accidental commits are prevented with ignore rules and automated scanning.

When these pieces are in place, configuration becomes predictable and secrets become harder to leak. The app still fails when it should, but it fails in a way that points to the fix rather than the exposure.

9.4 Avoiding Unsafe Deserialization and Dangerous APIs

Deserialization turns bytes into objects. That convenience is also the main reason it can become dangerous: the input can carry instructions, not just data. The goal is simple—treat incoming data as data, not as something allowed to decide what code to run.

What “unsafe deserialization” usually means

Unsafe deserialization typically happens when:

- The code accepts an untrusted payload (from a client, message queue, file upload, or cache).
- The deserializer can instantiate arbitrary types or invoke special hooks during object creation.
- The payload is allowed to control class names, constructors, or method calls.

A safe system either:

- Uses a format that does not support code execution (like JSON with strict schemas), or
- Restricts deserialization to an allowlist of known types and disables dangerous behaviors.

Mind map: where the risk comes from

Mind map: Unsafe deserialization & dangerous APIs

Click here to view the mind map: Unsafe deserialization & dangerous APIs

Dangerous APIs to treat as “no by default”

Even if you never deserialize, some APIs can recreate the same risk pattern by letting input influence code execution.

1) Dynamic code execution

If user input reaches `eval`, `exec`, or runtime compilation, the input can become executable logic.

Example (unsafe):

```
# user_input comes from a request
result = eval(user_input)
```

Example (safer):

```
import ast

def safe_int(expr: str) -> int:
    node = ast.parse(expr, mode='eval')
    if not isinstance(node.body, ast.Constant) or not isinstance(node.body.value, int):
        raise ValueError('Only integer literals are allowed')
    return node.body.value
```

The key idea is not “sanitize the string,” but “only accept a narrow structure.”

2) Template engines with untrusted templates

Rendering a template that comes from a user can allow access to internals depending on the engine’s capabilities.

Example (unsafe):

```
// templateText comes from a request
res.send(renderTemplate(templateText, data));
```

Example (safer):

- Only render server-owned templates.
- Treat user input as data, not as template source.

Deserialization patterns that commonly go wrong

Pattern A: Accepting type metadata

Some formats include a "type" field that tells the deserializer what class to create. If you accept that field from untrusted input, you've handed the attacker a class-selection menu.

Example (unsafe concept):

```
// Pseudocode: payload includes a type name
Object obj = riskyDeserializer.deserialize(payload);
```

Safer approach:

- Use a format without type metadata.
- Or ignore the type field and map to a known schema.

Pattern B: Deserializing into a broad type

If you deserialize into `Object`, `Any`, or a generic map without validation, you lose control over structure. Attackers can exploit unexpected shapes to trigger edge cases in downstream code.

Example (unsafe):

```
const obj: any = JSON.parse(req.body);
// later code assumes fields exist and are safe
```

Safer approach:

- Validate the shape before use.
- Convert into a specific type.

Pattern C: Polymorphic decoding

Polymorphism is useful, but it's also where "input chooses the class" often hides.

Example (unsafe concept):

```
// Pseudocode: serializer picks derived type based on payload
BaseMessage msg = serializer.Deserialize<BaseMessage>(payload);
```

Safer approach:

- Disable automatic polymorphism.
- Use an explicit discriminator you validate against an allowlist.

Practical mitigations you can apply immediately

1) Prefer data-only formats and strict schemas

JSON is safer than many binary formats because it doesn't inherently carry executable type information. Still, JSON can be malicious in shape, size, and content.

Example (schema-first validation in spirit):

```
type LoginRequest = { user: string; password: string };

function parseLoginRequest(body: unknown): LoginRequest {
  if (typeof body !== 'object' || body === null) throw new Error('Invalid body');
  const b = body as any;
  if (typeof b.user !== 'string' || typeof b.password !== 'string') throw new Error('Bad fields');
  return { user: b.user, password: b.password };
}
```

This prevents "unexpected fields" from silently flowing into business logic.

2) Use allowlists for types when you must deserialize

If your system must deserialize into objects, restrict what can be created.

Example (conceptual allowlist):

```
ALLOWED_TYPES = {
  'UserCreated': UserCreated,
  'OrderPlaced': OrderPlaced,
}

def safe_deserialize(payload: dict):
  t = payload.get('type')
  cls = ALLOWED_TYPES.get(t)
  if cls is None:
    raise ValueError('Type not allowed')
  return cls.from_dict(payload)
```

Notice that the allowlist controls the class, and `from_dict` should only map fields—no arbitrary constructor side effects.

3) Disable dangerous hooks and features

Many runtimes have "magic" methods or hooks that run during object creation or property setting. If those hooks can perform I/O, execute commands, or access the filesystem, they become an attack surface.

Mitigation checklist:

- Turn off automatic gadget-like behaviors (where available).
- Avoid deserializing into classes with side-effectful constructors.
- Prefer immutable data transfer objects (DTOs) that only store fields.

4) Validate size and structure before parsing

Even "safe" deserialization can be abused for denial of service.

Example (simple guard):

```
if len(body) > 1<<20 { // 1 MiB
  return errors.New("payload too large")
}
```

Then validate required fields and reject unknown shapes.

5) Add tests that prove the defenses

Write tests that feed malformed or malicious payloads and assert rejection.

Example test ideas (language-agnostic):

- Payload with an unknown `type` field.
- Payload with correct `type` but wrong field types (e.g., string instead of number).
- Payload with nested structures that exceed expected depth.
- Payload that includes extra fields that should be ignored or rejected.

Worked example: turning “object deserialization” into “message parsing”

Suppose you receive a message that should become a `PaymentEvent`.

Unsafe approach:

- Deserialize directly into `PaymentEvent` using a mechanism that can instantiate arbitrary types.

Safer approach:

- Parse into a plain structure (map/dict).
- Validate fields.
- Construct a DTO with explicit mapping.

Example (conceptual):

```
def parse_payment_event(raw: dict) -> PaymentEvent:
  if raw.get('type') != 'PaymentEvent':
    raise ValueError('Wrong event type')
  amount = raw.get('amount')
  if not isinstance(amount, (int, float)) or amount <= 0:
    raise ValueError('Bad amount')
  return PaymentEvent(order_id=str(raw.get('order_id')), amount=float(amount))
```

This keeps the input from selecting classes or triggering side effects.

Quick checklist

- Treat all incoming payloads as untrusted.
- Avoid deserializers that can create arbitrary types from input.
- Use strict schemas and explicit field validation.
- If type selection is required, enforce an allowlist and map fields manually.
- Disable or avoid side-effectful constructors and deserialization hooks.
- Add tests for unknown types, wrong shapes, and oversized payloads.

When you follow these rules, deserialization becomes a predictable data transformation rather than a doorway into code execution.

9.5 Reviewing Code for Authorization and Access Control Bugs

Authorization bugs are rarely about “missing a permission check” in one obvious place. More often, they appear as mismatches between *who the user is*, *what the code thinks the user is allowed to do*, and *what the data actually represents*. This section gives you a practical review checklist, plus examples you can map directly onto your own code.

The core question reviewers should ask

For any endpoint or function that returns or modifies data, ask:

1. **Where does the user identity come from?** (token/session → user id)
2. **Where is authorization decided?** (policy/guard/service)
3. **Where is the authorization enforced?** (query filters, object-level checks, side effects)
4. **What data shape is assumed?** (ownership fields, tenant ids, roles)
5. **What happens on failure?** (status codes, error messages, logging)

If any step is missing or inconsistent, you likely have an access control bug.

Mind map: authorization review flow

Click here to view the mind map: Authorization & Access Control Bug Review

A practical checklist (use it like a code review rubric)

A. Identity and trust boundaries

- Confirm the code uses a server-validated identity (session/JWT verification), not a client-supplied user id.
- Check whether the code trusts request headers like `X-User-Id` or `X-Tenant-Id`. If it does, treat it as suspicious.

B. Authorization decision vs enforcement

- Ensure the code does not "decide" authorization and then forget to apply it to the database query.
- Look for patterns like: "if user has permission, proceed," followed by a query that does not include ownership/tenant constraints.

C. Object-level access

- For endpoints that accept an object id (e.g., `documentId`), verify the object is fetched in a way that binds it to the user's allowed scope.
- Prefer fetching with constraints: `getByIdAndOwner(documentId, userId)` rather than fetching by id and checking afterward.

D. Tenant and scope boundaries

- If your app is multi-tenant, confirm every data access includes tenant scoping.
- Watch for joins and "helper" queries that omit tenant filters.

E. Write paths and side effects

- Many reviews focus on reads. Make sure the same authorization rules apply to writes: updates, deletes, exports, background jobs, and webhooks.
- Check asynchronous flows: a job might run with insufficient context or with a stale authorization decision.

F. Error handling and information leakage

- Ensure forbidden responses don't leak whether an object exists.
- A common pattern is returning `404` for both "not found" and "not allowed," depending on your security posture.

Example 1: IDOR via missing ownership filter

Buggy code (conceptual):

- The endpoint checks that the user has a general permission.
- It then loads the object by id only.

```
def update_document(request, document_id):
    user = request.user
    if not user.has_perm("documents:edit"):
        return 403

    doc = Document.get(id=document_id)  # missing owner/tenant constraint
    doc.title = request.json["title"]
    doc.save()
    return 200
```

Why it's wrong: any user with `documents:edit` can update someone else's document by guessing ids.

Safer pattern: bind the lookup to the user's scope.

```python
def update_document(request, document_id):
    user = request.user
    if not user.has_perm("documents:edit"):
        return 403

    doc = Document.get_or_none(id=document_id, owner_id=user.id)
    if doc is None:
        return 404

    doc.title = request.json["title"]
    doc.save()
    return 200
```

Note the review nuance: the permission check is still useful, but the enforcement happens in the query.

Example 2: “Check then query” mismatch

Buggy code:

- The code authorizes access to a project.
- Then it queries tasks using only `project_id`.

```javascript
app.get('/projects/:projectId/tasks', (req, res) => {
  const userId = req.user.id;
  const projectId = req.params.projectId;

  authorizeUserForProject(userId, projectId); // throws if forbidden

  const tasks = db.tasks.findMany({ where: { projectId } });
  res.json(tasks);
});
```

Why it’s wrong: if `authorizeUserForProject` is correct but the tasks query doesn’t include tenant/ownership constraints, you can still leak tasks if `projectId` crosses boundaries due to a bug elsewhere (or a future refactor).

Safer pattern: enforce scope in the data access layer.

```javascript
app.get('/projects/:projectId/tasks', (req, res) => {
  const userId = req.user.id;
  const projectId = req.params.projectId;

  const tasks = db.tasks.findMany({
    where: { projectId, ownerId: userId }
  });

  res.json(tasks);
});
```

This doesn’t remove the need for authorization logic; it ensures enforcement is hard to forget.

Example 3: Confused deputy in service-to-service calls

Bug pattern: a service uses an identity from the request body instead of the authenticated caller.

```
type UpdateRequest struct {
  UserID string `json:"userId"`
  DocID  string `json:"docId"`
}

func (h *Handler) UpdateDoc(ctx context.Context, req UpdateRequest) {
  // Bug: req.UserID is untrusted
  if !h.auth.CanEdit(req.UserID) {
    return
  }
  h.repo.UpdateDoc(req.DocID, req.UserID)
}
```

Review fix: derive the user id from verified auth context and treat request fields as data, not identity.

Example 4: Missing checks on export and background jobs

A common review miss: "We checked the endpoint, so the job is safe." That's only true if the job uses the same scope.

Review rule: every path that performs side effects must carry enough authorization context to enforce it at execution time.

- If a background job exports data, confirm it filters by tenant/owner.
- If it receives a list of ids, confirm the job re-validates scope or re-derives scope from the authenticated job context.

How to structure negative tests during review

When you review code, you should also review the tests that would catch it. For each endpoint, ensure you have at least:

- **Forbidden user**: same endpoint, different user id.
- **Wrong tenant**: same object id exists in another tenant.
- **Missing object**: object id doesn't exist (to confirm error behavior).

Example test idea (language-agnostic):

- User A has permission.
- User A requests `document_id` owned by User B.
- Expect `404` or `403` (based on your policy), and verify no update occurred.

A final reviewer's pass: "authorization invariants"

Before approving, confirm these invariants hold across the code path:

- **Invariant 1:** every database query that returns protected objects includes the correct scope (tenant/owner).
- **Invariant 2:** every write operation is gated by the same scope rules as reads.
- **Invariant 3:** identity comes only from verified auth context.
- **Invariant 4:** error responses do not reveal sensitive existence information more than intended.

If you can't state these invariants for the endpoint, that's a sign the code needs either clearer structure or more explicit enforcement.

10. Working With APIs and Integrations

10.1 Generating Robust Client Code for REST APIs

Robust REST client code does three jobs at once: it builds correct requests, it interprets responses safely, and it fails in ways that are easy to debug. The trick is to treat the API contract as data, not as vibes.

What "robust client code" means in practice

A good client:

- **Encodes inputs correctly** (path params, query params, headers, body).
- **Validates outputs** (status codes, required fields, types).
- **Handles errors consistently** (including non-JSON error bodies).

- **Supports timeouts and retries carefully** (only where they make sense).
- **Keeps behavior predictable** (no silent fallbacks that hide bugs).

Mind map: client responsibilities

Click here to view the mind map: REST Client

Step 1: Start from the contract (even if it's informal)

Before generating code, write down what you expect for each endpoint:

- **Method + path**: e.g., `GET /users/{id}`.
- **Inputs**: which are required, which are optional, and their types.
- **Outputs**: success payload shape and which status codes are considered success.
- **Error payload**: what fields appear when things go wrong.

If the API spec is missing, you can still extract a contract from examples: request/response pairs, plus a list of observed status codes.

Step 2: Generate a typed request layer

A common failure mode is mixing "stringly typed" parameters with ad-hoc URL building. Instead, centralize URL construction and serialization.

Example: building a request safely (TypeScript)

```
type User = { id: string; name: string; email?: string };

type ApiError = { code?: string; message: string; details?: unknown };

class ApiClient {
  constructor(private baseUrl: string) {}

  async getUser(id: string): Promise<User> {
    const url = new URL(`/users/${encodeURIComponent(id)}`, this.baseUrl);
    const res = await fetch(url.toString(), {
      method: 'GET',
      headers: { 'Accept': 'application/json' },
      // timeout handled by wrapper in real code
    });

    return this.handleJson<User>(res);
  }

  private async handleJson<T>(res: Response): Promise<T> {
    const contentType = res.headers.get('content-type') || '';
    const isJson = contentType.includes('application/json');

    if (!res.ok) {
      const errBody = isJson ? await res.json().catch(() => null) : null;
      const apiErr: ApiError =
        errBody && typeof errBody === 'object' && 'message' in errBody
          ? (errBody as ApiError)
          : { message: `HTTP ${res.status}` };
      throw new Error(apiErr.message);
    }

    if (!isJson) throw new Error(`Expected JSON, got: ${contentType || 'none'}`);
    return (await res.json()) as T;
  }
}
```

This example is intentionally strict: it refuses non-JSON success bodies and it tries to parse JSON error bodies only when the server claims they are JSON.

Step 3: Map status codes to behavior

Not every non-2xx response should become the same error. For example:

- `400` often means client input is wrong.
- `401/403` means auth/permissions.
- `404` might be expected in “lookup” flows.
- `409` often indicates a conflict you may want to surface differently.

A robust client exposes this as structured errors, not just `Error("HTTP 500")`.

Example: structured HTTP error mapping (TypeScript)

```
class HttpError extends Error {
  constructor(
    public status: number,
    public body: unknown,
    message: string
  ) { super(message); }
}

function mapStatusToMessage(status: number, body: any): string {
  const msg = body?.message;
  if (typeof msg === 'string' && msg.trim()) return msg;
  if (status === 404) return 'Resource not found';
  if (status === 409) return 'Conflict';
  return `Request failed with status ${status}`;
}
```

Then, in `handleJson`, throw `new HttpError(res.status, errBody, mapStatusToMessage(...))`.

Step 4: Validate response shape (without turning it into a thesis)

Type assertions (`as T`) don’t validate anything. For robustness, add lightweight checks for required fields.

Example: minimal runtime validation

```
function isUser(x: any): x is User {
  return x && typeof x === 'object' &&
    typeof x.id === 'string' &&
    typeof x.name === 'string' &&
    (x.email === undefined || typeof x.email === 'string');
}

async function parseUser(res: Response): Promise<User> {
  if (!res.ok) throw new Error(`HTTP ${res.status}`);
  const data = await res.json();
  if (!isUser(data)) throw new Error('Invalid user payload');
  return data;
}
```

This catches mismatches early, which is especially useful when the server changes or when a proxy returns an HTML error page.

Step 5: Handle query params and pagination predictably

Query params are where bugs hide: missing encoding, wrong defaults, and inconsistent pagination.

Example: query builder and pagination helper

```ts
type ListUsersParams = { page?: number; pageSize?: number; q?: string };

function buildListUsersUrl(baseUrl: string, params: ListUsersParams) {
  const url = new URL('/users', baseUrl);
  if (params.page !== undefined) url.searchParams.set('page', String(params.page));
  if (params.pageSize !== undefined) url.searchParams.set('pageSize', String(params.pageSize));
  if (params.q !== undefined) url.searchParams.set('q', params.q);
  return url;
}
```

For pagination, decide whether the API uses `page/pageSize`, `limit/offset`, or cursor tokens, and encode that choice into the client method signature so callers don't guess.

Step 6: Make retries and timeouts explicit

Retries are not universal. A safe rule of thumb:

- Retry **idempotent** requests (GET, PUT in many systems) when the failure is likely transient (timeouts, connection resets).
- Avoid retrying non-idempotent POST unless the API explicitly supports it via idempotency keys.

Even if you don't implement retries yet, design the client so adding them later doesn't require rewriting every method.

Mind map: error handling strategy

Click here to view the mind map: Error Handling

Putting it together: a robust method template

A consistent method pattern reduces surprises:

1. Build URL with encoding.
2. Set headers (Accept, Content-Type when needed).
3. Perform request with timeout.
4. If `!res.ok`, parse error body carefully and throw a structured error.
5. Parse success body, validate shape, return typed result.

That pattern is boring in the best way: it makes debugging feel like following a checklist instead of reading tea leaves.

10.2 Handling Retries, Timeouts, and Backoff Correctly

Retries are useful when failures are temporary, but harmful when they're permanent or when they amplify load. The goal is simple: retry only the right failures, stop at the right time, and wait in a way that reduces contention.

Core principles (in practical terms)

1. **Timeouts bound damage.** A retry without a timeout can turn a small outage into a pile of stuck requests.
2. **Retries need a stop rule.** Use a maximum attempt count and/or a total time budget.
3. **Backoff should reduce synchronized retries.** If everyone retries at the same interval, you get a thundering herd.
4. **Not all errors deserve retries.** Retry network glitches and transient server errors; don't retry validation errors or "not found."
5. **Idempotency decides safety.** Retrying a non-idempotent operation can create duplicates unless you use safeguards.

Mind map: retry strategy

Retry Strategy Mind Map

Click here to view the mind map: Retry Strategy

Timeouts: choose them like a grown-up

A timeout should reflect the service's expected behavior and your tolerance. If you set it too low, you'll retry healthy requests; too high, and you'll waste threads.

A common pattern is to separate **connect timeout** and **read timeout**:

- **Connect timeout**: how long to wait for a TCP connection.
- **Read timeout**: how long to wait for the response after the connection is established.

Example: suppose your API typically responds in 200–400 ms, but occasionally takes up to 1.5 s under load. A reasonable starting point might be:

- connect timeout: 300 ms
- read timeout: 2 s

Then you retry only when the failure is a timeout or a transient server error.

Backoff: exponential with jitter (and a cap)

Exponential backoff increases delay as attempts grow. Jitter randomizes the delay so multiple clients don't retry in lockstep.

A practical formula for attempt `n` (starting at 0) is:

$$\text{delay}(n) = \min(\text{cap}, ; (\text{base} \times 2^n) + \text{jitter})$$

Where jitter can be random in a range, such as `0..base`.

Mind map: backoff mechanics

Backoff Mechanics Mind Map

Click here to view the mind map: Backoff Mechanics

Retry policy: map errors to decisions

Create a small decision table in code. The key is to classify errors consistently.

Example decision table

Condition	Retry?	Why
Timeout (connect/read)	Yes	Often transient
Connection reset	Yes	Could be a temporary disruption
HTTP 502/503/504	Yes	Commonly temporary upstream issues
HTTP 429	Yes (with care)	Rate limiting; respect Retry-After if present
HTTP 400/422	No	Client input is wrong
HTTP 401/403	No	Auth/permission won't fix itself
HTTP 404	No	Usually permanent
HTTP 409 (conflict)	Maybe	Depends on operation semantics

Example: safe retry for GET requests

For idempotent reads, retries are usually safe. Here's a compact approach in JavaScript-like pseudocode.

```
async function fetchWithRetry(url, fetchFn) {
  const maxAttempts = 4;
  const baseMs = 100;
  const capMs = 2000;
  const totalBudgetMs = 5000;
  const start = Date.now();

  for (let attempt = 0; attempt < maxAttempts; attempt++) {
    try {
      return await fetchFn(url, { timeoutMs: 2000 });
    } catch (err) {
      const retryable = isRetryable(err);
      const elapsed = Date.now() - start;
      if (!retryable || attempt === maxAttempts - 1 || elapsed > totalBudgetMs) {
        throw err;
      }
      const jitter = Math.floor(Math.random() * baseMs);
      const delay = Math.min(capMs, baseMs * (2 ** attempt) + jitter);
      await sleep(delay);
    }
  }
}
```

Key details:

- `timeoutMs` prevents hanging.
- `isRetryable(err)` centralizes the decision.
- `totalBudgetMs` stops retries even if attempts remain.
- Jitter reduces synchronized retries.

Example: retries for POST with idempotency keys

For writes, retries can duplicate side effects unless the server can deduplicate. Use an **idempotency key** so repeated requests map to the same logical operation.

Workflow:

1. Client generates a unique key per logical action.
2. Client sends it with the request.
3. Server stores the key and returns the same result for duplicates.

```
async function postWithRetry(url, body, idempotencyKey, postFn) {
  const maxAttempts = 3;
  const baseMs = 150;
  const capMs = 1500;

  for (let attempt = 0; attempt < maxAttempts; attempt++) {
    try {
      return await postFn(url, body, {
        timeoutMs: 2000,
        headers: { 'Idempotency-Key': idempotencyKey }
      });
    } catch (err) {
      const retryable = isRetryable(err);
      if (!retryable || attempt === maxAttempts - 1) throw err;
      const jitter = Math.floor(Math.random() * baseMs);
      const delay = Math.min(capMs, baseMs * (2 ** attempt) + jitter);
      await sleep(delay);
    }
  }
}
```

This pattern keeps retries from turning a temporary failure into duplicate records.

Handling Retry-After for 429

When a server responds with HTTP 429, it often includes `Retry-After`. Respecting it is better than guessing.

Rules:

- If `Retry-After` is present, wait that long (bounded by your total budget).
- If it's missing, fall back to your backoff.

Logging and metrics: make retries visible

Retries should be measurable. At minimum, record:

- attempt number
- error category (timeout, 5xx, connection reset)
- final outcome (success or final failure)
- total time spent

This helps you distinguish "we recovered" from "we kept retrying and still failed."

Common mistakes to avoid

- **Retrying everything.** You'll amplify failures and waste time.
- **No jitter.** Many clients will retry together.
- **No cap or no budget.** Delays can grow until requests pile up.
- **Retrying non-idempotent writes without protection.** Use idempotency keys or other safeguards.
- **Using one timeout for everything.** Connect and read often need different values.

Quick checklist

- [ ] Timeout is set for each attempt.
- [ ] Retryable errors are explicitly identified.
- [ ] Max attempts and total budget are enforced.
- [ ] Backoff is exponential with jitter and a cap.
- [ ] Writes use idempotency keys (or equivalent).
- [ ] 429 handling respects `Retry-After` when available.
- [ ] Retries are logged with attempt counts and error categories.

10.3 Validating Schemas and Handling Versioning Safely

When you validate schemas, you're doing two jobs at once: confirming that incoming data matches what your code expects, and creating a clear boundary for what "valid" means. When you handle versioning safely, you're making sure that boundary doesn't quietly move under your feet.

What to validate (and what to ignore)

Start by validating the smallest set of properties that protect correctness.

- **Validate required fields**: If `userId` is required to look up a record, treat it as required.
- **Validate types and shapes**: A string where you expect a number should fail early.
- **Validate constraints**: For example, `age` must be `>= 0`.
- **Validate enumerations**: If `status` can only be `"active" | "paused"`, reject anything else.
- **Validate cross-field rules**: Example: `endDate` must be after `startDate`.

Avoid validating everything "just because you can." Over-validation makes migrations painful and turns harmless differences into hard failures.

Mind map: schema validation and versioning

Click here to view the mind map: Schema Validation + Versioning (Safe Handling)

A practical schema validation example

Assume you receive events from clients. You want to validate the payload before processing.

Incoming payload (JSON):

- `schemaVersion`: integer
- `eventType`: string
- `userId`: string
- `occurredAt`: ISO timestamp
- `metadata`: object (optional)

A good validation approach is to validate **per version**.

Example: versioned validation in code

```ts
type EventV1 = {
  schemaVersion: 1;
  eventType: "login" | "logout";
  userId: string;
  occurredAt: string; // ISO
  metadata?: Record<string, unknown>;
};

function validateEvent(payload: any): EventV1 {
  if (payload?.schemaVersion !== 1) throw new Error("Unsupported schemaVersion");
  if (payload?.eventType !== "login" && payload?.eventType !== "logout") throw new Error("Bad eventType");
  if (typeof payload?.userId !== "string" || payload.userId.length === 0) throw new Error("Bad userId");
  if (typeof payload?.occurredAt !== "string" || Number.isNaN(Date.parse(payload.occurredAt))) {
    throw new Error("Bad occurredAt");
  }
  if (payload?.metadata !== undefined && typeof payload.metadata !== "object") throw new Error("Bad metadata");
  return payload as EventV1;
}
```

This is intentionally strict. If the payload doesn't match, you stop before any side effects.

Handling versioning safely: define compatibility rules

Versioning becomes safe when you write down what changes are allowed.

Use these rules as a starting point:

- **Backward compatible (safe to accept old + new):**
  - Add new optional fields.
  - Widen constraints carefully (e.g., allow additional enum values if your code can handle them).
  - Add new nested objects as optional.
- **Breaking (requires new version or explicit migration):**
  - Rename fields.
  - Change the meaning of a field.
  - Change a field type (string → number).
  - Remove required fields.

A common mistake is treating "optional" as "free." If your code relies on a field for correctness, it's effectively required even if the schema marks it optional.

Versioning patterns that work in practice

1) Version in the payload

Add `schemaVersion` to the payload so the receiver can choose the right validator.

Example decision logic:

- If `schemaVersion === 1`, validate with V1 rules.
- If `schemaVersion === 2`, validate with V2 rules.
- Otherwise, reject with a clear error.

This pattern is useful when you can't control the endpoint URL.

2) Version in the endpoint

Use routes like `/api/v1/events` and `/api/v2/events`.

This pattern is useful when you want different behavior and different response shapes.

3) Version in headers

Use something like `Content-Type` plus a custom version header.

This pattern is useful when you want to keep payloads clean, but it requires careful client coordination.

Validating requests vs responses

Validation isn't only for incoming data.

- **Validate requests** to prevent invalid inputs from causing incorrect behavior.
- **Validate responses** to catch bugs where your server accidentally emits the wrong shape.

Response validation can be lightweight: check required fields and types, not every optional detail.

Error handling that helps debugging without leaking details

When validation fails, return an error that is specific enough to act on.

A good error response includes:

- A stable error code (e.g., `INVALID_EVENT_SCHEMA`)
- The schema version you expected
- A list of field issues (e.g., `userId must be a non-empty string`)

Avoid echoing the entire payload if it may contain sensitive data.

Example: mapping validation errors to a structured response

```
function formatValidationError(version: number, issues: string[]) {
  return {
    errorCode: "INVALID_EVENT_SCHEMA",
    expectedSchemaVersion: version,
    issues: issues.slice(0, 10)
  };
}
```

Then your handler can return `formatValidationError(1, [...])` with an appropriate HTTP status.

Mind map: versioning decisions

Click here to view the mind map: Versioning Decisions

Testing versioned schemas

Write tests that cover:

- Valid payloads for each version.
- Invalid payloads that fail for the right reason.
- Edge cases like missing optional objects, empty strings, and wrong enum values.

A useful test pattern is to keep fixtures small and focused.

Example test cases (conceptual):

- V1 valid: `eventType: "login"`, `userId: "u123"`, `occurredAt: "2026-01-01T00:00:00Z"`
- V1 invalid: `occurredAt: "not-a-date"`
- V1 invalid: `eventType: "signup"` (unknown enum)

- V1 invalid: `schemaVersion: 2` (wrong version)

A safe migration approach (when you must change meaning)

If you need to change a field's meaning, don't pretend it's the same field under a new name. Instead:

- Introduce a new version.
- Validate using the new rules.
- If you want to support old clients, migrate old payloads explicitly by mapping fields with clear rules.

Migration should be deterministic and testable. If you can't write a mapping without guesswork, treat it as breaking and require the new version.

Summary

Validate schemas to enforce a precise contract, and handle versioning by making compatibility rules explicit. Validate per version, return structured errors, and test each version's behavior. This keeps your code predictable even when inputs evolve.

10.4 Writing Idempotent Operations With Examples

Idempotent operations produce the same result even if you run them multiple times with the same inputs. In practice, this means you can safely retry after timeouts, handle duplicate messages, and avoid "double charge" bugs. The trick is to design the operation so that the second run either does nothing or converges to the same final state.

What "idempotent" means in code

An operation is idempotent if:

- Applying it once yields state S_1.
- Applying it again yields the same state S_1 (not S_2).

A useful mental model: the operation should be a state transition that is safe to repeat.

Mind map: idempotency design choices

Click here to view the mind map: Idempotent Operations: Design Map

Technique 1: Idempotency keys (request-level)

Use an idempotency key to identify a request. Store the outcome the first time, and return the stored outcome on retries.

Example scenario: an endpoint creates a payment. The client retries if it doesn't get a response, but the server might already have created the payment.

Data model (conceptual):

- `idempotency_requests(idempotency_key, user_id, status, result_json, created_at)`
- `payments(payment_id, user_id, amount, status, created_at)`

Flow:

1. Client sends `Idempotency-Key: <uuid>`.
2. Server starts a transaction.
3. Insert into `idempotency_requests` with a unique constraint on `(idempotency_key, user_id)`.
4. If insert succeeds, perform the side effect and store the result.
5. If insert fails due to uniqueness, fetch the stored result and return it.

Why this works: the second run doesn't create a second payment; it reuses the recorded outcome.

Technique 2: Upsert / merge semantics (data-level)

If the operation's effect can be expressed as "set this record to these values," you can use upsert patterns.

Example: "ensure a user has a profile row."

- Non-idempotent approach: insert a new profile row every time.

- Idempotent approach: upsert by a stable key (e.g., `user_id`).

SQL example (PostgreSQL):

```
INSERT INTO user_profiles (user_id, display_name, updated_at)
VALUES ($1, $2, NOW())
ON CONFLICT (user_id)
DO UPDATE SET
  display_name = EXCLUDED.display_name,
  updated_at = NOW();
```

This is idempotent with respect to the final row contents. Running it twice with the same `$2` yields the same `display_name` (though `updated_at` changes; if you need strict equality, set `updated_at` only on first insert).

Technique 3: Conditional updates (compare-and-set)

When you need to prevent double transitions, update only if the record is in an expected state.

Example: "mark an order as shipped."

- If it's already shipped, do nothing.
- If it's not shipped, transition once.

SQL example:

```
UPDATE orders
SET status = 'shipped', shipped_at = NOW()
WHERE order_id = $1 AND status <> 'shipped'
RETURNING order_id, status, shipped_at;
```

If the update returns no rows, the operation has already been applied (or the order is in a state you don't want to ship). This pattern is naturally idempotent because the second call won't match the `WHERE` clause.

Technique 4: Deduplication tables for side effects

For operations that call external systems (email, webhooks, third-party APIs), you can't rely on database constraints alone. Use a deduplication table to record that the side effect was performed.

Example: send a "welcome email" exactly once per user.

- Table: `outbox_events(event_type, user_id, processed_at, payload_hash)` with a unique constraint on `(event_type, user_id)`.
- When sending, insert the outbox row first.
- Only if the insert succeeds do you call the external email service.

This prevents duplicate sends when retries happen after the external call.

Technique 5: Deterministic behavior and stable identifiers

Some operations become idempotent by making the output deterministic.

Example: generating an invoice number from `(customer_id, invoice_period)`.

- If you compute the invoice identifier deterministically and store it with a unique constraint, retries won't create duplicates.

Even when you still need idempotency keys for request tracking, deterministic identifiers help ensure the data-level effect converges.

Putting it together: an end-to-end payment example

Goal: `POST /payments` creates a payment once, even if the client retries.

Implementation sketch (transactional):

1. Read `Idempotency-Key`.
2. Begin transaction.

3. Attempt to insert a row into `idempotency_requests`.
4. If the row already exists, return the stored `result_json`.
5. Otherwise, create the payment row.
6. Store `result_json` in `idempotency_requests`.
7. Commit.

Key detail: the unique constraint is what makes step 3 safe under concurrency.

Common pitfalls (and how to avoid them)

1. **Storing "success" too early.** If you mark the idempotency record as completed before the side effect finishes, a crash can leave you with a "completed" record but no real side effect.
2. **Using a weak idempotency key.** Keys must be stable per logical request. If you include timestamps or random fields, retries won't match.
3. **Ignoring partial writes.** If you write multiple tables, wrap them in a transaction or use compensating logic.
4. **Returning different responses on retry.** Idempotent operations should return the same outcome for the same request. If you can't, at least return a consistent "already processed" response.
5. **Relying on application-level checks without constraints.** "Check then insert" without a unique constraint is race-condition bait.

Testing idempotency (practical checklist)

- **Repeat the same request N times** and verify:
  - Row counts don't grow unexpectedly.
  - External side effects happen once (use a fake sender in tests).
  - The final state matches the first successful run.
- **Simulate failure after side effects** by injecting a crash right after the payment insert but before the idempotency record is updated; confirm your transaction boundaries or ordering prevent inconsistent "completed" markers.
- **Run concurrent retries** to ensure unique constraints and transaction logic behave correctly.

Quick reference: choosing the right pattern

- You control the request and can pass a key: **Idempotency keys**.
- The operation is "set a row to these values": **Upsert**.
- You need a one-time state transition: **Conditional update**.
- You call external systems: **Deduplication/outbox**.
- You can derive stable identifiers: **Deterministic IDs + constraints**.

Idempotency is less about a single keyword and more about designing the operation so that retries don't change the meaning of the request. Once you pick a pattern, enforce it with constraints and tests, and the code will behave predictably even when the network doesn't.

10.5 Testing Integrations With Mocks and Contract Checks

Integration tests fail for two different reasons: the code is wrong, or the other system behaves differently than you assumed. Mocks catch the first case quickly, while contract checks catch the second case by verifying the shape and meaning of requests and responses. The trick is to use both without turning your tests into a pile of brittle expectations.

Mind map: integration testing with mocks + contract checks

Click here to view the mind map: Integration testing (10.5)

A practical split: unit logic vs integration wiring

Before mocks and contracts, decide what you're testing.

- **Unit tests** cover transformations you own: mapping DTOs to domain objects, computing totals, validating inputs.
- **Integration tests** cover wiring: your client builds the right HTTP request, your code handles the right response codes, and your retry logic triggers under the right conditions.
- **Contract checks** cover compatibility: the other side's payload shape and error format match what you expect.

If you skip unit tests and rely only on mocks, you'll end up testing your own code indirectly through expectations. If you skip contract checks and rely only on real calls, you'll get slow tests and confusing failures.

Mocks: realistic behavior, not just “it returns 200”

A good mock answers three questions: **what request did you receive**, **what response did you return**, and **what failure modes did you simulate**.

Example: mocking an HTTP integration

Suppose your service calls `POST /v1/payments/charge` and expects a response like:

- `200` with `{ "paymentId": "...", "status": "succeeded" }`
- `400` with `{ "error": { "code": "invalid_request", "message": "..." } }`

A mock should match on the important parts of the request and return payloads that resemble production.

```
# Example using a generic mock server pattern (pseudo-Python)

def mock_charge_endpoint(request):
    assert request.method == "POST"
    assert request.path == "/v1/payments/charge"
    assert request.headers["Content-Type"] == "application/json"

    body = request.json()
    assert "amount" in body and "currency" in body

    if body["amount"] <= 0:
        return {
            "status": 400,
            "json": {"error": {"code": "invalid_request", "message": "amount must be > 0"}}
        }

    return {
        "status": 200,
        "json": {"paymentId": "pay_123", "status": "succeeded"}
    }
```

Key details:

- The mock asserts request invariants. This catches bugs in request construction.
- The mock returns realistic error shapes. This catches bugs in error parsing.
- The mock simulates a meaningful failure mode (invalid amount), not a random one.

Example: testing retry behavior with a mock

If your client retries on `503` but not on `400`, your mock should produce a sequence of responses.

```
# Pseudo-code for sequential mock responses

responses = [
  {"status": 503, "json": {"error": {"code": "service_unavailable", "message": "try later"}}},
  {"status": 503, "json": {"error": {"code": "service_unavailable", "message": "try later"}}},
  {"status": 200, "json": {"paymentId": "pay_123", "status": "succeeded"}}
]

mock_server.set_route("POST", "/v1/payments/charge", responses)

result = client.charge(amount=10, currency="USD")
assert result.payment_id == "pay_123"
assert mock_server.call_count("POST", "/v1/payments/charge") == 3
```

This test is about your retry policy, not about the payment provider. The mock gives you control over the sequence.

Contract checks: verify shape and meaning, not just fields

A contract check answers: **does the provider response still match what the consumer expects?**

You can implement contract checks in two common styles:

1. **Consumer-driven contracts**: your service defines what it expects, and the provider verifies it can satisfy those expectations.

2. **Provider-driven contracts**: the provider defines what it guarantees, and your service verifies your requests and parsing logic match those guarantees.

Even if your project doesn't use a dedicated contract framework, you can still do contract checks by validating JSON payloads against schemas and by asserting error object structure.

Example: contract schema for a success response

Define a schema-like set of rules for the response.

- `paymentId` is a non-empty string
- `status` is one of `succeeded`, `failed`, `pending`

```
{
  "type": "object",
  "required": ["paymentId", "status"],
  "properties": {
    "paymentId": {"type": "string", "minLength": 1},
    "status": {"type": "string", "enum": ["succeeded", "failed", "pending"]}
  },
  "additionalProperties": false
}
```

Your integration test can validate the mock response against this schema, and your contract test can validate real provider responses (or provider-supplied fixtures) against it.

Example: contract schema for an error response

Error payloads often drift because teams add fields or change nesting. Lock down the structure you rely on.

- `error.code` is a string
- `error.message` is a string
- top-level response is either success or error, not both

```
{
  "type": "object",
  "required": ["error"],
  "properties": {
    "error": {
      "type": "object",
      "required": ["code", "message"],
      "properties": {
        "code": {"type": "string", "minLength": 1},
        "message": {"type": "string", "minLength": 1}
      },
      "additionalProperties": false
    }
  },
  "additionalProperties": false
}
```

Example: contract check in a test

A contract check should fail with a message that points to the mismatch. Validate both the HTTP status and the payload shape.

```
# Pseudo-code for contract validation

resp = http_client.post("/v1/payments/charge", json={"amount": 10, "currency": "USD"})

assert resp.status_code == 200
payload = resp.json()
validate_json_schema(payload, success_schema)

# For an error case
resp2 = http_client.post("/v1/payments/charge", json={"amount": 0, "currency": "USD"})
assert resp2.status_code == 400
validate_json_schema(resp2.json(), error_schema)
```

Designing tests that don't fight each other

To keep mocks and contract checks from duplicating effort, assign responsibilities:

- **Mocks** verify your request construction and your response handling logic.
- **Contract checks** verify the payload shape you depend on.

A common pattern is:

1. Unit tests validate transformations.
2. Integration tests use mocks to validate wiring and control flow.
3. Contract tests validate payload compatibility using schemas.

When a contract test fails, treat it as a compatibility issue first. When an integration test fails but contract tests pass, treat it as a bug in your client logic.

A cohesive example workflow

Imagine you're adding a new field `status` to your domain model.

- Update your consumer code to parse `status`.
- Update your mock responses to include `status`.
- Add a contract schema rule that `status` must exist and be one of the allowed values.
- Run integration tests: they confirm your parsing and mapping.
- Run contract checks: they confirm the provider's response still matches your expectations.

This approach keeps failures actionable. You'll know whether the issue is your code, your assumptions, or the other side's payload.

Checklist: what to include in mocks and contract checks

- **Mocks**
  - Match on method, path, and required headers
  - Return realistic success and error payloads
  - Simulate at least one failure mode relevant to your logic (retry, timeout, validation)
  - Assert request invariants that would break integration (required fields, correct URL)
- **Contract checks**
  - Validate required fields and allowed values
  - Validate error object nesting and required fields
  - Validate that success and error payloads are mutually exclusive (if your code assumes that)
  - Validate `additionalProperties` rules if you rely on strictness

When you do this, your integration tests stop being "did it run" checks and become "did it match the contract we both agreed to" checks—without needing a live dependency for every run.

11. Managing Dependencies and Build Systems

11.1 Updating Dependencies Without Breaking Builds

Dependency updates are one of those tasks that look simple until they aren't. The goal is not to "update everything," but to change as little as necessary while still moving forward. A safe update process has three parts: control the change, prove it works, and keep the evidence.

The core mindset: small, testable changes

Before touching versions, decide what "breaking" means for your project. For example:

- **Build breaks**: compilation fails, bundling fails, tests don't run.
- **Behavior breaks**: tests pass but outputs differ, APIs behave differently, or performance regresses.
- **Operational breaks**: runtime errors appear only under real configuration.

A practical rule: update **one dependency group at a time** (or one direct dependency at a time) and run the full verification steps for each change.

Mind map: the dependency update workflow

Click here to view the mind map: Dependency Update Workflow (Safe Mode)

Step 1: Update with a lockfile and a clear scope

If your project uses a lockfile, treat it as part of the build contract. Updating dependencies without regenerating the lockfile often creates "works on my machine" situations.

Example (Node.js / npm):

- Update a single dependency in `package.json`.
- Regenerate `package-lock.json`.
- Commit both files.

A safe workflow looks like:

1. Change one dependency version.
2. Run `npm install` to refresh the lockfile.
3. Run `npm test` and your build script.

If you use `npm update` broadly, you may accidentally pull in a large transitive change set. Prefer explicit version bumps for direct dependencies.

Step 2: Inspect the diff before running tests

After updating, inspect what actually changed. You're looking for patterns that correlate with breakage:

- **Major version jumps**: often include breaking API changes.
- **Dependency removals**: may indicate dropped features or different defaults.
- **New transitive dependencies**: can alter behavior indirectly.

Example (what to look for in a lockfile diff):

- A library upgraded from `2.x` to `3.x`.
- A transitive package replaced with a different major version.
- A new optional dependency that changes runtime code paths.

If the diff is large, don't skip tests. Large diffs are exactly where "it compiled" stops being a reliable signal.

Step 3: Run verification in the right order

Run checks in a sequence that reduces wasted time:

1. **Typecheck / lint** (catches obvious incompatibilities).
2. **Build** (ensures bundling and compilation succeed).

3. **Unit tests** (fast feedback on logic changes).
4. **Integration tests** (catches wiring issues and configuration problems).

Example (generic CI-like command order):

- `npm run lint`
- `npm run typecheck`
- `npm run build`
- `npm test`

If you only run `npm test` and compilation fails, you lose time. If you only run `npm run build`, you might miss behavior changes that tests would catch.

Step 4: Handle common breakage patterns with targeted fixes

When something breaks, resist the urge to "make it green" by loosening constraints immediately. Fix the root cause.

Pattern A: API changes in direct dependencies

Symptoms: compilation errors, missing functions, changed method signatures.

Example: Suppose a library changed `client.getUser(id)` to `client.fetchUser({ id })`.

- Update the call sites.
- Adjust tests that assert the old response shape.

A good prompt to your future self: "What changed in the contract?" If you can name the contract change, the fix is usually straightforward.

Pattern B: Transitive dependency behavior changes

Symptoms: tests fail in unexpected places, or runtime errors appear only in certain flows.

Example: A JSON parser dependency changed how it handles trailing commas.

- Identify which code path parses the input.
- Add a test for the exact input that fails.
- Update parsing logic or configuration.

This is where small, incremental updates help: you can attribute the failure to the most recent change.

Pattern C: Peer dependency or version range conflicts

Symptoms: install warnings, missing modules at runtime, or inconsistent dependency trees.

Example: A plugin requires `react@^18`, but the project uses `react@17`.

- Either upgrade the peer dependency (with its own verification),
- or pin the plugin to a compatible version.

The key is to make the compatibility decision explicit rather than letting the package manager guess.

Step 5: Keep the lockfile committed and consistent

A lockfile ensures the build uses the same resolved versions across environments. If you update dependencies, you must commit the lockfile changes.

Example (Python / pip-tools style):

- Update input requirements.
- Regenerate the compiled requirements file.
- Commit both.

Even if your build system can resolve versions dynamically, reproducibility is what prevents "it worked in CI" surprises.

Step 6: Use constraints to prevent accidental large jumps

Sometimes you want updates, but not uncontrolled ones. Version constraints can limit how far a dependency moves.

Example (conceptual):

- If you allow `^2.3.0`, you may get minor/patch updates automatically.
- If you allow `^3.0.0`, you may pull in breaking changes when you didn't intend to.

A practical approach:

- Allow safe ranges for patch/minor updates.
- Treat major upgrades as deliberate events with full verification.

Step 7: Document the change in the PR notes

Documentation here is not a wall of text. It's a short record of what you changed and how you verified it.

Include:

- Which dependencies were updated.
- Whether any major versions changed.
- The commands you ran (build, tests, lint).
- Any follow-up adjustments (code changes, test updates).

This makes future updates faster because you can reuse the same verification pattern.

A concrete mini-checklist you can reuse

- [ ] Update one direct dependency (or one small batch).
- [ ] Regenerate and commit the lockfile.
- [ ] Inspect the diff for major jumps and large transitive changes.
- [ ] Run lint/typecheck → build → unit tests → integration tests.
- [ ] Fix root causes (API changes, parsing behavior, peer conflicts).
- [ ] Add/adjust tests when behavior changes.
- [ ] Record what changed and what you ran.

When you follow this sequence, dependency updates become a controlled engineering task instead of a guessing game. The build stays trustworthy, and failures become explainable rather than mysterious.

11.2 Prompting AI to Explain Build Failures From Logs

Build failures are usually not mysterious. They're just loud. Your job is to turn the noise in the logs into a short, testable diagnosis. The AI can help, but only if you feed it the right slices of evidence and ask for the right kind of explanation.

What to collect before prompting

Start by extracting three things from your build output:

1. **The first failure point**: the earliest line that indicates an error (often "error:", "failed", "cannot find", "undefined reference").
2. **The immediate context**: 30–80 lines around that first failure, including any "Caused by" blocks.
3. **The build command and environment**: the exact command you ran (or the CI step), plus key versions (compiler/runtime) if shown.

A common mistake is pasting the entire log. AI can summarize, but it can also miss the real trigger if the failure is buried under thousands of lines.

Mind map: log-to-diagnosis workflow

Build Failure Explanation Mind Map

Click here to view the mind map: Goal: Identify root cause and next action

Prompting strategy that works

Use a prompt structure that forces the assistant to (a) quote the relevant log lines, (b) explain the mechanism, and (c) propose a small set of next steps.

Template prompt (copy/paste):

```
You are a build engineer. Explain this build failure using only the log excerpt I provide.

1) Quote the 3–8 most relevant lines from the excerpt.
2) In plain English, explain what those lines mean and why the build stopped.
3) Classify the failure (compile/link/dependency/script/environment).
4) List 3 likely root causes, ordered by probability.
5) For each root cause, give one concrete next check or command I can run.
6) Keep it grounded: do not guess beyond what the log suggests.

Log excerpt:
<<<PASTE HERE>>>

Build command:
<<<PASTE HERE>>>
```

The "quote the relevant lines" instruction prevents the assistant from hand-waving. The "ordered by probability" instruction keeps the output actionable.

Example 1: Missing dependency during compilation

Log excerpt (example):

```
error: package org.example.util does not exist
import org.example.util.StringUtils;
       ^
1 error
```

Prompt you can use:

```
Explain this failure. Quote the relevant lines and classify it.
Then propose likely causes and the next check for each.

Log excerpt:
<<<error: package org.example.util does not exist
import org.example.util.StringUtils;
       ^
1 error>>>

Build command:
<<<mvn -q -DskipTests package>>>
```

What a good AI answer should include:

- The assistant should connect the missing package to dependency resolution or module configuration.
- It should suggest checks like verifying the dependency is declared, the group/artifact coordinates match, and the repository is available.

Concrete next checks (examples of what to ask for):

- "Which dependency provides `org.example.util`? Check `pom.xml` / `build.gradle` for the correct coordinates."
- "Run the build with dependency insight (or print the resolved dependency tree) to confirm the artifact is present."

Example 2: Linking error due to missing symbol

Log excerpt (example):

```
/usr/bin/ld: undefined reference to `crypto_sign'
collect2: error: ld returned 1 exit status
```

Prompt:

```
Explain this linking failure. Quote the relevant lines.
Classify it as compile/link/dependency/script/environment.
Give 3 likely root causes and one next check for each.

Log excerpt:
<<</usr/bin/ld: undefined reference to `crypto_sign'
collect2: error: ld returned 1 exit status>>>

Build command:
<<<gcc -O2 main.c -o app>>>
```

Reasoning the assistant should show (without theatrics):

- Undefined reference usually means the compiler saw a declaration, but the linker couldn't find the implementation in the linked objects/libraries.
- The next checks should focus on whether the correct library is linked and whether the symbol name matches (C vs C++, name mangling, version mismatch).

Concrete next checks:

- "Confirm the link command includes the library that defines `crypto_sign`."
- "If using C++: check for missing `extern "C"` around the function declaration."
- "If using a versioned library: verify the installed library version exports that symbol."

Example 3: Build script failure (not a code error)

Log excerpt (example):

```
> node scripts/build-assets.js
Error: ENOENT: no such file or directory, open 'dist/config.json'
```

Prompt:

```
Explain why the build script failed. Quote the relevant lines.
Classify the failure type.
Provide likely causes and next checks.

Log excerpt:
<<<> node scripts/build-assets.js
Error: ENOENT: no such file or directory, open 'dist/config.json'>>>

Build command:
<<<npm run build>>>
```

What to look for:

- The assistant should treat this as a filesystem/path issue rather than a compilation problem.
- The next checks should focus on build order (is `dist/` created earlier?), working directory, and whether the file is generated or expected to be committed.

Concrete next checks:

- "Verify the script's working directory and the relative path to `dist/config.json`."
- "Check whether another step generates `dist/config.json` and whether it runs before this script in CI."

Mind map: common failure categories and what to ask

Click here to view the mind map: Failure Categories -> What to Ask

A practical "tightening" loop

If the first AI response is too broad, tighten the prompt with one extra constraint:

- Ask it to **only use lines between X and Y**.
- Ask it to **output a single most likely root cause** first, then alternatives.
- Ask it to **suggest one command** that would confirm or refute the top cause.

Example follow-up prompt:

```
Your answer lists multiple causes. Pick the single most likely one.
Cite the exact log lines that justify it.
Then propose one command or check that would confirm it in under 2 minutes.
```

Quick checklist for your final diagnosis

Before you act on the explanation, verify that the proposed fix matches the failure type:

- If the log says **compile** errors, don't start by changing link flags.
- If the log says **ENOENT** or **permission denied**, don't rewrite code; fix paths, order, or environment.
- If the log says **undefined reference**, focus on linking inputs and symbol compatibility.

When you prompt this way, the AI becomes a structured reader of your logs, not a guesser of what you meant. That's the whole trick: evidence in, grounded explanation out.

11.3 Optimizing Build Performance With Caching and Targets

Build time is often "death by a thousand rebuilds." The fix is usually not one magic switch, but a set of habits: cache what is safe to reuse, and build only what you actually need. This section focuses on two levers—caching and targets—and shows how to combine them without turning your build into a guessing game.

The core idea: reuse work, not results

Caching works when the build system can prove that an output depends only on inputs you already have. If your build reads environment variables, downloads files, or depends on "whatever is on disk," you must either model those inputs explicitly or avoid caching those steps.

Targets work by narrowing the build graph. Instead of "build everything," you ask for "build this," and the system rebuilds only the prerequisites that are out of date.

Mind map: caching and targets

Click here to view the mind map: Build Performance: Caching + Targets

Caching: make reuse predictable

1) Cache the expensive, stable steps

Typical candidates:

- **Compilation outputs** (object files, intermediate artifacts).
- **Dependency resolution** (downloaded packages, lockfile parsing).
- **Code generation** (protobuf, OpenAPI clients) when inputs are explicit.

Example: suppose your build generates `client.ts` from `openapi.yaml`. If the generator output depends on the generator version and the YAML content, your cache key must include both. Otherwise, you'll reuse stale generated code and chase bugs that look like "random failures."

2) Build cache keys from inputs, not vibes

A good cache key includes:

- Source content (or a hash of it)
- Compiler/tool versions
- Build flags that affect output

- Any configuration files that influence the build

Example: if you toggle `-O2` vs `-O0`, the object files are not interchangeable. Your cache should treat them as different.

3) Avoid hidden inputs

Common hidden inputs:

- Absolute paths embedded into generated code
- Build steps that read files without declaring them
- Commands that depend on current working directory
- Environment variables that change behavior

Concrete example: a code generator writes a header like `// built at /home/user/project/...`. Even if the content is otherwise identical, the output changes and caching becomes ineffective. Prefer relative paths or omit path data from generated outputs.

4) Know when to invalidate

Sometimes you must clear caches, but do it intentionally:

- Toolchain upgrade (compiler, linker, generator)
- Major build script changes
- Switching build modes (debug vs release)

If your build system supports "cache busting" via a versioned key, use it. If not, clear the relevant cache directories rather than wiping everything blindly.

Targets: build less, faster

Targets are the "ask" you give the build system. The build system then computes the dependency graph and rebuilds only what's required.

1) Use target granularity that matches your intent

If you always run the top-level target (like `build`), you'll pay for unrelated work. Prefer:

- `compile` targets when you only changed code
- `test` targets when you changed logic
- `lint` targets when you changed formatting rules

Example workflow:

- You edit `src/auth/login.ts`.
- Run `compile` for the affected module (or the smallest target that includes it).
- Then run `test` for the test suite that covers that module.

This keeps feedback tight and avoids rebuilding assets you didn't touch.

2) Keep target dependencies accurate

If a target lists too many prerequisites, it becomes a rebuild magnet. If it lists too few, you get stale outputs.

Concrete example: imagine a frontend build where `bundle` depends on `styles` and `scripts`. If `bundle` also depends on `images` even when you didn't change them, you'll rebuild images every time. Split targets so `bundle` depends on only what it truly consumes.

3) Prefer "file-level" dependencies when possible

When the build system can track dependencies at the file level, incremental builds improve dramatically.

Example: for TypeScript, if your build tool can track which `.ts` files feed which `.js` outputs, it can recompile only the changed files and their direct dependents. If it recompiles the entire project on any change, caching helps less because you're still doing too much work.

Putting it together: a practical example

Consider a simple monorepo with:

- `packages/api` (server)

- `packages/web` (frontend)
- Shared `packages/common`

Step A: warm caches once

Run a full build for the first time after setting up the repo or after a toolchain change. This populates:

- dependency caches
- compilation caches
- generated artifacts

Step B: iterate with targets

Now you change only `packages/common`.

- Build only the targets that depend on `common`.
- Then run tests for the impacted packages.

If your build graph is correct, `packages/web` and `packages/api` will rebuild, but unrelated packages won't.

Step C: verify with targeted tests

After compilation, run tests that match the scope of change. If you changed shared logic, run both API and web tests that exercise it. If you changed only a UI component, run the web tests only.

Troubleshooting: when caching doesn't help

If build time doesn't improve after enabling caching:

- **Check cache hit rate**: if it's near zero, your cache keys are too specific or inputs aren't modeled.
- **Look for non-determinism**: timestamps, random IDs, or path-dependent outputs break reuse.
- **Inspect dependency declarations**: missing dependencies cause rebuilds or stale outputs.
- **Confirm target correctness**: if your "small" target still pulls in everything, split it.

A quick sanity test: make a change that should affect only one module, then run the smallest target. If the build system recompiles unrelated modules, your target graph is too broad.

Checklist for this subsection

- Cache outputs only when inputs are explicit and outputs are deterministic.
- Include tool versions and build flags in cache keys.
- Avoid hidden inputs like environment variables and absolute paths.
- Use smaller targets that match your intent (compile vs test vs bundle).
- Keep target dependencies accurate and not overly broad.
- Troubleshoot with cache hit rate, non-determinism checks, and dependency graph inspection.

11.4 Enforcing Reproducible Builds With Lockfiles

Reproducible builds mean that the same source code, built under the same declared conditions, produces the same dependency versions and (as much as practical) the same build outputs. Lockfiles are the mechanism that turns "compatible versions" into "exact versions," so your build stops depending on what happened to be latest last Tuesday.

Why lockfiles matter (beyond "pin versions")

A lockfile records more than version numbers. It captures the resolved dependency graph: which transitive packages were pulled in, which versions they resolved to, and often the integrity hashes used to verify downloads. Without it, two developers can run the same command and end up with different transitive trees, even if both start from the same declared ranges.

A subtle point: reproducibility is not only about *what* versions you get, but also about *how* you get them. Integrity checks and deterministic resolution reduce the chance that a dependency tarball changes while keeping the same version label.

Mind map: reproducible builds with lockfiles

Click here to view the mind map: Goal: same inputs → same dependency graph → same build behavior

Node.js example: npm (package-lock.json)

1. Commit the lockfile.

- `package.json` declares ranges:
  - `"react": "^18.2.0"`
- `package-lock.json` records the resolved versions:
  - `react@18.2.0` (and the exact transitive tree)

2. Install using the lockfile.

- Use `npm ci` in CI and for clean installs.
- `npm ci` refuses to proceed if `package-lock.json` doesn't match `package.json`, which is exactly the kind of "stop early" behavior you want.

Example workflow:

- Developer updates `package.json`.
- Developer runs `npm install` to update `package-lock.json`.
- Developer opens a PR that includes both files.
- CI runs `npm ci` and fails if the lockfile is out of sync.

Node.js example: Yarn (yarn.lock)

Yarn's lockfile plays the same role, but the enforcement mechanism differs by version and configuration. The key idea is consistent: use the command that installs from the lockfile and errors on mismatch.

A practical rule:

- In CI, prefer the "clean install" command that does not attempt to update the lockfile.
- Ensure the CI job uses the same Yarn major version as the repo expects.

If your repo uses a Yarn version manager setting (like a pinned Yarn version), treat that pin as part of reproducibility. A different Yarn version can resolve dependencies differently.

Python example: pip-tools (requirements.in + requirements.txt)

Python's ecosystem often uses `requirements.txt` directly, but reproducibility improves when you separate "what you want" from "what you resolved." A common pattern:

- `requirements.in` contains ranges or high-level specs.
- `requirements.txt` is the compiled, pinned output.

Example:

- `requirements.in`
  - `requests>=2.31`
  - `pydantic>=2.0`
- `requirements.txt` contains exact pins:
  - `requests==2.32.3`
  - `pydantic==2.7.4`

Then your build uses only `requirements.txt`. The compiled file is your lockfile equivalent.

Enforcement rule:

- Only regenerate `requirements.txt` in controlled changes.
- CI installs from `requirements.txt` and never regenerates it.

Java example: Maven and Gradle

Java build tools already have lock-like behavior, but you still need to ensure it's actually enforced.

Maven

Maven uses a combination of `pom.xml` and a local repository cache. For reproducibility, you typically rely on:

- Fixed dependency versions in `pom.xml` (avoid broad ranges).
- A consistent dependency resolution strategy.
- Optional use of a dependency lock mechanism where available.

If you use dependency locking, commit the lock artifacts and configure Maven to use them during builds.

Gradle

Gradle can generate dependency lock files. The enforcement pattern is:

- Generate locks in a controlled environment.
- Commit the lock files.
- Configure Gradle to use them and fail when resolution would change.

The important nuance: caching can make builds fast, but it doesn't guarantee correctness. The lock file is what guarantees the resolved graph.

Repository hygiene checklist

- **Commit lockfiles**: `package-lock.json`, `yarn.lock`, `requirements.txt` (compiled), `poetry.lock`, Gradle/Maven lock artifacts.
- **Avoid "update on install" in CI**: CI should install from the lock, not regenerate it.
- **Pin the package manager version**: reproducibility includes the resolver behavior.
- **Keep platform assumptions explicit**: some dependencies vary by OS/architecture. If your lockfile is platform-specific, ensure CI matches.
- **Treat lockfile changes as meaningful**: require the same review rigor as code changes.

CI enforcement patterns (practical)

A good CI job does three things: clean install, lockfile consistency checks, and deterministic build flags.

Example: Node.js CI steps

```
# Clean install using the lockfile
npm ci

# Optional: verify lockfile matches package.json
npm ls --depth=0

# Build
npm run build
```

```
If `npm ci` fails due to mismatch, you catch drift immediately.

### Example: Python CI steps

```bash
# Install exactly pinned dependencies
python -m pip install --requirement requirements.txt

# Build/test
pytest -q
```
```

```
If someone edits `requirements.in` without regenerating `requirements.txt`, CI will still install the old pins, which is fine for

## Common failure modes (and how lockfiles prevent them)

1) **Lockfile not committed**: developers resolve dependencies locally and CI resolves differently.
2) **CI runs a command that updates the lock**: builds become “whatever the resolver feels like today.”
3) **Lockfile drift**: `package.json` changes but lockfile doesn’t. `npm ci`-style mismatch checks catch this.
4) **Different resolver versions**: two machines use different package manager majors, producing different graphs.
5) **Non-deterministic build outputs**: even with fixed dependencies, builds can differ if the build embeds timestamps or reads fr

## A simple policy that works

- Developers update dependencies by changing the high-level spec files.
- They regenerate the lockfile in the same change.
- CI installs using the lockfile and fails on mismatch.
- CI builds with consistent flags and a clean install.

This policy keeps the “what got installed” question answerable from the repository state, not from memory or guesswork.
```

11.5 Handling Platform Differences With Clear Constraints

Platform differences are rarely about “which code is correct.” They’re about assumptions: path separators, line endings, case sensitivity, time zones, character encodings, shell behavior, and how tools interpret the same command. The goal is to make those assumptions explicit, then constrain the assistant (and your team) to respect them.

Start with a platform checklist (and make it part of the prompt)

Before asking for changes, capture the platform facts that affect behavior. Keep this short and concrete.

- **OS**: Linux, macOS, Windows (and versions if relevant)
- **Shell**: bash/zsh, PowerShell, cmd
- **Filesystem**: case-sensitive vs case-insensitive
- **Line endings**: LF vs CRLF
- **Encoding**: UTF-8 vs something else
- **Runtime**: Node/Python/.NET/Java version
- **Build tool**: make/cmake/gradle/msbuild
- **Environment variables**: naming and availability
- **Networking**: proxy behavior, DNS quirks, TLS settings

When you request code, include this checklist plus the “must not change” list.

Example prompt fragment (copyable):

- Target platforms: Linux (ext4), macOS (APFS case-insensitive), Windows (NTFS)
- Must preserve: public API signatures, existing test names, output format
- Must use: UTF-8 everywhere; no hardcoded absolute paths
- Must handle: path separators and case-insensitive filenames

Use constraints that prevent the most common mistakes

AI assistants often produce code that works on the author’s machine. Constraints stop that.

1. **Paths**

- Constraint: “Use `path` utilities; never concatenate paths with `/`.”
- Example (Node.js):
  - Bad: `const file = dir + "/" + name;`
  - Better: `const file = path.join(dir, name);`

2. **Case sensitivity**

- Constraint: “Treat filenames case-insensitively on Windows/macOS; avoid relying on exact case.”

- Example: if you map filenames to IDs, normalize keys:
  - `key = filename.toLowerCase()` (and document it).

3. **Line endings**

- Constraint: "Preserve existing line endings in generated files, or normalize to LF consistently across platforms."
- Example: in Git, configure `.gitattributes` for consistent behavior, then ask the assistant to avoid manual newline hacks.

4. **Shell commands**

- Constraint: "Provide platform-specific command variants or use a cross-platform tool."
- Example: `rm -rf` is not a Windows command. Prefer:
  - Node-based cleanup scripts, or
  - `rimraf` (if the project already uses it), or
  - conditional scripts in package.json with clear names.

5. **Time and locale**

- Constraint: "Use UTC for timestamps in logs and tests; avoid locale-dependent formatting."
- Example: format dates with explicit locale/timezone settings rather than relying on defaults.

6. **Environment variables**

- Constraint: "Do not assume variables exist; validate and provide clear errors."
- Example: if `API_URL` is required, check it early and fail with a message that includes what was missing.

Ask for "behavioral equivalence," not "same code"

Platform-safe changes focus on outcomes.

Prompt pattern:

- "Implement the same behavior on all target platforms."
- "List platform-specific differences you handled."
- "Add tests that prove equivalence."

This nudges the assistant to explain what it changed and why, which makes review faster.

Mind map: platform differences and constraints

Platform Differences Mind Map (Constraints-First)

Click here to view the mind map: Platform Differences (Constraints-First)

Concrete examples you can reuse

Example 1: Fixing a path bug across OSes (Python)

- Problem: code uses string concatenation for file paths.
- Constraint: "Use `pathlib.Path` and keep behavior identical."

Before:

- `full = folder + "/" + filename`

After:

- `full = Path(folder) / filename`

Add a test that uses a filename with mixed case and verify lookup behavior matches the chosen normalization strategy.

Example 2: Making a filename lookup consistent (case-insensitive)

- Problem: tests pass on Linux but fail on Windows because the code assumes exact case.
- Constraint: "Normalize keys; document normalization."

Approach:

- When building a map, store keys in a normalized form (e.g., lowercased).
- When reading input, normalize the input the same way.

Test:

- Provide `Report.TXT` and `report.txt` and assert they resolve to the same record.

Example 3: Cross-platform command execution (Node.js)

- Problem: build scripts call `rm` and `cp`.
- Constraint: "Use Node's filesystem APIs or a cross-platform abstraction already in the project."

Instead of shell commands, implement cleanup and copy using `fs/promises` and `path`.

Test:

- Run the script in CI on each OS and assert the expected files exist and old ones are removed.

How to structure the assistant request for platform-safe output

Use a three-part request: context, constraints, and acceptance checks.

Template (short and strict):

- Context: what breaks and on which platform(s)
- Constraints: what must not change, what must be cross-platform
- Acceptance checks: what tests or outputs prove it's fixed

Example request:

- "On Windows, the build fails because the script uses `/` paths and `rm -rf`. Fix it so the build succeeds on Linux/macOS/Windows. Must use `path` utilities and avoid shell-specific commands. Acceptance: add/adjust tests for path joining and update the build script so it runs without OS-specific command assumptions."

Review rubric: what to look for in the generated patch

When you review the assistant's output, scan for these items first.

- **No hardcoded separators**: `/` or `\` used directly in path construction
- **No OS-specific commands**: `rm`, `cp`, `chmod` without a cross-platform wrapper
- **Explicit encoding**: file reads/writes specify UTF-8 when relevant
- **Stable formatting**: timestamps and numbers aren't locale-dependent
- **Normalization is consistent**: case/newline normalization applied in both read and write paths
- **Tests match the claim**: tests cover the normalization logic and the platform-sensitive behavior

If a patch changes behavior, require a short explanation tied to the constraints you provided. That's the difference between "it works on my machine" and "it works everywhere we care about."

12. Documentation and Developer Experience

12.1 Generating Clear Docstrings and Function Comments

Clear docstrings and comments are a contract: they tell the next reader what the code expects, what it guarantees, and what it does when things go wrong. When you write them well, you reduce the number of "what does this do?" questions and the number of times you have to read the whole function to answer them.

What "clear" means in practice

A good docstring answers five questions, in this order:

1. **What** the function does (one sentence).
2. **Inputs** it expects (types, shapes, units, and constraints).
3. **Outputs** it returns (types and meaning).
4. **Side effects** (I/O, mutation, logging, network calls).

5. **Failure behavior** (exceptions, error values, and conditions).

If you can't answer one of these, that's a sign the function itself needs clarification or a small refactor.

Docstring structure that works across languages

Use a consistent structure so readers can scan quickly.

- **Summary line:** short, imperative or descriptive.
- **Extended description:** only if the summary needs context.
- **Parameters/Arguments:** each item gets a one-line meaning and constraints.
- **Returns:** what the return value represents.
- **Raises/Errors:** what triggers failure.
- **Notes:** edge cases that aren't obvious from the code.

Mind map: docstring components

Click here to view the mind map: Docstrings & Comments

Examples: good docstrings vs. vague ones

Example 1: Python function

Vague docstring (not enough information):

```
def parse_user(s):
    """Parse user."""
    ...
```

Clear docstring (answers the five questions):

```
def parse_user(s: str) -> dict:
    """Parse a user record from a single line.

    The input must be a comma-separated string in the format:
    "id,email,role". Whitespace around fields is allowed.

    Args:
        s: A non-empty line containing exactly three comma-separated fields.

    Returns:
        A dict with keys: "id" (int), "email" (str), and "role" (str).

    Raises:
        ValueError: If the line is empty, has the wrong number of fields,
            or if "id" cannot be converted to an integer.

    Side effects:
        None.
    """
    ...
```

Notice how the docstring states the **format**, the **field count**, and the **failure conditions**. That's what prevents misuses like passing JSON or assuming missing fields are tolerated.

Example 2: JavaScript function

```javascript
/**
 * Compute the average of a list of numbers.
 *
 * @param {number[]} values - Non-empty array of finite numbers.
 * @returns {number} The arithmetic mean.
 * @throws {TypeError} If values is not an array.
 * @throws {RangeError} If values is empty.
 */
function average(values) {
  if (!Array.isArray(values)) throw new TypeError('values must be an array');
  if (values.length === 0) throw new RangeError('values must be non-empty');
  // ...
}
```

This docstring is short but still precise about constraints and errors.

Comments: when they help and when they don't

Comments are most useful for explaining **why** something is done, not **what** is done.

Good comment targets:

- **Non-obvious decisions:** "We use X because Y."
- **Edge cases:** "This branch handles empty input."
- **Invariants:** "At this point, index i is always within bounds."
- **Tradeoffs:** "We accept O(n log n) here because n is small."

Avoid comments that restate the code line-by-line. If the code is readable, the comment should add reasoning.

Example: comment that adds reasoning

```python
# We sort by timestamp before grouping so each group is contiguous.
# Without sorting, the same user could appear in multiple groups.
rows.sort(key=lambda r: r.timestamp)
```

Example: comment that repeats the code

```python
# Increment i
i += 1
```

The second comment doesn't add information; the line already communicates the action.

Making docstrings match reality

A docstring that lies is worse than no docstring. Keep it aligned with behavior by following a simple workflow:

1. Write the docstring based on the intended contract.
2. Implement the function.
3. Run tests that cover normal and failure cases.
4. Adjust the docstring to reflect what the code actually does.

If you change behavior later, update the docstring first, then adjust tests.

Handling tricky parts: units, formats, and constraints

Many bugs come from mismatched assumptions. Docstrings should state these explicitly.

Example: units and time formats

```python
def seconds_to_millis(seconds: float) -> int:
    """Convert seconds to whole milliseconds.

    Args:
        seconds: Duration in seconds. Must be >= 0.

    Returns:
        Milliseconds rounded down to an integer.

    Raises:
        ValueError: If seconds is negative.
    """
    if seconds < 0:
        raise ValueError('seconds must be >= 0')
    return int(seconds * 1000)
```

The “rounded down” detail prevents surprises for callers who expect rounding to nearest.

A practical template you can reuse

Use a template that matches your language’s conventions.

```
Summary: <one sentence what it does>

Args:
  <name>: <type/meaning/constraints>

Returns:
  <type>: <meaning>

Raises/Errors:
  <exception/error>: <trigger condition>

Side effects:
  <none or list>

Notes:
  <edge cases or invariants>
```

Mind map: writing comments that earn their keep

Click here to view the mind map: Comments

Quick self-check before you commit

Before saving, verify:

- The summary matches the function’s actual behavior.
- Every parameter has meaning and constraints.
- The return value is described in terms of what it represents.
- Failure behavior is explicit.
- Any comment that remains adds reasoning, not repetition.

When these checks pass, your docstrings become reliable guides rather than decorative text.

12.2 Writing Usage Examples and Readme Sections

A README is a contract: it tells a reader what the project does, how to run it, and what to expect when things go right or wrong. Usage examples are the part of that contract that people actually try. If your examples are precise, readers spend less time guessing and more time building.

What a strong README section includes

A good usage section answers these questions in order:

1. **What problem does this solve?** One or two sentences are enough.
2. **What do I need to run it?** List prerequisites and versions.
3. **How do I run it?** Provide a minimal "happy path" command.
4. **What inputs does it accept?** Show required flags or environment variables.
5. **What output should I see?** Include a short snippet or description.
6. **How do I troubleshoot common failures?** Mention the top two or three errors.

A practical rule: every command in the README should be copy-pastable, and every example should include the exact working directory assumptions.

Mind map: README usage structure

Click here to view the mind map: Usage & README Examples

Writing usage examples that don't waste time

Usage examples should be small, specific, and honest about constraints.

Example 1: Minimal run (happy path)

Use a single command that works from a clean checkout.

```
# From the project root
npm ci
npm run dev -- --port 3000
```

Then show what "success" looks like.

```
Server listening on http://localhost:3000
GET /health -> 200 {"status":"ok"}
```

Why this helps: readers can confirm the app is alive before they start changing anything.

Example 2: One feature, one example

If the project has a main feature, demonstrate it with the smallest meaningful input.

```
curl -sS -X POST http://localhost:3000/api/convert \
  -H 'Content-Type: application/json' \
  -d '{"text":"Hello"}'
```

Expected output:

```
{"result":"HELLO","mode":"uppercase"}
```

Why this helps: it ties the example to a concrete behavior, not a vague description.

Example 3: Configuration via environment variables

Show the exact variable names and include a "what happens if you omit it" note.

```
export APP_MODE=uppercase
export APP_TIMEOUT_MS=5000
npm run dev -- --port 3000
```

Add a short note in the README:

> If `APP_MODE` is not set, the service defaults to `lowercase`.

Why this helps: readers learn defaults without hunting through code.

Readme section template (copyable)

Use this structure for the “Usage” portion of your README.

```
Usage

Prerequisites
- Runtime: Node.js >= 20
- Package manager: npm
- Environment variables: see Configuration

Quickstart
1. Install dependencies:
   ```bash
   npm ci
```

2. Start the service:

   ```
   npm run dev -- --port 3000
   ```

3. Verify:

   ```
   curl -sS http://localhost:3000/health
   ```

Configuration

- `APP_MODE` (default: `lowercase`): controls output casing
- `APP_TIMEOUT_MS` (default: `3000`): request timeout in milliseconds

Examples

Convert text

```
curl -sS -X POST http://localhost:3000/api/convert \
  -H 'Content-Type: application/json' \
  -d '{"text":"Hello"}'
```

Expected response:

```
{"result":"hello","mode":"lowercase"}
```

Troubleshooting

- **Server won’t start**: check port conflicts; try `--port 3001`.
- **Requests time out**: increase `APP_TIMEOUT_MS` and verify network access.
```

````
### Troubleshooting examples that are actually useful

Troubleshooting works best when each entry follows a consistent pattern:

- **Symptom** (exact error message or behavior)
- **Most likely cause** (one sentence)
- **Fix** (one or two steps)

Example:

> **Symptom:** `EADDRINUSE: address already in use` when starting the server.
> **Cause:** Port is already taken by another process.
> **Fix:** Stop the other process or start with `npm run dev -- --port 3001`.

This avoids generic advice like "check your configuration." Readers can act immediately.

### Showing outputs and artifacts

If your tool creates files, mention where they go and what they're called.

Example:

> Running `npm run build` writes artifacts to `dist/`. The entry point is `dist/index.js`.

If your tool returns values, include a short example response. If it logs, show one or two representative lines.

### Keeping examples aligned with the code

Examples drift when they're written once and never updated. A simple maintenance habit helps:

- Treat each README command as something you can run in a fresh environment.
- If you change a flag name or endpoint path, update the README example in the same commit.

A small "sanity check" mindset prevents the most common failure mode: documentation that looks right but doesn't run.

### Mind map: example quality checklist

```markdown
mindmap
  root((Usage Example Checklist))
    Copy-pastable
      Exact commands
      Correct working directory
    Minimal
      One feature per example
      Small inputs
    Specific outputs
      Expected response snippet
      Where files are written
    Configuration clarity
      Env var names
      Defaults stated
    Troubleshooting
      Symptom -> cause -> fix
      Top two issues
    Consistency
      Same endpoints/flags as code
      Updated in same change
```
````

A complete mini-usage section example

Below is a compact usage section that demonstrates the patterns above.

```
Usage

Quickstart
1. Install:
   ```bash
   npm ci
```

2. Start:

```
   npm run dev -- --port 3000
```

3. Verify health:

```
   curl -sS http://localhost:3000/health
```

Expected:

```
{"status":"ok"}
```

Convert text

Set casing mode:

```
export APP_MODE=uppercase
```

Call the endpoint:

```
curl -sS -X POST http://localhost:3000/api/convert \
  -H 'Content-Type: application/json' \
  -d '{"text":"Hello"}'
```

Expected:

```
{"result":"HELLO","mode":"uppercase"}
```

Troubleshooting

- `ECONNREFUSED`: server isn't running; start it and retry.
- `400 Bad Request`: request body is missing `text`; send `{ "text": "..." }`.

```
When readers can run the commands immediately and recognize the expected outputs, the README becomes a tool rather than a wall of
```

12.3 Creating Runbooks for Debugging and Operations

A runbook is a written procedure for handling a specific kind of problem: what to check first, what evidence to collect, and what to do next. The goal is not to make every incident feel "handled," but to make the next person's first hour more predictable than the last person's.

What a good runbook contains
```

1. **Scope and triggers**
    - What situations it covers (e.g., “API returns 500s after a deploy” or “queue lag exceeds 5 minutes”).
    - What it does *not* cover (e.g., “database corruption” or “auth provider outage”).
2. **Safety and rollback rules**
    - The first action if the system is degraded (often: stop the bleeding, then investigate).
    - Clear rollback criteria (e.g., “rollback if error rate stays above 2% for 10 minutes”).
3. **Evidence checklist**
    - Links to the exact dashboards/log queries you use, plus what you expect to see.
    - A short list of “must capture” artifacts: request IDs, timestamps, affected endpoints, and sample payloads.
4. **Step-by-step procedure**
    - Ordered steps that reduce branching. Each step should end with a decision: “If X, do Y; otherwise go to Z.”
5. **Common causes and targeted tests**
    - A few likely culprits, each paired with a quick confirmation test.
6. **Escalation and handoff**
    - Who to involve when (and what to send them).
    - A handoff template so the next team doesn’t start from scratch.
7. **Post-incident notes**
    - A section for what you learned, what you changed, and what you’d do differently next time.

Mind map: runbook structure

Click here to view the mind map: Runbook (Debugging + Operations)

Writing a runbook that people actually follow

Runbooks fail when they read like a checklist of everything you know. Instead, write them like a sequence of questions.

A useful pattern is: **Confirm** → **Isolate** → **Explain** → **Fix** → **Verify**.

- **Confirm**: prove the problem exists and define its boundaries.
- **Isolate**: narrow to service, dependency, or data path.
- **Explain**: identify the most likely mechanism behind the symptom.
- **Fix**: apply the smallest change that addresses the mechanism.
- **Verify**: show that the system is healthy in the same way it was unhealthy.

Concrete example: runbook for “API 500s after deploy”

Title: API 500s after deploy

Scope: HTTP 500 responses on `/checkout` and `/refund` within 30 minutes of a deployment.

Safety:

- If error rate exceeds **2%** for **10 minutes**, rollback the last deployment.
- If the issue is widespread across endpoints, stop further deploys and notify the on-call lead.

Evidence checklist (capture before changing anything):

- Deployment version and timestamp.
- Error rate graph screenshot (or export).
- Sample request IDs from logs.
- Top 10 stack traces for 500s.

Procedure:

1. **Confirm symptom**
   - Check the error-rate dashboard for the affected endpoints.
   - Decision: if 500s are not limited to the endpoints above, treat as broader incident and follow the “Service-wide errors” runbook.
2. **Isolate to code vs dependency**
   - Compare latency and error rate for dependencies (e.g., payment provider calls).
   - Decision: if dependency timeouts spike at the same time, focus on dependency behavior; otherwise focus on application code paths.
3. **Explain using logs**
   - Filter logs by request IDs and look for the first exception in the request chain.
   - Decision: if the first exception is a deserialization/validation error, check schema changes and request parsing.
   - Decision: if the first exception is a null reference or missing field, check recent model changes and feature flags.
4. **Fix with the smallest safe action**
   - If a schema mismatch is suspected, temporarily enable backward-compatible parsing (feature flag) rather than rewriting the whole endpoint.
   - If a code regression is suspected, rollback.
5. **Verify recovery**
   - Confirm error rate drops below **0.2%** for **5 minutes**.
   - Confirm successful responses include expected fields (spot-check logs for response payload shape).

Escalation handoff template:

- What happened (symptom + time window)
- What changed (deployment version)
- Evidence (top stack traces + request IDs)
- Current hypothesis (one sentence)
- Actions taken (feature flags, rollback status)

Concrete example: runbook for “queue lag rising”

Title: Queue lag rising

Scope: Consumer falls behind; backlog grows for `orders-processor` queue.

Safety:

- If backlog exceeds **N** and processing time per message increases, pause non-critical producers to reduce load.

Evidence checklist:

- Backlog size over time.
- Consumer processing time distribution.
- Error rate for message handling.
- Dead-letter queue growth.

Procedure:

1. **Confirm** backlog growth and whether it’s tied to a deployment.
2. **Isolate** whether lag is due to throughput (slow processing) or intake (more messages than usual).
3. **Explain** by checking for increased retries and dead-letter entries.
4. **Fix**
   - If retries spike due to a specific validation error, add a targeted guardrail to skip/mark bad messages and prevent retry storms.
   - If processing time increases without errors, profile the hot path by sampling traces from a small time window.
5. **Verify** backlog slope turns downward and retry rate stabilizes.

Mind map: decision points inside a runbook

Click here to view the mind map: Decision points

Runbook writing tips that prevent common mistakes

- **Use numbers where possible.** “Rollback if error rate stays high” becomes actionable when you specify thresholds and time windows.

- **Prefer one hypothesis at a time.** A runbook should guide the reader toward a single working explanation, not a list of unrelated possibilities.
- **Make verification match the symptom.** If the problem is "500s," verify "500s" return to normal, not just "CPU looks fine."
- **Include a handoff template.** Most delays happen when the next person lacks the evidence you already collected.

A short runbook template you can copy

Runbook: [Problem name]

Scope

- Triggers:
- Out of scope:

Safety

- Immediate containment:
- Rollback criteria:

Evidence to capture

- Time window:
- Metrics:
- Logs/IDs:

Procedure (Confirm → Isolate → Explain → Fix → Verify)

1. Confirm:
2. Isolate:
3. Explain:
4. Fix:
5. Verify:

Escalation

- When to escalate:
- What to send:

Post-incident notes

- What changed:
- Root cause hypothesis:
- Preventive actions:

A runbook is successful when it reduces uncertainty for the next person. If you can read your runbook and predict what evidence will be collected and what decision will be made at each step, you've written something worth keeping.

12.4 Documenting APIs and Error Contracts

Good API documentation does two jobs at once: it tells people what success looks like, and it makes failure predictable. Error contracts are the part that prevents "it crashed" from becoming "it crashed in a way we can't reproduce."

What an error contract includes

An error contract is a written agreement between the API and its callers. For each endpoint (and often for each operation), document:

- **HTTP status code rules**: which codes are used for which categories of problems.
- **Error response schema**: the exact JSON shape (field names, types, required vs optional fields).
- **Error codes**: stable identifiers that don't change when wording changes.
- **Human-readable message policy**: whether messages are for end users, developers, or logs.
- **Correlation and trace fields**: how callers can connect an error to logs (for example, `requestId`).

- **Retry guidance**: whether the client should retry, and under what conditions.
- **Validation details**: how field-level errors are represented.

A useful rule of thumb: if a developer can't write a correct `switch` statement over error codes, the contract is missing something.

Mind map: API documentation and error contracts

Click here to view the mind map: API Documentation

A practical error response schema

Pick one schema and reuse it across the API. Here's a common pattern that stays readable and supports automation.

Error response (example schema)

- `error`: object
  - `code` (string, required): stable error identifier like `VALIDATION_FAILED`.
  - `message` (string, required): short description suitable for logs.
  - `requestId` (string, required): correlation token.
  - `details` (object, optional): structured extra information.
  - `retryable` (boolean, optional): whether the client should retry.

Field-level validation details

- `details.validationErrors` (array, optional)
  - each item has `field` (string), `issue` (string), and `constraint` (string, optional).

This structure makes it easy to:

- show a friendly message in the UI using `message` or a separate mapping layer,
- highlight specific fields using `validationErrors`,
- decide retries using `retryable`.

Mind map: error contract fields

Click here to view the mind map: Error Contract Fields

Documenting status code mapping

Status codes should be predictable. A typical mapping looks like this:

- **400 Bad Request**: malformed input or failed validation.
- **401 Unauthorized**: missing/invalid authentication.
- **403 Forbidden**: authenticated but not allowed.
- **404 Not Found**: resource doesn't exist (or is intentionally hidden).
- **409 Conflict**: request conflicts with current state.
- **422 Unprocessable Entity**: semantic validation failures (optional; use consistently).
- **429 Too Many Requests**: rate limiting.
- **500 Internal Server Error**: unexpected server failure.
- **503 Service Unavailable**: temporary inability to process.

If you use both `400` and `422`, document the difference in one sentence per endpoint category. Otherwise, pick one and stick to it.

Examples that teach the contract

Examples should be concrete and minimal. Show the request context and the exact response body.

Example: validation error (400)

Request:

- `POST /v1/users`

- Body: `{ "email": "not-an-email" }`

Response:

```
{
  "error": {
    "code": "VALIDATION_FAILED",
    "message": "Request body failed validation.",
    "requestId": "req_9f3a2c",
    "details": {
      "validationErrors": [
        {
          "field": "email",
          "issue": "must be a valid email address",
          "constraint": "email_format"
        }
      ]
    },
    "retryable": false
  }
}
```

Documentation notes to include:

- `retryable` is `false` for validation failures.
- `validationErrors` is an array even if there's only one issue.
- `field` uses the request's field name, not a display label.

Example: conflict error (409)

Request:

- `POST /v1/subscriptions`
- Body: `{ "userId": "u1", "planId": "p1" }`

Response:

```
{
  "error": {
    "code": "SUBSCRIPTION_ALREADY_EXISTS",
    "message": "User already has an active subscription for this plan.",
    "requestId": "req_2a10bd",
    "details": {
      "existingSubscriptionId": "sub_77"
    },
    "retryable": false
  }
}
```

Documentation notes to include:

- `SUBSCRIPTION_ALREADY_EXISTS` is stable and can be handled without parsing `message`.
- `existingSubscriptionId` is present when the conflict is resolvable by referencing the existing resource.

Example: authorization error (403)

Response:

```
{
  "error": {
    "code": "FORBIDDEN",
    "message": "You do not have permission to perform this action.",
    "requestId": "req_1c0e5a",
    "retryable": false
  }
}
```

Documentation notes to include:

- Avoid leaking whether a resource exists when that would create an information disclosure.
- Keep `details` empty unless you have a safe, consistent reason to add it.

Documenting retry behavior without guesswork

If clients should retry, say so explicitly. Use `retryable` and document the conditions.

Example documentation line:

- "For `code: RATE_LIMITED`, clients may retry after the time indicated by `Retry-After` header."

If you don't include `Retry-After`, don't imply it. The contract should match the actual response.

Consistency rules that prevent documentation drift

- **One error schema per API version**: changing field names breaks clients.
- **Stable error codes**: messages can change; codes should not.
- **Required fields stay required**: if `requestId` is always present, keep it that way.
- **Validation format is uniform**: every endpoint uses the same `validationErrors` structure.

Mind map: where to place error contract info

Click here to view the mind map: Endpoint Page

A compact template you can reuse

Use a consistent layout for every endpoint.

Click here to view the mind map: Errors

Final checklist for error contract documentation

- Every documented error includes a **status code**, **error code**, and **example body**.
- The error schema is **identical across endpoints**.
- `requestId` (or equivalent) is **documented as always present**.
- Validation errors are **field-specific** and use a consistent structure.
- Retry guidance is **explicit** and matches actual headers/fields.

When these pieces are in place, callers can handle failures deterministically, and your future self won't have to reverse-engineer what "that one endpoint" does.

12.5 Keeping Documentation in Sync With Code Changes

Documentation goes stale for predictable reasons: the code changes faster than the docs, the docs are updated by someone who didn't touch the code, or the docs are updated but not in the same place the reader expects. Keeping documentation in sync is less about willpower and more about wiring documentation updates into the same workflow that updates code.

Why "in sync" is a measurable goal

"In sync" means a reader can use the documentation to do the same thing the code does today, without guessing. That includes:

- **Behavior**: inputs, outputs, error cases, and side effects match.
- **Contracts**: function signatures, API payload shapes, and status codes match.
- **Constraints**: limits, timeouts, and validation rules match.
- **Examples**: sample requests, commands, and code snippets still run.

A practical trick: treat documentation as another interface. If the interface changes, the docs must change too.

A workflow that forces updates

1. **Co-locate docs with the code they describe** Put the "source of truth" near the implementation. For example, keep API docs next to the route handlers, and keep function-level docs next to the functions.
2. **Use doc ownership rules** Assign responsibility per area. If a module has a maintainer, that maintainer is responsible for doc accuracy when they change behavior.
3. **Update docs in the same commit as the code** If you merge code without docs, you create a gap that will be filled later with guesses. Same-commit updates reduce that risk.
4. **Require a doc check in review** Add a short checklist item to pull requests: "Docs updated for behavior changes." Reviewers should verify that the doc changes correspond to the code changes.
5. **Make examples executable** If an example can be run as a test or a small script, it should be. If it can't, at least validate it with a lightweight check (like a schema validation or a compile step).

Mind map: where documentation can drift

Click here to view the mind map: Keeping Documentation in Sync

Inline documentation: keep it close and specific

Inline docs should answer the questions a developer asks while editing code: what the function expects, what it returns, and what can go wrong.

Example: before and after a behavior change

Before (doc is vague):

- "Parses input and returns a result."

After (doc matches behavior):

- "Accepts a JSON string. Returns `{ok: true, value}` on success. Returns `{ok: false, error}` when parsing fails, without throwing."

The key is to document the *decision points*: whether errors throw or return, whether fields are optional, and what normalization happens.

API documentation: treat payloads and errors as contracts

For APIs, "in sync" means the documented request/response shapes match the actual serialization and error mapping.

Example: documenting an error contract

If your endpoint returns:

- `200` with `{data: ...}` on success
- `400` with `{error: {code, message}}` on validation failures

Then the docs should include:

- the exact `code` values
- which fields are present in the error payload
- whether `message` is stable or meant for humans

A common failure mode is documenting only the happy path. When behavior changes in validation, clients break in ways that look like "random" failures.

Executable examples: documentation that proves itself

Examples drift when they're copied once and never checked. A simple approach is to turn examples into small tests.

Example: doc example as a testable snippet (pseudo-code)

```
Example in docs:
POST /v1/charge
Body: {"amount": 100, "currency": "USD"}
Expect: 201 Created with {"id": "..."}

Doc test:
- Send request with fixed amount/currency
- Assert status code == 201
- Assert response has key "id"
- Assert response.id matches UUID pattern
```

Even if you can't run the full integration test in every environment, you can still validate the example's structure (schema, required keys, and status code expectations).

Doc tests for schemas and outputs

If your API uses schemas, you can validate that the docs' example payloads match the schema.

Example: schema validation for a documented response

- Docs show an example response.
- A test loads the example JSON.
- The test validates it against the response schema.

This catches the classic mismatch: the docs say `snake_case`, the code returns `camelCase`, or a field becomes optional.

Versioning: keep old behavior documented when it matters

When breaking changes happen, "in sync" doesn't mean overwriting everything. It means the documentation you show for a given version matches that version's behavior.

Example: versioned endpoint docs

- `/v1/*` docs describe the v1 payloads and error codes.
- `/v2/*` docs describe the v2 payloads and error codes.

If you only update the latest docs, you create a mismatch for anyone reading older guides or integrating against a pinned version.

A PR checklist that actually works

A checklist should be short enough to use and specific enough to catch drift.

PR Doc Sync Checklist

- [ ] Did this change behavior, inputs, outputs, or error handling?
- [ ] If yes, did I update the relevant inline docs and API docs in the same commit?
- [ ] Did I update any examples affected by the change?
- [ ] Did I add or update a doc test (schema validation or runnable example)?
- [ ] Did I verify the docs match the current code paths (not an older branch)?

Keeping guides and runbooks aligned with reality

Guides and runbooks are often written once and then forgotten. They drift because they include operational details like commands, flags, and environment variables.

Example: runbook step that becomes wrong

- Runbook says: "Set `LOG_LEVEL=debug`."
- Code now reads `LOG_LEVEL` only in one component, and the other uses `APP_LOG_LEVEL`.

To prevent this, treat runbooks like code:

- Keep command examples parameterized.

- Prefer “copy/paste ready” commands that match the current CLI flags.
- Update runbooks when configuration keys or defaults change.

Documentation review: what to look for

When reviewing doc changes, focus on correspondence rather than writing style.

- **Does each documented claim map to a code path?**
- **Are defaults stated where the code has defaults?**
- **Are error cases documented with the same shape the code returns?**
- **Do examples match the current request/response formats?**

A reviewer doesn’t need to be a poet. They need to be a careful reader who checks that the docs describe what the code actually does.

A lightweight “doc diff” habit

Before merging, compare what changed in code to what changed in docs. If the code change is about validation rules, the doc change should mention validation rules. If the code change is about performance, the doc change should mention any new limits or timeouts.

This habit turns documentation from an afterthought into a direct response to the code change, which is the simplest way to keep everything aligned.

13. Advanced Prompting for Complex Tasks

13.1 Decomposing Large Changes Into Safe Steps

Large changes fail in predictable ways: one part is correct, another part is wrong, and the combined result is hard to reason about. The fix is to split the work into steps that each have a clear goal, a small blast radius, and a quick way to confirm success.

The core idea: “one behavior change per step”

A safe step does three things:

1. **Names the behavior** it will change (what the user or caller will observe).
2. **Limits the scope** to the smallest set of files/modules needed.
3. **Defines a check** that proves the step worked (tests, a log assertion, or a targeted run).

If you can’t state the behavior in one sentence, the step is too big.

Mind map: step decomposition workflow

Click here to view the mind map: Decomposing Large Changes Into Safe Steps

Step 0: Write a “behavior contract”

Before prompting an assistant to generate code, write a short contract. It prevents the common failure mode where the assistant “does the right thing” in code but not in behavior.

Example behavior contract (refactor + new feature):

- Current behavior: `POST /orders` validates input and writes an order.
- New behavior: add `discountCode` support.
- Invariants: existing validation rules remain; response schema stays backward compatible.
- Constraints: keep database transaction boundaries unchanged.

This contract becomes the checklist for each step.

Step 1: Create a baseline you can trust

A baseline is not just “tests pass.” It’s also “you know what to measure.”

Practical checks:

- Run the full test suite once.
- Run the specific tests that cover the affected area.
- Capture one or two representative logs or outputs.

Example: If you're changing order creation, run tests for:

- invalid payloads
- successful order creation
- any existing discount-related behavior (even if it's "none")

Step 2: Add tests that describe the new behavior (before code)

When you add tests first, you reduce the chance of implementing the wrong behavior. The assistant can help generate tests, but you still decide what "correct" means.

Example test cases for `discountCode`:

- Valid code applies a discount.
- Invalid code returns a clear validation error.
- Missing code behaves exactly like before.

Prompt pattern:

- "Create unit tests for `applyDiscount(order, discountCode)` covering: valid, invalid, missing. Keep existing behavior unchanged for missing."

Step 3: Introduce scaffolding without changing behavior

Scaffolding is temporary structure that lets you insert the new logic later.

Common scaffolding techniques:

- Add a new function that is currently unused.
- Add a new field to a DTO but ignore it in the handler.
- Add a feature flag that defaults to the old path.

Example (feature flag):

- Add `ENABLE_DISCOUNT_CODES=false`.
- In the handler, route to the old behavior when the flag is false.

This step should not change outputs. Your check is "existing tests still pass and new tests fail in the expected way."

Step 4: Implement the new behavior behind the toggle

Now you can implement the behavior, but keep it isolated.

Example implementation plan:

1. Parse `discountCode` from the request.
2. Validate it using a dedicated function.
3. Apply the discount to the computed totals.
4. Keep response schema stable.

Prompt pattern:

- "Implement discount application only when `ENABLE_DISCOUNT_CODES=true`. Do not modify existing validation rules. Add minimal code changes to the order handler."

Validation:

- Run the new tests with the flag enabled.
- Run the existing tests with the flag disabled.

Step 5: Remove the old path (only after checks are green)

Once the new behavior is proven, you can delete the old code path.

Safety rules for removal:

- Remove one layer at a time (e.g., handler routing first, then helper functions).
- Keep the public interface stable.
- Ensure error messages remain consistent unless you intentionally change them.

Example:

- Replace `if flag then old else new` with `always new`.
- Delete the old helper after tests confirm identical behavior for cases that should match.

Step 6: Clean up and normalize

Cleanup prevents "temporary" code from becoming permanent.

Checklist:

- Remove unused flags or scaffolding.
- Ensure consistent naming and error handling.
- Confirm formatting/lint rules are satisfied.

Example cleanup prompt:

- "After tests pass, remove the feature flag and dead code. Keep function signatures unchanged. Ensure error messages for invalid discount codes match existing validation style."

How to ask the assistant for each step

A large change prompt often produces a large patch. Instead, prompt per step with explicit boundaries.

Use this template:

- **Goal:** what behavior changes.
- **Scope:** which files/functions.
- **Constraints:** what must not change.
- **Output format:** patch/diff, or "explain then implement."
- **Checks:** which tests to run.

Example step prompt (scoped):

- "Goal: add discountCode validation to `validateOrderInput`. Scope: only `order_validation.py` and its tests. Constraints: do not change the handler response shape. Checks: run `test_validateOrderInput_*`."

Common decomposition mistakes (and quick fixes)

1. **Step changes multiple behaviors.**
   - Fix: split by observable outcomes (validation vs persistence vs response formatting).
2. **Step scope is too wide.**
   - Fix: move logic into a new helper and keep the caller unchanged.
3. **No explicit check for success.**
   - Fix: require a test or a targeted run per step.
4. **Tests are added after code.**
   - Fix: write failing tests first, then implement.

Mini case study: adding a new request field safely

Task: Add `clientRequestId` to an endpoint and echo it back.

- **Step A (contract + tests):** Add tests asserting the field is echoed when present and omitted when absent.
- **Step B (scaffolding):** Parse the field but don't include it in the response yet.
- **Step C (new behavior behind toggle):** When enabled, include it in the response.
- **Step D (remove toggle):** Always include it; delete the old routing.
- **Step E (cleanup):** Remove temporary parsing comments and ensure consistent naming.

Each step has a single observable change, so failures point to a specific stage instead of a tangled patch.

A practical “step size” rule

If a step would require more than one page of code review to understand, it’s probably too large. Aim for changes that a reviewer (or future you) can summarize in a few sentences: what changed, where, and how you know it’s correct.

13.2 Requesting Patch Style Outputs and Applying Diffs

When you ask an assistant for “patch style” output, you’re asking for changes that can be applied with minimal interpretation. The goal is simple: produce a small, reviewable set of edits that you can apply safely, then verify with tests.

What “patch style” means in practice

A patch-style response typically includes:

- **A diff** (or unified diff) that shows exactly what lines change.
- **A short rationale** for why the change is needed.
- **Constraints** like “do not refactor unrelated code” or “keep public APIs unchanged.”

If the assistant instead returns a full file, you lose the ability to quickly see what changed. That’s not automatically wrong, but it makes review slower and increases the chance you miss a subtle behavior change.

Mind map: patch workflow

Click here to view the mind map: Patch workflow

Prompting for diffs that are actually usable

Use a prompt structure that forces the assistant to behave like a careful editor.

Prompt checklist (copy/paste friendly):

1. **State the goal**: fix a bug, add a guard, improve performance in a specific function.
2. **Constrain the scope**: list files and functions, or say “only change these files.”
3. **Preserve behavior**: “keep existing public interfaces,” “do not change output format,” “do not reorder fields.”
4. **Request unified diff**: “Return a unified diff only.”
5. **Limit context**: “Include enough lines to apply cleanly, but avoid unrelated surrounding code.”
6. **Ask for a patch summary**: 3–5 bullet points max.

Example prompt (bug fix)

You have a failing test and want a minimal fix.

```
You are editing a Python service.
Goal: Fix failing test `test_parse_handles_empty_string`.
Constraints:
- Only modify `app/parsing.py`.
- Do not change function signatures.
- Preserve behavior for non-empty inputs.
Output:
- Return a unified diff only (no full file).
- Include a short bullet summary after the diff.
Here is the current function and the failing assertion:
[PASTE RELEVANT CODE]
[PASTE TEST FAILURE MESSAGE]
```

Applying diffs safely

Even a good patch can fail to apply cleanly if the code has drifted. Treat patch application like a controlled experiment.

Recommended workflow

1. **Create a branch**.

2. **Apply the diff**.
3. **Resolve conflicts** (if any).
4. **Run targeted tests**.
5. **Commit with a message that matches the patch summary**.

Applying with git

If the assistant returns a unified diff, you can apply it directly.

```
# 1) Save the assistant output to a file
# e.g., patch.diff

# 2) Apply
git apply --verbose patch.diff

# 3) If it fails, inspect and resolve
# then re-run git apply or edit manually.
```

If you want to avoid partially applied changes, use the dry-run first.

```
git apply --check --verbose patch.diff
```

When diffs don't apply cleanly

Common reasons:

- The assistant referenced line numbers that don't match your current file.
- The patch includes too little context.
- The assistant changed whitespace in a way that breaks matching.

Fix strategy:

- Ask the assistant to **increase context** by a small amount (for example, "include 5–10 lines around each change").
- Or ask it to **rebase the patch** against the exact current snippet you provide.

Example follow-up prompt

```
The patch did not apply cleanly.
Please regenerate the unified diff using the exact current contents of `app/parsing.py`.
Include 8 lines of context around each change.
Return only the diff plus a brief summary.
```

Making patches minimal (and reviewable)

A patch should be small enough that a reviewer (you, in five minutes) can answer: "What changed, and why?"

Practical tactics:

- **Avoid formatting-only edits**: ask for "no whitespace-only changes."
- **Avoid broad renames**: ask for "do not rename variables unless required."
- **Prefer local fixes**: ask for "change only the smallest function that addresses the failure."

Example: patch for a guard clause

Suppose you have a function that crashes on empty input.

Current code (snippet):

- `parse_date(s)` calls `s[0]` without checking `s`.

Assistant patch output (illustrative):

```
diff --git a/app/parsing.py b/app/parsing.py
index 1111111..2222222 100644
 a/app/parsing.py
+++ b/app/parsing.py
@@ -12,6 +12,10 @@ def parse_date(s: str) -> datetime:
     """Parse a date string."""
+    if s == "":
+        raise ValueError("date string must not be empty")
+
     first = s[0]
     if first.isdigit():
         return _parse_iso(s)
```

Why this patch is easy to verify:

- It changes only one function.
- It adds a single guard with a clear error message.
- It doesn't alter logic for non-empty strings.

Example: patch that preserves output formatting

If you're optimizing, you still want the same output.

Constraint to include in your prompt:

- "Do not change the returned JSON keys or ordering."
- "Keep the same error messages."

Patch style helps here because you can confirm the diff doesn't touch serialization code.

Mind map: diff quality checks

Click here to view the mind map: Diff quality

A practical "patch request" template

Use this when you want the assistant to behave like it's preparing a pull request, not rewriting your project.

```
Task: [what to change]
Files: [list files]
Constraints:
- Only modify: [specific files]
- Preserve: [public APIs, output format, error messages]
- No refactors outside the fix
Output format:
- Unified diff only
- Include 8 lines of context around each change
- No whitespace-only edits
After the diff: 3-5 bullet summary items
```

Verification after applying

After you apply the patch, don't just run "all tests" by default. Run what proves the change.

- If it's a parsing bug, run the parsing test module.
- If it's a performance tweak, run the benchmark-like test (or the closest deterministic test) plus the unit tests that cover edge cases.

If the patch changes behavior, update or add tests—but keep those changes separate so the diff stays focused. A patch that fixes code and tests at once can be correct, but it's harder to review because you're mixing "what changed" with "how we proved it."

Patch style outputs work best when you treat them as small, precise proposals: apply, verify, and only then expand scope if needed.

13.3 Using Constraints for Performance, Security, and Style

Constraints turn a vague request into something the assistant can't "interpret away." You're not just asking for code; you're specifying what must be true about the code: speed targets, safety rules, formatting, and invariants. The trick is to write constraints that are testable, not just preferences.

Constraint mindset: make requirements executable

A good constraint usually maps to one of these checks:

- **Static checks**: lint rules, type checks, formatting, forbidden APIs.
- **Dynamic checks**: unit tests, property tests, runtime assertions.
- **Measured checks**: benchmarks, timeouts, memory limits.
- **Behavioral checks**: "must not change output," "must preserve error semantics."

When you prompt, include constraints in a predictable order: **scope** → **invariants** → **performance** → **security** → **style** → **output format**.

Mind map: constraint categories and how they connect

Click here to view the mind map: Constraints for Performance, Security, and Style

Prompt template: constraints that actually work

Use a template you can reuse across tasks.

You are editing an existing codebase.

SCOPE:

- File(s): <...>
- Change type: <bugfix/refactor/feature>

INVARIANTS (must remain true):

- <e.g., public function signature unchanged>
- <e.g., error messages preserved for callers>

PERFORMANCE CONSTRAINTS:

- Target: <e.g., handle 100k items>
- Complexity: <e.g., O(n log n) or better>
- No new N+1 queries: <yes/no>
- Include a benchmark or a micro-test for the hot path.

SECURITY CONSTRAINTS:

- Validate inputs: <rules>
- Parameterize queries: <required>
- Do not log secrets: <required>
- Avoid dangerous APIs: <list>

STYLE CONSTRAINTS:

- Follow existing naming and formatting.
- Keep functions under <N> lines.
- Use existing error types.

OUTPUT FORMAT:

- Provide a patch (diff) and the new/updated tests.
- End with a checklist mapping each constraint to what changed.

Example: performance + security constraints in one change

Scenario: You're adding a search endpoint. A naive implementation builds SQL strings and loops over results to compute fields, causing both injection risk and slow performance.

Constraint-driven prompt (JavaScript/TypeScript-ish pseudocode):

- Performance: avoid per-row queries; complexity should be linear in result size.
- Security: parameterize all user inputs; reject overly long query strings.
- Style: keep the handler under 60 lines; reuse existing DB client helpers.

Assistant output should follow constraints like this:

- Use a parameterized query: `WHERE term ILIKE $1` with `$1` bound.
- Add a limit: `LIMIT 50` to prevent unbounded work.
- Compute derived fields in SQL or in a single pass over the returned rows.
- Add tests for: empty term, long term rejection, and that the query uses parameters (often asserted by mocking the DB call).

Concrete code sketch (showing the constraint intent):

```
// Constraints: parameterized query, max term length, no per-row queries.
const MAX_TERM = 80;

export async function searchHandler(req, res) {
  const term = String(req.query.term ?? "");
  if (term.length > MAX_TERM) return res.status(400).json({ error: "term too long" });

  const rows = await db.query(
    "SELECT id, title FROM items WHERE title ILIKE $1 ORDER BY id DESC LIMIT 50",
    [`\%${term}\%`]
  );

  const results = rows.map(r => ({ id: r.id, title: r.title }));
  return res.json({ results });
}
```

Notice what's missing: no string concatenation for SQL, no loop that calls the database again, and a clear bound on work.

Example: style constraints that prevent "cleanup" from breaking behavior

Scenario: You ask for a refactor, but the assistant might change error types or reorder logic. Style constraints can include behavioral guardrails.

Add invariants like:

- "Do not change the error code mapping."
- "Preserve the order of validation checks."
- "Keep the same return shape."

Then add style constraints like:

- "Extract helper functions only if they reduce duplication by at least 2 call sites."
- "No new exported functions."

Prompt snippet:

- INVARIANTS: preserve error semantics.
- STYLE: keep handler under 40 lines; extracted helpers must be private.
- OUTPUT: include tests that assert exact error codes.

This forces the assistant to treat refactoring as a controlled transformation, not a redesign.

Security constraints: write them as "allowed vs forbidden"

Security constraints are easiest when phrased as explicit bans.

Examples of forbidden patterns:

- **No dynamic SQL**: forbid string interpolation into query text.
- **No shell execution with user input**: forbid `exec` / `spawn` with concatenated args.
- **No unsafe deserialization**: forbid `pickle.loads`, `yaml.load` without safe mode, etc.
- **No secret logging**: forbid printing tokens, passwords, or raw headers.

Constraint phrasing that works well:

- "All database calls must use parameter placeholders; do not build query strings."
- "If input is invalid, return 400 with a stable error code; do not include raw input in error messages."

Performance constraints: specify what "fast enough" means

Performance constraints should include either a target size or a measurable threshold.

Good constraint examples:

- "For 100k items, the endpoint must respond within 200ms in the existing benchmark environment."
- "No more than 1 database query per request."
- "Avoid $O(n^2)$ loops over arrays larger than 10k."

If you can't benchmark, use structural constraints:

- "Use a single pass over the data."
- "Precompute maps for lookups instead of repeated `.find()`."

Output discipline: require a constraint checklist

Ask for a short checklist at the end. It's not for decoration; it's a forcing function.

Checklist format example:

- Performance: ✅ single query, ✅ single pass mapping, ✅ limit applied.
- Security: ✅ parameterized SQL, ✅ term length validation, ✅ no secret logging.
- Style: ✅ handler under 60 lines, ✅ reused existing error type, ✅ formatter compliant.

This makes it easier to review whether the assistant actually followed the constraints.

A final mini-mind map: constraint writing checklist

Click here to view the mind map: Constraint Writing Checklist

When constraints are specific and testable, the assistant's output becomes easier to trust and easier to review. You spend less time arguing about intent and more time verifying results.

13.4 Guiding AI to Preserve Existing Behavior

When you ask an assistant to change code, it will often "improve" things you didn't ask it to touch. Preserving existing behavior is mostly about making the boundary between *allowed change* and *forbidden change* explicit, then forcing the assistant to prove it stayed inside the boundary.

The core idea: define the behavior contract

Start by writing a short contract that the assistant must not violate. A behavior contract can include:

- **Inputs**: accepted types, allowed ranges, and what happens on invalid input.
- **Outputs**: return values, error types, and side effects.
- **Invariants**: ordering guarantees, idempotency, and state transitions.
- **Observability**: logs, metrics, and emitted events (if they matter).
- **Performance constraints**: only if they are already part of the system's expectations.

A good contract is short enough to paste into a prompt, but specific enough that a wrong change becomes obvious.

Mind map: "Preserve behavior" checklist

Click here to view the mind map: Preserve Existing Behavior (Prompt Checklist)

Prompt pattern that works: “constrain, then request”

Use a consistent structure:

1. **State the goal** (what you want changed).
2. **State the contract** (what must remain true).
3. **State the boundaries** (what must not change).
4. **Request a minimal patch** (prefer diffs or line-level edits).
5. **Request a verification plan** (what to run and what to compare).

Here’s a template you can adapt.

> Goal: <what you want to change>
>
> Behavior contract (must not change):
>
> - Inputs: <…>
> - Outputs: <…>
> - Side effects: <…>
> - Invariants: <…>
> - Observability: <…>
>
> Change boundaries:
>
> - Allowed: <…>
> - Forbidden: <…>
> - Preserve: function signatures, error types, and existing log messages.
>
> Deliverable:
>
> - Provide a minimal patch (diff-style) only for the allowed edits.
> - List files/functions changed.
> - Explain how each contract item is preserved.
> - Provide exact commands to run tests and a small before/after comparison.

Example 1: Refactor without changing exceptions

Suppose you have a function that validates a request and throws specific exceptions. You want to refactor for readability, but you must preserve exception types and messages because callers catch them.

Existing behavior (contract):

- If `userId` is missing: throw `BadRequestError("userId is required")`.
- If `userId` is non-numeric: throw `BadRequestError("userId must be a number")`.
- Otherwise: return a normalized integer.

Bad prompt (common failure):

> “Refactor this validation function and improve error handling.”

This invites the assistant to change messages, consolidate errors, or introduce a different exception type.

Better prompt (behavior-preserving):

> “Refactor the function for readability. Preserve the exact exception types and messages for the two error cases. Do not change the function signature or return type. Provide a minimal patch and include a short mapping from each old branch to the new code path.”

What to ask the assistant to do:

- Keep the same branching logic structure, even if reorganized.
- If it introduces helper functions, ensure they still throw the same exceptions with the same messages.
- Avoid “helpful” changes like returning `null` instead of throwing.

Example 2: Preserve ordering and pagination semantics

A frequent behavior trap is pagination. If you change query logic, you might accidentally alter ordering or the meaning of a cursor.

Existing behavior (contract):

- Results are sorted by `createdAt` descending.
- Pagination uses `cursorId` such that items with `createdAt` equal to the cursor are handled deterministically.
- The endpoint returns the same page boundaries for the same dataset.

Prompt that preserves behavior:

> "Optimize the query by reducing redundant conditions, but preserve the exact ordering and cursor semantics. Do not change the sort keys, do not change the cursor comparison operators, and do not change the shape of the response. Provide a minimal diff and include a test case that asserts the page boundary for equal `createdAt` values."

Why this works: it tells the assistant which parts are "sacred" (sort keys and cursor comparisons) and which parts are negotiable (redundant conditions).

Example 3: Keep log messages stable (when they're part of the contract)

Sometimes tests or monitoring rely on log message text. If you change logs, you can break downstream parsing.

Contract item:

- Log message must remain exactly: `"Cache miss for user"` followed by userId.

Prompt constraint:

> "You may restructure the caching logic, but keep the log message prefix and the placement of the log call relative to the cache lookup. Preserve the number of log emissions per request."

Extra proof request:

- "Add/adjust a unit test that asserts the log call count and message prefix."

Patch discipline: ask for minimal diffs and explicit mapping

Even when the assistant claims it preserved behavior, you want a way to audit the change quickly.

Ask for:

- **Minimal diff**: only the lines needed for the goal.
- **Function signature preservation**: no parameter renames that change call sites.
- **Explicit mapping**: "Old branch A -> new branch A."

This reduces the chance of silent behavior drift.

Verification loop: make the assistant tell you what to run

A behavior-preserving change should come with a verification plan that matches the contract.

Ask for:

- "Run existing unit tests that cover the contract items."
- "Add tests only when a contract item is currently untested."
- "Provide a small script or snippet to compare outputs for a fixed set of inputs."

If the assistant can't name what to run, it probably didn't understand the contract.

Mind map: "Proof of preservation"

Click here to view the mind map: Proof of Preservation

A final prompt example you can reuse

Goal: Improve readability of <function/module>.

Behavior contract (do not change):

- For input X -> output Y.
- For invalid input A -> throw <ErrorType> with message "...".
- Preserve ordering and pagination semantics: <describe keys/cursor rules>.
- Preserve side effects: <what is written/emitted>.
- Preserve log message prefix "Cache miss for user" and log call count.

Change boundaries:

- Allowed: reorganize code, extract helpers, rename local variables.
- Forbidden: change exception types/messages, change sort keys, change cursor operators, change response schema.

Deliverable:

- Provide a minimal diff.
- List each contract item and the exact code location that preserves it.
- Provide commands to run tests and a small input set for before/after comparison.

Preserving behavior is less about "being careful" and more about forcing the assistant to operate inside a clearly stated contract, then requiring a minimal patch and a verification plan that matches that contract.

13.5 Handling Ambiguity With Follow Up Questions

When a coding assistant's output doesn't match what you meant, the problem is usually not "bad AI," but missing constraints. Ambiguity shows up as vague requirements, unclear inputs/outputs, or multiple plausible interpretations of the same task. Follow-up questions turn guesswork into a shared spec.

A practical rule: ask for the smallest missing piece

Start with the question that would most reduce uncertainty. If you're unsure about behavior, ask about edge cases. If you're unsure about interfaces, ask about types and expected shapes. If you're unsure about performance, ask for the target constraints.

Mind map: ambiguity sources and follow-up targets

Click here to view the mind map: Ambiguity

A follow-up question template you can reuse

Use a short structure: **Goal** → **Current assumption** → **Question** → **Example**.

Example template (fill in the blanks):

> Goal: Implement `X`. Current assumption: We should treat missing field `Y` as `Z`. Question: Should `Y` be required, optional, or defaulted? Example: If input is `{...}` and `Y` is absent, what should the output be?

This style forces the assistant to commit to a specific interpretation.

Example 1: ambiguous function behavior

You ask for a function that "parses a date string." The assistant returns something that accepts many formats, but your app expects one.

Your follow-up:

- "What exact input formats should be accepted? For example, should `2026-03-24` and `03/24/2026` both work, or only ISO-8601?"
- "If the input is invalid, should the function return `null`, throw an error, or return a default date?"
- "What timezone should be used when the input has no timezone information?"

Why this works: date parsing is a classic ambiguity trap because multiple behaviors are reasonable. By asking for accepted formats, invalid handling, and timezone rules, you eliminate the assistant's freedom to choose a "reasonable" default.

Example 2: unclear data shapes in API code

You request a handler that "returns user profile." The assistant invents fields and a response shape.

Your follow-up:

- "What is the exact response schema? Please list fields and types."
- "Which fields are optional? Which are guaranteed?"
- "Should the handler include nested objects (e.g., `settings`) or flatten them?"

Add a concrete example:

- "Given user id `42`, here is the expected JSON response: `{...}`. Use that shape."

If you can't provide a full example, provide a minimal one with only the fields you care about. The assistant can then align the rest.

Example 3: multiple plausible algorithms

You ask to "optimize this loop." The assistant proposes a complex change that's hard to verify.

Your follow-up:

- "What is the bottleneck evidence? Is it CPU time, database calls, or memory usage?"
- "What are the input sizes in production? Give typical and worst-case ranges."
- "Are we allowed to change the algorithmic approach, or must we keep the same data access pattern?"

Then constrain the output:

- "Propose two options: one minimal change and one bigger refactor. For each, state expected complexity and how you'd test it."

This turns "optimize" into a decision with measurable tradeoffs.

Example 4: unclear error handling and logging

You ask for "better error messages." The assistant changes wording but not behavior.

Your follow-up:

- "Should errors be user-facing, developer-facing, or both?"
- "Do we log stack traces? If yes, where and at what level?"
- "What error codes should be returned for each failure mode?"

Add a mapping table in your question:

- "If validation fails, return code `4001`. If the downstream service times out, return code `5032`. Otherwise return `5000`."

Now the assistant can implement consistent behavior rather than guessing.

Example 5: scope ambiguity in refactors

You ask to "clean up this module." The assistant rewrites too much.

Your follow-up:

- "Which functions must remain unchanged in signature?"
- "Is changing internal structure allowed as long as behavior stays the same?"
- "Do we need to preserve public exports exactly?"
- "What is the maximum file churn allowed? For example, only modify `moduleA.ts`."

A good follow-up also asks for boundaries: "Keep the diff small" is vague, but "Only change these three functions" is actionable.

How to ask follow-ups without slowing down

1. **Batch questions.** Ask 3–6 targeted questions at once, not one per message.
2. **Use "either/or" when possible.** "Return `null` or throw?" is faster than "How should it handle errors?"
3. **Provide one example input and expected output.** Even a tiny example anchors the assistant's interpretation.
4. **Request a clarification before code.** If the assistant starts coding immediately, stop and ask for a short spec first.

Mini checklist before accepting generated code

- Did we define inputs, outputs, and edge cases?
- Did we specify error handling and logging expectations?
- Did we constrain scope and allowed refactors?
- Did we provide at least one example to confirm behavior?

A compact follow-up set you can copy

Use this when you're stuck and need the assistant to ask you the right questions.

> Please confirm: (1) input shape and types, (2) output shape and types, (3) required vs optional fields, (4) behavior on invalid input, (5) timezone/units assumptions, (6) which files/functions you will modify, and (7) what tests you will add or update.

This list covers the most common ambiguity points. If the assistant answers these directly, the next code iteration is usually straightforward.

14. End to End Case Studies

14.1 Case Study Debugging a Production Crash With Logs

A production crash is rarely "mysterious." Logs usually contain the story; the trick is reading them in the order the system experienced them. This case study shows a practical workflow: isolate the first failure, connect it to the code path, reproduce locally with the smallest input, then validate the fix with targeted tests.

Scenario

An API endpoint intermittently returns HTTP 500. After a deploy, the error rate spikes. The service logs show a stack trace, but the stack trace alone doesn't explain why it happens only sometimes.

What we have in logs

- A request ID for each incoming request.
- A timestamped stack trace.
- A few structured fields (like `userId`, `tenantId`, `route`, `payloadSize`).
- Occasional warnings right before the crash.

Step 1: Find the earliest signal

Start by searching for the first occurrence after the deploy time. Don't focus on the last line of the stack trace yet; the earliest log entry often contains the real trigger.

Example log excerpt (sanitized):

```
12:04:11.982Z INFO requestId=7f3a route=/v1/quote tenantId=acme payloadSize=812
12:04:11.991Z WARN requestId=7f3a cacheMiss key=pricing:acme:standard
12:04:12.004Z ERROR requestId=7f3a NullPointerException at QuoteService.java:118
12:04:12.004Z ERROR requestId=7f3a causedBy=java.lang.NullPointerException: ratePlan is null
```

The key detail is `ratePlan is null`, which is more specific than the generic exception. The warning about `cacheMiss` is also suspicious because it suggests a fallback path.

Step 2: Build a mind map of the failure path

Use a mind map to connect logs to code responsibilities: request handling, data fetching, caching, and calculation.

Mind map: Production crash investigation

Click here to view the mind map: Production crash investigation

Step 3: Map the stack trace to a concrete code location

QuoteService.java:118 is where `ratePlan` is dereferenced. The goal is to identify the exact line that assumes non-null.

A simplified version of the problematic code might look like this:

```
// QuoteService.java (simplified)
RatePlan ratePlan = ratePlanCache.getOrFetch(tenantId, planType);
BigDecimal price = ratePlan.getBasePrice().multiply(quantity);
return new Quote(price);
```

If `getOrFetch` returns `null` on cache miss for some tenants, the crash is deterministic for those inputs, even if it appears “intermittent” due to traffic patterns.

Step 4: Use logs to narrow the input conditions

Because the crash is tied to `ratePlan is null`, we want to know when `ratePlanCache.getOrFetch` returns null.

Look for patterns in the structured fields across multiple failing request IDs:

- Do failures share the same `tenantId`?
- Do they share the same `planType`?
- Do they correlate with `payloadSize` or a specific route variant?

Example comparison (sanitized):

- Failing: tenantId=acme, planType=standard, cacheMiss=true
- Passing: tenantId=acme, planType=standard, cacheMiss=false
- Failing: tenantId=globex, planType=standard, cacheMiss=true

This suggests: cache hits work; cache misses fail.

Step 5: Reproduce locally with a minimal test case

The fastest path is to reproduce the cache-miss fallback behavior.

Create a unit test that forces `getOrFetch` to take the fallback path and returns null from the underlying fetch.

```
@Test
void cacheMissFallbackShouldNotReturnNullRatePlan() {
  RatePlanFetcher fetcher = mock(RatePlanFetcher.class);
  when(fetcher.fetch(anyString(), anyString())).thenReturn(null);

  RatePlanCache cache = new RatePlanCache(fetcher);
  RatePlan ratePlan = cache.getOrFetch("acme", "standard");

  assertNotNull(ratePlan);
}
```

If the test fails, it confirms the bug: the cache layer allows null to escape. If the test passes, the null might be created later in the pipeline, so the stack trace line would need re-checking.

Step 6: Fix with a behavior that matches the domain

A fix should decide what “no rate plan” means. In many pricing systems, it’s better to return a controlled error (like HTTP 400 or 404) than to crash.

A safe approach:

- Make `getOrFetch` never return null.
- If the fetcher returns null, throw a domain exception with context.
- Convert that exception to an appropriate HTTP response.

Example fix sketch:

```
RatePlan ratePlan = ratePlanCache.getOrFetch(tenantId, planType);
if (ratePlan == null) {
  throw new MissingRatePlanException(tenantId, planType);
}
BigDecimal price = ratePlan.getBasePrice().multiply(quantity);
```

Then ensure the controller maps `MissingRatePlanException` to a non-500 response.

Step 7: Validate with targeted tests and log checks

Validation has two parts: correctness and observability.

1. Unit tests

- Cache miss with missing rate plan returns the expected domain error.
- Cache miss with valid rate plan returns a quote.
- Cache hit path still works.

2. Integration test

- Send a request that previously crashed.
- Assert the HTTP status and error body.

3. Log verification

- Confirm the new error includes `requestId`, `tenantId`, and `planType`.
- Confirm the stack trace no longer shows NullPointerException.

Example expected log after fix:

```
ERROR requestId=7f3a MissingRatePlanException tenantId=acme planType=standard
```

Step 8: Add one small piece of instrumentation (only where it matters)

If the underlying fetcher sometimes returns null due to data issues, you want a single log line that explains why.

Add logging inside the fetcher when it cannot find a rate plan:

- Include tenant and plan type.
- Avoid logging full payloads.

This keeps future debugging grounded: you'll know whether the problem is "cache fallback returned null" or "data lookup found nothing."

Summary of the reasoning chain

- The stack trace pointed to a dereference in QuoteService.
- The logs added a more specific message: `ratePlan is null`.
- Comparing failing vs passing requests showed a correlation with cache misses.
- A minimal unit test reproduced the null escape.
- The fix changed behavior from "crash" to "controlled error," and tests confirmed the new contract.

The result is not just fewer 500s; it's a clearer system. When rate plan data is missing, the service responds consistently and the logs explain the cause without forcing you to interpret a NullPointerException like it's a riddle.

14.2 Case Study Refactoring a Legacy Module With Tests

A legacy module often "works" because it has survived real usage, not because it is easy to change. The goal here is to refactor without changing behavior, while gradually improving readability and test coverage. The trick is to let tests describe the current behavior first, then refactor in small steps.

Starting point: what "legacy" looked like

The module handled order totals. It had three pain points:

- Business rules were spread across multiple functions with overlapping responsibilities.
- Edge cases were handled implicitly (for example, negative quantities or missing tax rates).
- Tests existed, but they were sparse and mostly checked final numbers, not intermediate decisions.

The first refactor target was a function that computed totals from an order object. It accepted a loosely shaped input and returned a number.

Example of the legacy shape

- Input: `{ items: [{ qty, unitPrice }], taxRate?, discount? }`
- Output: `total` as a floating-point number

The legacy function mixed validation, arithmetic, and rule selection. That made it hard to reason about failures.

Step 1: Create a behavior map (before changing code)

Before touching the module, capture what it currently does. The easiest way is to write a "behavior map" from existing tests and a few manual runs.

Mind map: behavior map

Click here to view the mind map: Behavior map: Legacy order total

This map becomes the acceptance criteria for refactoring. Even if the behavior is not ideal, it is the behavior we must preserve.

Step 2: Add tests that pin down behavior

If you refactor first, you risk changing behavior and then not knowing why. If you test first, you can refactor with confidence.

Mind map: test plan

Click here to view the mind map: Test plan: Pin down behavior

Concrete test examples

Use small, explicit fixtures so failures are easy to interpret.

- Empty items:
  - Input: `{ items: [] }`
  - Expected: `0`
- Missing taxRate:
  - Input: `{ items: [{ qty: 2, unitPrice: 10 }] }`
  - Expected: `20`
- Discount applied after tax (legacy order):
  - Input: `{ items: [{ qty: 1, unitPrice: 100 }], taxRate: 0.1, discount: 5 }`
  - Expected: `105` (since tax makes 110, then subtract 5)

A common mistake is to "fix" the discount order while refactoring. The tests prevent that.

Step 3: Use an AI assistant to propose a refactor plan, not a rewrite

When prompting, ask for a plan that preserves behavior and produces a patch in small steps. The assistant should help you identify seams: where to extract functions, where to normalize inputs, and what to keep unchanged.

Prompting approach (example)

Ask for:

- A list of pure helper functions to extract
- A suggested refactor sequence
- A checklist of behavior invariants to preserve

Then implement the plan manually, using tests as the guardrail.

Step 4: Refactor by extracting pure functions

The first extraction should be the simplest: compute subtotal from items. This function can be pure and easy to test.

Example: extracting subtotal

- Legacy: subtotal logic embedded in the total function
- Refactor: `computeSubtotal(items)` returns `sum(qty * unitPrice)`

Keep the exact arithmetic behavior. If the legacy used raw floating math, keep it.

Next, extract tax calculation:

- `computeTax(subtotal, taxRate)`
- If `taxRate` is missing, return `0` (matching the behavior map)

Then extract discount:

- `computeDiscount(subtotalPlusTax, discount)`
- If `discount` is missing, return `0`
- Apply discount after tax, matching the legacy order

Finally, the main function becomes orchestration:

- `total = subtotal + tax - discount`

Even if the module is still “not pretty,” it becomes understandable.

Step 5: Normalize inputs without changing outputs

Legacy code often tolerates missing fields. A refactor should preserve that tolerance.

A safe pattern is to normalize optional fields at the boundary:

- `taxRate = order.taxRate ?? 0`
- `discount = order.discount ?? 0`

Be careful: if the legacy treated `null` differently from `undefined`, tests should capture that. If tests don’t cover it, add a test before normalizing.

Step 6: Add targeted tests for extracted helpers

Once helpers exist, test them directly. This reduces the chance that a future change breaks a sub-rule while the overall total still passes a coarse test.

Example helper tests

- `computeSubtotal([{ qty: 2, unitPrice: 10 }])` -> `20`
- `computeTax(100, undefined)` -> `0`
- `computeTax(100, 0.1)` -> `10`
- `computeDiscount(110, undefined)` -> `0`
- `computeDiscount(110, 5)` -> `5`

Step 7: Refactor naming and structure after behavior is stable

With tests passing, you can improve readability:

- Rename variables to match business meaning (`subtotal`, `taxAmount`, `discountAmount`).
- Remove dead branches that tests prove are unreachable.
- Consolidate duplicated arithmetic.

A good rule: if you can’t explain a line in one sentence, it probably belongs in a helper with a descriptive name.

Step 8: Use the assistant to review for subtle behavior drift

After each refactor step, ask the assistant to compare the old and new logic at the rule level. The assistant should point out where behavior could drift, such as:

- Discount order (before vs after tax)
- Default values for missing fields
- Rounding differences
- Exception behavior (legacy might never throw)

Then verify with tests.

Step 9: Final verification and cleanup

At the end:

- All tests pass.
- The main function reads like a checklist.
- Helpers are small and pure.
- Edge cases are documented by tests, not comments.

Mind map: refactor checklist

Refactor checklist

- [ ] Tests describe current behavior
- [ ] Extract pure helpers first
- [ ] Preserve arithmetic and default handling
- [ ] Keep discount/tax order identical
- [ ] Normalize inputs only after tests cover null/undefined
- [ ] Add helper-level tests
- [ ] Refactor naming after behavior is stable
- [ ] Review for drift: rounding, ordering, exceptions

This approach turns a risky rewrite into a sequence of small, verifiable changes. The module becomes easier to maintain, and the tests become a living specification of what the legacy code actually did.

14.3 Case Study Optimizing a Slow Endpoint With Benchmarks

Scenario

A web service exposes `GET /reports/{id}`. Under load, the endpoint averages ~2.8s and sometimes spikes above 6s. The response is a JSON report built from: (1) a primary record, (2) related rows, and (3) a computed summary.

The goal is not to “make it faster” in general; it’s to reduce the measured latency while keeping the output identical. We’ll use benchmarks to separate “faster code” from “faster luck.”

Baseline: measure before changing

Start with a repeatable benchmark that hits the endpoint with realistic parameters and concurrency.

Benchmark plan

- Warm up the service (ignore the first few runs).
- Run a fixed number of requests (e.g., 200) at a fixed concurrency (e.g., 10).
- Record p50, p95, and error rate.
- Capture server-side timing breakdown (DB time, serialization time, external calls if any).

Example benchmark output (baseline)

- p50: 2.1s
- p95: 5.7s
- DB time: 1.6s average
- CPU time: 0.3s average
- Serialization: 0.1s average

The breakdown suggests the database dominates, and the p95 spike likely comes from query variability or N+1 patterns.

Mind map: what to investigate

Mind map: Optimizing the slow endpoint

Click here to view the mind map: Endpoint: GET /reports/{id}

Step 1: confirm query count and shape

Add lightweight instrumentation around the data access layer. For each request, log:

- number of SQL statements
- total DB time
- top 3 slow queries by duration

What you might find

- 1 query for the report
- 1 query for each related row (classic N+1)
- a summary query that scans too many rows

Example (symptom)

- 1 report query
- 42 related queries
- 1 summary query

Even if each related query is "only" 30–40ms, 42 of them becomes a few seconds, and p95 grows when the database is busy.

Step 2: collapse N+1 into fewer queries

Replace per-row lookups with a single query that fetches all related rows in one go.

Before (conceptual)

- Fetch report
- For each related id: fetch related row

After (conceptual)

- Fetch report
- Fetch all related rows with `WHERE related_id IN (...)`
- Build the response in memory

Easy-to-understand example (SQL shape)

- Before: `SELECT * FROM related WHERE id = ?` repeated
- After: `SELECT * FROM related WHERE id IN (?, ?, ...)`

Benchmark expectation If query count drops from ~44 to ~2–3, DB time should fall sharply and p95 should tighten.

Step 3: add or correct indexes for the summary query

The summary query might look like:

- filter by `report_id`
- group by `category`
- order by `created_at`

If the table is large, the database may scan many rows. Check the query plan and ensure there's an index that matches the filter and grouping.

Index example

- If you filter by `report_id` and then group by `category`, an index on `(report_id, category)` can help.
- If you also sort by `created_at`, consider `(report_id, category, created_at)` depending on the database and query pattern.

Verification detail After adding an index, re-run the benchmark and confirm:

- DB time decreases
- p95 decreases more than p50 (often the case when scans are removed)

Step 4: reduce payload work (without changing output)

Suppose serialization shows 0.1s average, but it might still contribute to p95 when payloads are large. Confirm whether the endpoint returns more fields than needed.

A common issue is computing derived fields for every related row when only some are included in the response.

Example optimization

- Before: compute `score` for all related rows, then filter
- After: filter first, then compute `score` only for included rows

This doesn't change the JSON if the filtered rows were never returned, but it reduces CPU time and memory churn.

Step 5: re-run benchmarks and compare

Run the same benchmark suite as the baseline.

Example benchmark output (after query fixes + index)

- p50: 0.9s
- p95: 1.8s
- DB time: 0.6s average
- CPU time: 0.2s average
- Serialization: 0.1s average

The p95 improvement is the key win: it indicates the worst-case query path is no longer happening as often.

Step 6: ensure correctness with output equality checks

Performance work is only "done" when the response matches the baseline.

Practical approach

- Pick 20 representative report ids (small, medium, large).
- For each id, compare the JSON output from baseline and optimized versions.
- Compare after normalizing key order (JSON objects are unordered).

Example test snippet (conceptual)

- Call endpoint twice (or run both versions in test)
- Assert normalized JSON equality

This catches subtle differences like missing fields, ordering issues, or off-by-one grouping.

Mind map: the fix loop

Mind map: Benchmark-driven optimization loop

Click here to view the mind map: Benchmark-driven optimization loop

What to log during the case study (so it stays grounded)

For each benchmark run, record:

- request count and concurrency
- p50/p95 latency
- DB time average and p95 (if available)
- number of SQL statements per request (median and p95)
- top slow query signatures

This turns “it feels faster” into “the slow part moved.”

Final result summary

After collapsing N+1 queries, adding an index aligned with the summary query, and avoiding unnecessary per-row computations, the endpoint latency dropped from p50 2.1s to 0.9s and p95 5.7s to 1.8s. The response content remained identical across representative ids.

The best part is that the improvements are explainable: fewer queries reduced DB round-trips, the index reduced scan cost, and filtering before computation reduced CPU work. Benchmarks confirmed each change mattered, and p95 improved because the worst-case path was removed rather than merely averaged away.

14.4 Case Study Securing an API Against Injection Risks

A team maintains a small JSON API for managing customer profiles. A recent bug report mentions that searching by name sometimes returns unexpected records, and occasionally the API responds with a 500 error. The code path is simple: the client sends a `query` string, the server builds a database query, and returns matching rows.

The initial vulnerable behavior

The original implementation concatenates user input into a SQL string. It also logs the raw query for debugging. That combination creates two problems: injection (the database interprets attacker-controlled syntax) and information leakage (logs may capture sensitive payloads).

Example of the risky pattern (illustrative):

- Input: `query = "alice' OR '1'='1"`
- Constructed SQL: `SELECT * FROM customers WHERE name LIKE '%alice' OR '1'='1%'`
- Result: the `WHERE` clause becomes logically broader than intended.

Even if the API “usually works,” injection can still succeed silently by changing the meaning of the query.

Mind map: injection risk and defenses

Injection Risk Case Study Mind Map

Click here to view the mind map: Injection Risk Case Study

Step 1: Identify the exact injection point

The team traced the request handler to the data access layer. The handler passed `query` directly into a function that built SQL. The first fix is not to “sanitize” the string; it is to stop building SQL with string concatenation.

A good debugging move here is to capture the boundary: where user input becomes a query. Once you locate that boundary, you can enforce a single rule: user input must be treated as data, not syntax.

Step 2: Replace string concatenation with parameterized queries

In the revised code, the SQL statement is fixed, and the user input becomes a bound parameter. The server still supports substring search, so it uses a pattern parameter.

Example (illustrative, parameterized):

- SQL: `SELECT id, name FROM customers WHERE name LIKE ?`
- Parameter: `"%" + query + "%"`

This prevents the database from interpreting quotes or operators inside `query` as part of SQL syntax.

Concrete behavior change:

- Input: `alice' OR '1'='1`
- Pattern parameter: `%alice' OR '1'='1%`
- Database interprets it as literal text to match, not as logic.

Step 3: Validate input with practical constraints

Parameterization blocks syntax injection, but validation still matters because it reduces attack surface and improves reliability.

The team added:

- Type checks: `query` must be a string.
- Length limits: reject queries longer than a reasonable maximum (for example, 100 characters).
- Character policy: allow typical name characters plus spaces and hyphens; reject control characters.

Example validation rules:

- Reject if `query` contains `\u0000` (null byte) or other control characters.
- Trim leading/trailing whitespace before building the pattern.
- If `query` is empty after trimming, return an empty result set rather than running a broad query.

These rules are not about being clever; they prevent accidental heavy queries and make malicious payloads less likely to reach the database.

Step 4: Fix error handling and logging

The original code logged the raw SQL string and returned database error messages to the client. The revised approach:

- Logs store only safe metadata (request id, endpoint, validation outcome).
- Database errors are mapped to generic API errors.
- The client receives a consistent response shape, such as `{ "error": "Invalid request" }`.

This avoids leaking details like table names, SQL dialect quirks, or stack traces.

Step 5: Apply least-privilege database permissions

Even with parameterization, defense in depth helps. The team created a database user for the API with permissions limited to the required operations (for example, `SELECT` on specific tables, and no permission to modify schema).

If an injection attempt somehow reached a dangerous query, the database would still refuse actions outside the allowed scope.

Step 6: Add tests that prove the fix

The team wrote tests that focus on behavior, not implementation details.

Unit tests for validation

- `query = ""` returns empty results.
- `query` longer than the limit returns `400`.
- `query` containing control characters returns `400`.

Integration tests for injection attempts

Run the API against a test database seeded with known rows.

- Seed: customers named `Alice`, `Bob`, `Carol`.
- Attack input: `alice' OR '1'='1`.
- Expected: only rows matching the literal substring `alice' OR '1'='1` (likely none), not all rows.

A second test uses a payload that often breaks naive escaping:

- `query = "%"` should return all customers only if the API explicitly allows wildcard behavior.
- If wildcard behavior is not intended, the validation should reject `%` and `_` or escape them so they are treated as literal characters.

This is a subtle point: parameterization prevents SQL syntax injection, but `LIKE` patterns still interpret `%` and `_` as wildcards. The team decided on a policy and encoded it.

Mind map: test cases and expected outcomes

Injection Test Plan Mind Map

Click here to view the mind map: Injection Test Plan

What the final endpoint behavior looked like

After changes, the API:

- Accepts `query` as a string with clear limits.
- Uses a fixed SQL statement with bound parameters.
- Escapes or rejects wildcard characters based on an explicit policy.
- Returns consistent error responses without database internals.
- Logs safe metadata only.

The team also removed the temptation to "fix" injection by escaping quotes. Escaping can be correct in some contexts, but it is easy to get wrong when queries evolve. Parameterization keeps the rule simple: user input is data.

Quick checklist the team adopted

- No SQL built by concatenating user input.
- Parameterized queries everywhere user input reaches the database.
- Validation for type, length, and control characters.
- Explicit policy for `LIKE` wildcards (`%` and `_`).
- Generic client errors; safe server logs.
- Least-privilege database credentials.
- Tests that assert behavior under injection attempts.

This case study ended with fewer surprises: the API still supports search, but it no longer treats user input as instructions.

14.5 Case Study Improving Integration Reliability With Retries

A team had a service that calls an external payments API. Most requests succeeded, but a noticeable slice failed intermittently: timeouts, occasional 502/503 responses, and a few "connection reset by peer" errors. The failures were annoying because they were not tied to a specific input; they happened under normal traffic.

Goal

Increase successful completion rate without creating new problems like duplicate charges, runaway retry storms, or masking real bugs.

Starting point: what the logs showed

The team reviewed request logs and found three patterns:

- **Network/transport failures**: timeouts and connection resets.
- **Transient server errors**: 502/503 from the upstream.
- **Non-transient failures**: 400/401/403 and validation errors.

They also confirmed that the payments endpoint supports an **idempotency key**. That single detail shaped the whole plan: retries could be safe if every attempt used the same idempotency key.

Mind map: reliability plan

Click here to view the mind map: Integration Reliability With Retries

Step 1: classify errors correctly

The team created a small mapping from error signals to retry decisions.

- **Retry**: timeouts, connection resets, 502, 503, 504.
- **Do not retry**: 400, 401, 403, 422 (validation), and any error that clearly indicates a bad request.

Concrete example: if the upstream returns `422 Unprocessable Entity`, retrying would just repeat the same invalid payload. The fix there is input validation, not retries.

Step 2: define a retry policy that is bounded

They used a policy with:

- **Max attempts**: 4 total tries (1 initial + 3 retries).
- **Backoff**: exponential backoff starting at 200 ms.
- **Jitter**: randomize each delay by ±20% to reduce synchronized retries.
- **Total time cap**: stop retrying if the elapsed time exceeds 2.5 seconds.
- **Per-attempt timeout**: keep each HTTP call timeout short (e.g., 800 ms) so retries don't stack into long hangs.

A simple way to reason about this: each retry costs time and load. The policy must ensure that, in the worst case, the system fails fast enough to remain usable.

Step 3: ensure idempotency across retries

For each logical payment operation, the service generates one idempotency key and reuses it for every retry attempt.

Concrete example:

- Logical operation: "charge invoice 123 for $49.00".
- Idempotency key: `charge:invoice:123` (or a UUID derived once per operation).
- Every retry sends the same key.

Without this, retries could create multiple charges when the upstream actually processed the first request but the response got lost.

Step 4: implement the retry loop with clear logging

Below is a minimal pattern showing the core mechanics. It retries only transient failures, uses backoff with jitter, and logs attempt details.

```
import random, time

TRANSIENT = {"timeout", "conn_reset", 502, 503, 504}

def retry_call(call, should_retry, max_attempts=4, base_ms=200, cap_s=2.5):
    start = time.time()
    attempt = 1
    while True:
        try:
            return call()
        except Exception as e:
            reason = getattr(e, "reason", None)
            status = getattr(e, "status", None)
            key = reason if reason else status
            if attempt >= max_attempts or not should_retry(key):
                raise
            elapsed = time.time() - start
            if elapsed >= cap_s:
                raise
            delay = (base_ms * (2 ** (attempt - 1))) / 1000
            delay *= random.uniform(0.8, 1.2)
            time.sleep(delay)
            attempt += 1
```

A key detail: `should_retry` should be strict. If it's too permissive, the service will keep retrying bad requests and waste resources.

Step 5: integrate with the payments client

The team wrapped the upstream call so that:

- The idempotency key is computed once per logical operation.
- The same key is included in headers for every attempt.
- The HTTP client timeout is set per attempt.

Concrete example behavior:

- Attempt 1: send request with idempotency key `charge:invoice:123`.
- If it times out, attempt 2 repeats the same request with the same key.
- If attempt 2 succeeds, the operation completes once.

Step 6: tests that prove the policy

They added tests that focus on decision-making rather than implementation details.

1. **Unit tests for retry classification**
    - Given a simulated timeout error → retry.
    - Given a simulated 422 error → no retry.
2. **Unit tests for backoff bounds**
    - Ensure the number of attempts never exceeds the max.
    - Ensure total elapsed time respects the cap.
3. **Integration tests with a fake upstream**
    - Fake upstream returns 503 twice, then 200.
    - Verify the service makes exactly 3 calls and includes the same idempotency key each time.
4. **Safety test for duplicate prevention**
    - Fake upstream records idempotency keys and returns a "already processed" response on repeated keys.
    - Verify the service treats repeated attempts as successful completion when appropriate.

Mind map: test strategy

Click here to view the mind map: Tests for Retry Reliability

Step 7: observability so failures stay explainable

They logged three fields on every attempt:

- `operation_id` (the logical payment operation)
- `attempt` (1..N)
- `failure_reason` (timeout, conn_reset, status code)

They also emitted metrics:

- `payment_success_total`
- `payment_retry_total` (count of retries, not attempts)
- `payment_final_failure_total` (after retries are exhausted)

This mattered because retries can hide the original failure if you only look at final outcomes. With attempt-level logging, the team could still see whether the system was failing due to timeouts, upstream 503s, or something else.

Results and what they learned

After deploying the bounded retry policy with idempotency:

- Intermittent timeouts and 502/503 spikes caused fewer final failures.
- Duplicate charges did not increase, because every attempt used the same idempotency key.
- The service remained responsive under failure because total retry time was capped.

The most important lesson was not "retries are good," but "retries must be selective and safe." The team treated retry logic like a small system with rules, tests, and visibility—because it is.

15. Operational Best Practices for Using AI in Coding Teams

15.1 Establishing Review and Acceptance Criteria

A good review process answers two questions: "What counts as correct?" and "How will we know quickly?" When you use AI coding assistants, the answers need to be explicit because the assistant can produce plausible code that compiles but still violates your project's rules.

Start with a short acceptance checklist

Use a checklist that reviewers can apply in under five minutes per change. Keep it stable across the team, but allow small variations by change type (bug fix vs. feature vs. refactor).

Baseline acceptance criteria (apply to most PRs):

- **Correctness:** The change fixes the stated issue or implements the requested behavior.
- **Safety:** No new security-sensitive behavior without review (authz checks, input handling, file/network access).
- **Tests:** Either existing tests pass and cover the behavior, or new tests are added for the changed logic.
- **Determinism:** No flaky timing, randomness without seeding, or reliance on environment-specific state.
- **Style and conventions:** Naming, formatting, and structure match the repository's norms.
- **Performance sanity:** No obvious algorithmic regression in hot paths; any heavier work is justified.
- **Documentation:** Comments or docs explain non-obvious decisions, and error messages remain consistent.

Example acceptance criteria statement (copy/paste into a PR template):

> "This PR is accepted if: (1) the reported bug is reproduced and fixed, (2) new/updated tests cover the behavior, (3) lint and type checks pass, (4) no public API changes occur without updating callers, and (5) any security-relevant logic is reviewed line-by-line."

Define review scope: what reviewers must check

AI-generated code often needs targeted scrutiny. Reviewers should focus on areas where mistakes are common and impact is high.

High-attention areas:

- **Boundary conditions:** empty inputs, null/undefined, off-by-one, inclusive/exclusive ranges.
- **Error handling:** exceptions vs. error returns, status codes, and message consistency.
- **Data transformations:** mapping fields, units, time zones, and serialization formats.
- **Concurrency and side effects:** shared state, locks, async ordering, idempotency.
- **Authorization and validation:** who can do what, and what inputs are allowed.

Low-attention areas (still reviewed, but faster):

- Straightforward boilerplate (e.g., getters/setters) when tests and style checks cover it.
- Mechanical renames that don't change behavior.

Require evidence, not vibes

Acceptance criteria should specify what evidence counts. "Looks right" is not evidence.

Evidence types that work well:

- **Test results:** unit/integration tests passing, plus at least one test that would fail without the fix.
- **Reproduction notes:** a short description of how the bug was reproduced before the change.
- **Diff rationale:** a brief note explaining why the approach was chosen, especially for tricky logic.
- **Static checks:** type checking, linting, and formatting.
- **Benchmark or profiling (only when needed):** a before/after measurement for performance-sensitive changes.

Example evidence snippet for a bug fix:

> "Before: `GET /orders?limit=0` returned 500 due to division by zero in pagination. After: returns an empty list with `limit=0` and `total` unchanged. Added test `test_pagination_limit_zero` to prevent regression."

Use mind maps to keep criteria consistent

Mind maps help teams align on what "good" means without turning the checklist into a wall of text.

Mind map: Review & Acceptance Criteria

Click here to view the mind map: Acceptance criteria

Add AI-specific guardrails (without making reviews miserable)

AI assistants can generate code that is syntactically correct but semantically off. Guardrails should be practical.

Guardrail 1: "No behavior change without a test." If the assistant changes logic, require a test that asserts the new behavior or the fixed bug.

- Bad: "We changed the function to be cleaner; no tests added."
- Good: "Refactor plus test `test_calculate_total_discounts_applied` to lock behavior."

Guardrail 2: "No silent API changes." If function signatures, response shapes, or error types change, require updates to callers and tests.

- Bad: "Changed return type from `int` to `float` to avoid rounding."
- Good: "Updated callers and added test verifying rounding rules."

Guardrail 3: "Security-sensitive lines get extra attention." When AI touches auth, validation, serialization, or SQL/query construction, reviewers should check the exact logic rather than trusting the assistant's explanation.

- Example: If the assistant rewrites a query, confirm parameterization and that user-controlled fields are not concatenated.

Provide concrete examples of acceptance outcomes

These examples show how criteria translate into decisions.

Example A: Accepted change (bug fix)

- The PR includes a failing test that reproduces the bug.
- The fix updates the logic and keeps error messages consistent.
- Lint/type checks pass.
- Reviewers confirm boundary handling for empty input.

Example B: Requested changes (missing evidence)

- The PR claims a logic fix but only updates code, with no new tests.
- The change touches pagination boundaries.
- Reviewer requests: add a test for `limit=0` and `offset` negative/large cases.

Example C: Accepted change (refactor with preserved behavior)

- The PR refactors a function for readability.
- No behavior changes are intended.
- Existing tests cover the function's behavior; reviewers confirm no signature changes.

Make criteria easy to apply during review

Reviewers should be able to answer "pass/fail" quickly.

Click here to view the mind map: Quick review rubric (use as a scoring guide)

A rubric is not a replacement for judgment; it prevents the review from becoming a subjective debate. If the PR scores low in tests or safety, reviewers should request changes even if the code "looks fine."

Define how to handle AI-generated explanations

AI assistants often provide explanations with the code. Treat explanations as a starting point, not proof.

Acceptance rule: the code and tests are the source of truth.

- If the explanation conflicts with the diff, reviewers follow the diff.
- If the explanation is vague ("optimized for performance"), reviewers request concrete evidence (tests, benchmarks, or a clear reasoning note).

Close with a clear acceptance decision template

Use a short template so reviewers can record decisions consistently.

Click here to view the mind map: Acceptance decision template

With these criteria in place, AI-assisted changes become easier to review because "good" is defined in observable terms: tests, boundaries, safety-sensitive logic, and consistent behavior.

15.2 Creating Prompt Templates for Common Workflows

Prompt templates turn “good results by luck” into “good results on purpose.” The goal is not to write clever prompts; it’s to standardize what you provide and what you ask for, so the assistant can produce code that matches your project’s rules.

Mind map: Prompt templates as a workflow system

Click here to view the mind map: Prompt Templates

Template design principles (practical, not theoretical)

1. **State the target and the boundary.** Tell the assistant what to change and what not to touch. Example boundary: “Do not change the public function signature.”
2. **Provide a small, representative example.** If you’re fixing logic, include one failing input and the expected output. If you’re optimizing, include a sample dataset shape.
3. **Ask for an output contract.** Specify whether you want a unified diff, a full file, or a patch-style change. Also specify what tests or checks to include.
4. **Require verification steps.** The assistant should propose commands (or at least checks) that confirm the change worked.
5. **Use a consistent “question-first” pattern.** When requirements are unclear, ask the assistant to ask you up to N clarifying questions before writing code.

Core prompt skeleton (copy/paste)

Use this skeleton for most workflows.

```
You are helping me modify an existing codebase.
Goal: <one sentence>
Boundary: <what must not change>
Context: <language, framework, versions, relevant files>
Constraints: <style, performance, security, dependencies>
Inputs/Examples: <failing case or sample data>
Acceptance criteria:
- <testable item 1>
- <testable item 2>
Output format: <diff|patch|full file>.
Also include: <tests to add/adjust> and <commands to run>.
If anything is missing, ask up to 3 clarifying questions first.
```

Workflow templates with examples

A) Template: “Implement a feature with tests”

When to use: You have a clear requirement and want correct code plus coverage.

```
Goal: Add <feature> to <module>.
Boundary: Do not change existing public APIs except where explicitly required.
Context: <paste relevant interfaces and current behavior>
Constraints: Follow existing patterns; add no new dependencies.
Inputs/Examples:
- Input: <example>
- Expected: <expected output>
Acceptance criteria:
- Unit tests cover normal and edge cases.
- Lint passes.
- Existing tests remain green.
Output format: Provide a unified diff.
Include: new/updated tests and the exact test command(s).
Ask up to 3 clarifying questions if needed.
```

Example prompt (small feature):

```
Goal: Implement a function parse_user_id(s) that accepts strings like "user:123".
Boundary: Keep the function name and return type unchanged.
Context: Current file has a stub and a TODO.
Constraints: Use existing error types; no new dependencies.
Inputs/Examples:
- "user:123" -> 123
- "user:-1" -> error
- "USER:123" -> error (case-sensitive)
Acceptance criteria:
- Add unit tests for all examples.
- Existing test suite passes.
Output format: Unified diff.
Include: tests and command to run.
```

B) Template: "Debug from stack trace"

When to use: You have a failure and want a targeted fix.

```
Goal: Fix the failing behavior causing the error.
Boundary: Keep behavior unchanged except for the bug.
Context:
- Language/runtime: <...>
- Stack trace: <paste>
- Recent changes: <optional>
Constraints: Prefer minimal code changes.
Inputs/Examples:
- Command that fails: <...>
- Observed error: <...>
Acceptance criteria:
- Tests pass.
- Add a regression test that would fail before the fix.
Output format: Unified diff.
Also include: explanation of root cause in 3-6 bullet points.
Ask up to 3 clarifying questions if needed.
```

Example prompt (logic bug):

```
Goal: Fix why checkout sometimes charges 0.
Boundary: Do not change payment provider integration.
Context:
Stack trace: <paste>
Relevant code:
<include the function that computes total>
Constraints: Keep rounding rules consistent with existing currency utilities.
Inputs/Examples:
- Cart: [{sku:"A", qty:1, price:0.99}] -> total should be 0.99
- Cart: [{sku:"B", qty:2, price:1.50}] -> total should be 3.00
Acceptance criteria:
- Add regression test for the failing cart.
- All tests pass.
Output format: Unified diff.
```

C) Template: "Refactor without changing behavior"

When to use: You want readability, structure, or reduced duplication.

```
Goal: Refactor <area> for readability and maintainability.
Boundary: No behavior changes; keep public interfaces identical.
Context: <paste current functions/classes>
Constraints: Preserve error messages and return values.
Acceptance criteria:
- All tests pass.
- No new branches that change outcomes.
- Add/adjust tests only if coverage is missing.
Output format: Unified diff.
Also include: a short list of refactor steps and what each step preserves.
Ask up to 3 clarifying questions if needed.
```

Example prompt (naming + extraction):

```
Goal: Refactor a long function that validates an order.
Boundary: Keep the same validation rules and error messages.
Context:
<include the function>
Constraints: Keep it in the same file.
Acceptance criteria:
- Tests pass.
- Extract helper functions with clear names.
Output format: Unified diff.
Include: 4-6 bullet explanation of what was preserved.
```

D) Template: "Optimize a hot path with measurable checks"

When to use: You have a performance issue and want improvements that you can verify.

```
Goal: Improve performance of <function/endpoint>.
Boundary: Output must be identical for all inputs.
Context:
- Current implementation: <paste>
- Metrics: <latency/throughput or profiling summary>
Constraints: Do not change external behavior; keep memory usage reasonable.
Inputs/Examples:
- Typical input size: <...>
- Worst-case input size: <...>
Acceptance criteria:
- Provide a benchmark plan and expected direction of change.
- Add a micro-benchmark or reuse existing one.
- All tests pass.
Output format: Unified diff.
Also include: complexity notes and where the time was saved.
Ask up to 3 clarifying questions if needed.
```

Example prompt (reduce repeated work):

```
Goal: Speed up render_report(rows) which is slow on large inputs.
Boundary: Keep the same output formatting.
Context:
<include render_report and any helper it calls>
Metrics:
- Profiling shows repeated parsing in a loop.
Constraints: No new dependencies.
Inputs/Examples:
- rows length: 10,000
- each row has fields: <shape>
Acceptance criteria:
- Add a benchmark comparing old vs new.
- Tests pass.
Output format: Unified diff.
```

Mind map: What to fill in each template

Click here to view the mind map: Fill-in Checklist

Prompt template library (short, reusable variants)

- **"Ask first" variant:** Add "If requirements are ambiguous, ask up to 3 questions before coding."
- **"Patch-only" variant:** Add "Output only a unified diff; do not include full files."
- **"Regression required" variant:** Add "Add a regression test that fails before the fix."
- **"Behavior lock" variant:** Add "Do not change outputs for any existing test cases."

A good template is boring in the best way: it makes the assistant's job smaller, your review easier, and your fixes more repeatable.

15.3 Managing AI Generated Code in Repositories

AI-generated code is easiest to manage when you treat it like any other incoming contribution: it must be reviewable, testable, and traceable. The goal is not to "trust the model," but to make it hard for mistakes to hide.

Core principles

1. **Make provenance visible**: every AI-produced change should carry enough metadata to answer "why is this here?" and "what should we verify?"
2. **Keep diffs small**: large, multi-purpose changes are where review quality goes to die. Ask for one logical change per request.
3. **Require tests as a gate**: if a change can't be validated automatically, it becomes a manual burden.
4. **Prefer patch-style edits**: when possible, ask for a diff/patch that touches only the relevant files and lines.
5. **Standardize acceptance criteria**: define what "done" means for correctness, style, and performance.

Repository workflow: from prompt to merged code

A practical workflow looks like this:

1. **Create a dedicated branch** for each AI-assisted task (e.g., `ai/fix-null-deref-2026-03-24`).
2. **Record the prompt and intent** in a short note in the PR description or a `docs/ai-notes/` entry.
3. **Run formatting and linting** locally before pushing.
4. **Add or update tests** that demonstrate the intended behavior.
5. **Review with a checklist** that focuses on risk areas: edge cases, error handling, and invariants.
6. **Merge only after CI passes** and the PR description includes what to verify.

This workflow works even if multiple people use the assistant, because it creates consistent artifacts for review.

Mind map: managing AI code artifacts

Click here to view the mind map: Managing AI Generated Code in Repositories

Provenance: what to record (and where)

A good provenance note is short but specific. Include:

- **Task statement**: "Fix null dereference in `parseUser()` when `email` is missing."
- **Constraints**: "Keep function signature unchanged; maintain backward compatibility."
- **What was generated**: "Assistant proposed guard clause + test for missing email."
- **Verification plan**: "Run unit tests for auth module; confirm no new warnings."

You can store this in the PR description, but a lightweight file in the repo also helps when you need to audit changes later.

Example: PR description template

```
Intent
Fix null dereference in `parseUser()` when `email` is missing.

Constraints
- Keep public function signatures unchanged.
- Preserve existing parsing rules.

AI output summary
- Added guard clause for missing `email`.
- Added unit test covering missing email.

Verification
- Ran formatter + linter.
- CI: unit tests + lint.
```

Change hygiene: keeping diffs reviewable

AI output often includes extra refactors "for clarity." That's fine in a separate branch, but it should not be mixed into a bug fix. A simple rule: **if the change isn't required to satisfy the task statement, it goes to another PR**.

Concrete tactics:

- Ask for "**only modify these files**" and "**only change the smallest necessary code region**."
- If the assistant proposes a refactor, respond with: "Keep behavior identical; only implement the bug fix and add tests."
- When reviewing, compare the diff to the task statement line-by-line. If a change doesn't map to the statement, it's a candidate for removal.

Verification: tests and checks that catch common failures

AI-generated code can fail in predictable ways: off-by-one errors, incorrect assumptions about input shape, missing error handling, and subtle behavior changes.

A good verification set includes:

- **Unit tests for edge cases** (empty inputs, null/undefined, boundary values).
- **Negative tests** for invalid inputs (assert error types/messages when appropriate).
- **Regression tests** that reproduce the original bug.

Example: adding a regression test

Suppose the bug is a crash when `email` is missing. The test should fail before the fix and pass after.

```
// parseUser.test.js
it('does not crash when email is missing', () => {
  const input = { id: 'u1' }; // email omitted
  expect(() => parseUser(input)).not.toThrow();
  const user = parseUser(input);
  expect(user.email).toBe(null);
});
```

If your codebase uses a different convention (e.g., `undefined` instead of `null`), align with existing behavior rather than inventing a new one.

Review discipline: a checklist that reduces risk

Use a checklist to keep reviews consistent across contributors.

- **Correctness**
  - Does the change handle the reported scenario?
  - Are boundary cases covered?
- **Behavior preservation**
  - Did we accidentally change output shape or error semantics?
- **Error handling**
  - Are failures explicit and actionable?
  - Are exceptions caught where callers expect them?

- **Complexity**
  - Did the change add unnecessary loops or repeated work?
- **Style and maintainability**
  - Are names clear and consistent?
  - Is the code aligned with local patterns?

A slightly playful but useful review question: **"If this were written by a new teammate, would we still understand it?"** If the answer is no, the diff needs tightening.

Repository structure: where AI notes live

If you want provenance beyond the PR description, create a dedicated folder and a simple naming convention.

Example structure:

- `docs/ai-notes/`
  - `2026-03-24-fix-parseUser-null.md`
  - `2026-03-24-optimize-query-pagination.md`

Each note should include:

- PR/branch reference
- Task statement
- Constraints
- Files touched
- Tests added/updated

Example: `docs/ai-notes/` entry

2026-03-24-fix-parseUser-null

PR: #123
Branch: ai/fix-parseUser-null

Intent Fix null dereference when `email` is missing.

Constraints

- Keep `parseUser(input)` signature.
- Preserve existing parsing rules.

Changes

- Added guard clause in parseUser.
- Added regression test: does not crash when email is missing.

Verification

- Ran unit tests and lint.

Commit message conventions (optional but helpful)

Commit messages can reinforce traceability. A lightweight convention is:

- `fix(parseUser): guard missing email (AI-assisted)`
- `test(parseUser): add regression for missing email`

This doesn't replace review, but it makes history easier to scan.

Practical "do and don't" summary

- **Do** require tests for behavior changes.
- **Do** keep diffs small and scoped.
- **Do** record intent and constraints.
- **Don't** accept refactors mixed into bug fixes.

- **Don't** merge code without CI passing.

When these habits are consistent, AI-generated code becomes just another input stream—manageable, reviewable, and accountable.

15.4 Tracking Decisions With Notes and Test Evidence

When you use an AI coding assistant, you're not just changing code—you're making decisions. The goal of this section is to help you record those decisions so that (1) teammates can understand why a change exists, (2) you can reproduce the reasoning later, and (3) tests provide concrete evidence that the change worked.

What to track (and what to skip)

Track decisions that affect behavior, correctness, performance, security, or maintainability. Skip tiny formatting changes unless they alter semantics (for example, changing a comparator from `<=` to `<`). A good rule: if someone could reasonably ask "why did you do it this way?", it belongs in your notes.

A decision record should answer four questions:

1. **Problem**: What was wrong or unclear?
2. **Decision**: What did we change and why that option?
3. **Evidence**: What tests, logs, or checks prove it?
4. **Constraints**: What must remain true (style rules, compatibility, time limits, API contracts)?

A lightweight decision note template

Use a short template in your PR description, a `DECISIONS.md` file, or a `docs/decisions/` folder. Keep it consistent so people can scan quickly.

> Tip: If the assistant produced multiple candidate solutions, record the comparison briefly. You don't need a novel—just enough to justify the chosen path.

Template (copy/paste):

- **Date**: YYYY-MM-DD
- **Context**: Component/module and relevant file paths
- **Problem**: Symptom or failing test name
- **Decision**: One or two sentences describing the change
- **Why this**: Constraints and reasoning (no more than 5 bullets)
- **Evidence**:
  - Tests run: `...`
  - Results: pass/fail summary
  - Notes: any follow-up observations
- **Rollback plan** (optional): What to revert if it regresses

Mind map: decision tracking workflow

Decision Tracking Mind Map

Click here to view the mind map: Decision Record

Mind map: evidence quality (what counts)

Evidence Quality Mind Map

Click here to view the mind map: Evidence

Practical example: recording an AI-assisted bug fix

Scenario: A service crashes when a request contains an empty list.

1. **Problem**: A unit test fails with a stack trace pointing to a function that assumes at least one element.
2. **Decision**: Add an early return for empty input and adjust downstream logic to handle "no items" as a valid case.

3. **Why this**:
    - The API contract allows empty lists.
    - The assistant suggested guarding the input, but you verify it matches the expected behavior.
    - You avoid changing the meaning of existing fields.
4. **Evidence**:
    - Run the specific failing test plus a small set of related tests.
    - Add a new test for the empty-list case if it wasn't already present.

Example decision note entry (short and concrete):

- **Date**: 2026-03-24
- **Context**: `src/orders/compute_totals.py`
- **Problem**: `test_compute_totals_empty_items` failing (IndexError)
- **Decision**: Return zero totals when `items` is empty; ensure callers treat it as a valid state.
- **Why this**:
    - Empty lists are allowed by the request schema.
    - Guarding at the boundary prevents cascading assumptions.
    - Keeps existing total calculation logic unchanged for non-empty inputs.
- **Evidence**:
    - Tests run: `pytest -q tests/orders/test_compute_totals.py`
    - Results: 1 failed -> 0 failed; added test for empty list
    - Notes: Verified no other callers rely on the previous crash behavior.

Practical example: recording a refactor without losing meaning

Scenario: The assistant proposes refactoring a function to reduce duplication, but you want to ensure behavior stays identical.

- **Problem**: Code duplication across two functions; risk of inconsistent updates.
- **Decision**: Extract a shared helper with the same input/output behavior; keep public function signatures unchanged.
- **Why this**:
    - You constrain the refactor to a pure transformation.
    - You request the assistant to preserve edge-case handling.
    - You confirm the helper is covered by existing tests.
- **Evidence**:
    - Run the full unit test suite for the module.
    - Run a small integration test set if the module touches external boundaries.

Example evidence lines to include:

- `pytest -q tests/payments/` (pass)
- `python -m compileall src/` (pass)
- `ruff check src/` (pass)

Capturing assistant suggestions responsibly

AI assistants often provide plausible changes. Your notes should record what you accepted and what you rejected.

When you reject a suggestion, write a one-line reason. Examples:

- "Rejected: changes API return type; would require updating multiple callers."
- "Rejected: adds caching without invalidation rules; risk of stale data."
- "Accepted with modification: kept guard clause but adjusted error message to match existing conventions."

This prevents the common "it worked for the assistant" problem from turning into "it worked once for us" folklore.

Where to store notes (and how to keep them findable)

Use one primary location per project:

- **PR description** for short-lived, review-focused decisions.
- `DECISIONS.md` for decisions that affect multiple future changes.
- **Issue comments** when the decision is tightly coupled to a specific bug report.

Make sure the entry includes file paths and test names so someone can jump straight to evidence.

A simple checklist for your final PR

Before merging, verify:

- The decision record states the problem clearly.
- The chosen approach is justified by constraints, not vibes.
- Evidence includes at least one relevant automated test run.
- Any new tests are named and explain the edge case.
- Notes mention any assistant suggestion that was modified or rejected.

Good decision tracking is mostly boring. That's the point: it turns "we think it's fine" into "we can show why it's fine," and it makes future debugging faster for everyone, including your future self.

15.5 Building a Consistent Workflow for Debug, Optimize, and Refactor

A consistent workflow keeps AI-assisted changes from turning into a pile of "almost right" edits. The goal is simple: every change should have a reason, a scope, and a way to prove it worked.

The core loop: Plan → Change → Prove

1. **Plan**: state the symptom, the suspected cause, and the smallest code area likely responsible.
2. **Change**: ask for a patch or a focused refactor, not a rewrite.
3. **Prove**: run the smallest set of checks that can falsify the hypothesis.

A good workflow makes it hard to skip steps. When you do skip, you should know exactly what you skipped and why.

Mind map: the workflow at a glance

Click here to view the mind map: Workflow: Debug → Optimize → Refactor

Step 1: Plan with a "single hypothesis" prompt

Before you ask for code, write one sentence that you can test. This prevents the assistant from guessing in multiple directions at once.

Example (debugging):

- Hypothesis: "`parseDate` returns `null` for valid ISO strings because it trims incorrectly before matching."

Example (optimization):

- Hypothesis: "The endpoint is slow because it performs an N+1 query pattern when mapping results."

Example (refactor):

- Hypothesis: "The module can be simplified by extracting a pure helper without changing output."

Then include evidence you already have:

- the failing test name or the exact error line
- the relevant input example
- the expected output

Step 2: Change in small, reviewable patches

Ask for a patch that touches only the scoped area. If the assistant proposes changes outside the scope, treat that as a red flag and request a narrower diff.

Example request for a debug fix (narrow scope):

- "Only modify `parseDate` and its direct callers. Keep the public signature unchanged. Provide a unified diff and explain why the fix addresses the hypothesis."

Example request for an optimization (preserve behavior):

- “Optimize the query logic to avoid N+1. Do not change the response schema. Provide a diff plus a short note on how you verified fewer queries occur.”

Example request for a refactor (behavior lock):

- “Refactor `UserService` by extracting `buildUserResponse` into a helper. Do not change any returned fields. Add/adjust unit tests if needed.”

A practical rule: if you can’t review the diff in a few minutes, it’s too big.

Step 3: Prove with the smallest falsification set

“Prove” doesn’t mean “run everything.” It means run the checks that can disprove your hypothesis.

Debug proving set

- the failing unit test (or the smallest reproduction test you can create)
- type checks / linting for the touched files
- one extra test case that covers the edge you suspect

Optimization proving set

- a benchmark that isolates the hot path
- a before/after comparison using the same input size
- a sanity check that output matches exactly (or within a defined tolerance)

Refactor proving set

- the full unit test suite for the module (or at least all tests that cover the affected functions)
- snapshot tests if you already use them
- one integration test if the refactor changes wiring rather than logic

Step 4: Use a “diff review checklist” before you merge

Even when tests pass, review for risk patterns. This checklist is short on purpose.

- **Scope**: Did the change stay within the planned files/functions?
- **Behavior**: Are there any silent changes to inputs/outputs (null handling, ordering, rounding)?
- **Error handling**: Did it swallow errors or change exception types?
- **Performance**: Did it add extra work in a loop or create new allocations?
- **Readability**: Are names and structure clearer, not just different?

Example of a risky diff to look for:

- Replacing a loop with a map that builds a large intermediate list.
- Changing a `try/except` to a broad catch without logging.
- Converting a stable sort to an unstable one.

Step 5: Record decisions so the next fix is faster

A workflow that forgets why it changed something forces you to re-argue the same points later.

Use a consistent note format in your PR description or a short internal log:

- **What**: symptom or goal
- **Hypothesis**: one sentence
- **Change**: what you asked the assistant to do (and what you constrained)
- **Proof**: commands/tests/benchmarks run and results
- **Risk**: one sentence about what could still be wrong

Example PR note (debug):

- What: failing test `test_parseDate_iso_zulu`
- Hypothesis: trimming removes the trailing `Z`, causing mismatch
- Change: update `parseDate` to normalize timezone markers before matching
- Proof: `pytest -k parseDate` (pass), lint (pass)

- Risk: other timezone formats might still fail

Step 6: Keep prompt templates consistent across the team

Consistency beats cleverness. A template reduces variation in how people ask for changes.

Template: Debug request

- "Given this error and this minimal input, fix the bug. Scope: only these files/functions. Constraints: keep public signatures. Output: unified diff + explanation tied to the hypothesis. Add/adjust tests if necessary."

Template: Optimize request

- "Improve performance for this hot path. Scope: only these functions. Constraints: keep response schema and semantics. Output: diff + what you measured and how."

Template: Refactor request

- "Refactor for clarity while preserving behavior. Scope: only these modules. Constraints: no semantic changes. Output: diff + list of extracted helpers and how tests confirm equivalence."

Step 7: A practical sequence for mixed work

When you're doing all three—debug, optimize, refactor—order matters.

1. **Debug first**: optimization and refactors can hide the real bug by changing timing or structure.
2. **Optimize next**: once behavior is correct, performance work can be measured.
3. **Refactor last**: structure changes after performance tuning reduce the chance you break the benchmark setup.

If you must refactor earlier, constrain it to "no behavior change" helpers and keep the proof set tight.

A complete mini-example: one ticket, three phases

Phase A: Debug

- Hypothesis: "`getUsers` returns duplicates because it merges pages incorrectly."
- Ask: "Fix only the merge logic; keep pagination contract; add a unit test for duplicate elimination."
- Prove: run the new unit test and the existing pagination tests.

Phase B: Optimize

- Hypothesis: "Duplicates cause extra work; also there's an N+1 query in mapping."
- Ask: "Optimize query/mapping to remove N+1; keep output identical; provide a benchmark before/after."
- Prove: run benchmark with the same dataset and confirm query count drops.

Phase C: Refactor

- Hypothesis: "The function is now correct but hard to read due to nested conditionals."
- Ask: "Refactor into small helpers; no semantic changes; update tests if needed."
- Prove: run the full test suite for the module.

This sequence keeps each phase accountable: debug fixes correctness, optimize improves measured performance, and refactor improves maintainability without changing what the system does.

MORE FROM RELATED INDUSTRIES

Software Development

AI Tools

MORE FROM RELATED ROLES

Programmers

Developers

📘 Green Buildings Modular Construction and Sustainable Infrastructure

📘 Practical Quantum Computing for Engineers

📘 AI Driven Software Development