ChatGPT

Vibe Engineering with Claude Code: End-2025 Playbook

Executive Summary

- **Structured AI–Human Collaboration:** *Vibe engineering* leverages disciplined workflows with AI coding agents. Instead of "vibe coding" (letting the AI freewheel), successful teams use *Claude Code* in planned, inspectable steps – e.g. reading relevant code, making a plan, then coding and verifying [1] [2]. This yields reliable results even on large (~1M LOC) codebases by keeping the human "in the loop" at key decision points.
- **Proven Claude Code Workflows:** Teams report best results using patterns like *"Explore-Plan-Code-Commit"* (explicit research & planning phases before any code) [3], *Test-Driven Development with Claude* (have it write failing tests, then code to pass them) [4] [5], iterative **visual-feedback loops** for UI changes (Claude compares screenshots to design mocks) [6], and even a guarded *"safe YOLO"* mode for rote chores like lint fixes [7]. These workflows are adapted based on context: legacy maintenance focuses on understanding and small fixes, rewrite parity work emphasizes preserving behavior, and greenfield development benefits from rapid scaffolding – each with distinct failure modes and mitigation strategies (detailed in Section A).
- **Context Management is Key:** Claude Code excels with a thoughtful context strategy. Users define a *"working set"* of relevant files/folders instead of dumping the whole monorepo into context [8] [9]. They practice **progressive disclosure** – start with high-level info and let Claude fetch details as needed [10] [11]. Persistent repository knowledge is stored in `CLAUDE.md` files (project docs, style guides, command tips) that load automatically [12] [13]. What *not* to include in context: huge vendor or build artifacts, secrets, or entire codebase snapshots (Claude will search or ask if needed). Long sessions are periodically reset with `/clear` to prevent diluted or stale context [14]. A pre-flight checklist (Section B) helps ensure Claude's context is focused and toxin-free before each task.
- **Memory and Persistence Model:** Rather than rely on hidden chat memory, Claude Code uses explicit files for long-term "memory." Teams persist stable facts – architecture decisions, naming conventions, *"golden" invariants – in version-controlled docs (e.g.* `CLAUDE.md` *at repo root)* [15] [16]. *Transient information (like a subagent's detailed analysis) is summarized and not blindly carried forward* [17] [18]. *This prevents misinformation buildup. Treat these memory files like code: review changes to them, enforce that no outdated assumptions linger unchallenged. Between major rewrite phases, it's common to* reset or prune memory*: e.g. clear obsolete migration instructions once a phase is done, to give Claude a clean slate for the next phase (Section C provides a memory template and governance rules).
- **Hard Guardrails, Not Gentle Nudges:** Claude operates best with strict **guardrails** that the team defines and enforces. Successful projects set scope limits ("only modify files in `/src/ui`, leave `/src/core` untouched" [19]), behavioral rules ("don't guess requirements – ask when uncertain" [20]), safety blocks (never read or write secrets or credentials – enforced via deny lists [21] [22]), quality requirements (e.g. require every code change to run tests and achieve zero failures), and architectural constraints (preserve public APIs and database schemas unless explicitly told to change [19]). Claude can be instructed to *STOP* on certain conditions – for example, if tests fail or if a planned change would exceed a risk threshold. Section D includes a one-page guardrail spec and an expanded policy document template that teams use to codify these rules for Claude (often in `.claude/rules/` or `CLAUDE.md`).

- **Least-Privilege Tool Access:** Claude Code's strength grows with tool integration – but these must be carefully permissioned. Successful setups grant Claude read-only access widely (code reading, searching, git history) while restricting write or exec privileges to only what's necessary [23] [24]. For example, Claude can be allowed to run the test suite and linters (so it can verify its changes), but perhaps not allowed to deploy or touch production data. Use Claude's allowlist/denylist settings to limit shell commands (e.g. allow `make build` or `npm test` but deny destructive commands) [25]. Give it git *read* access (blame, diff, log) freely, but gate git *write* operations behind user approval or branch protections [26] [27]. Logging and auditing are essential – every Claude tool action (file edit, bash command, etc.) can prompt for confirmation or be logged, ensuring traceability. Section E provides a tool-access matrix detailing the recommended privileges for each tool type and how to configure them safely (e.g. via `.claude/settings.json`).
- **Claude-First Development Workflows:** The playbook (Section F) details end-to-end example workflows optimized for Claude Code. These include: **(1) Bugfixing legacy code** with Claude – using it to isolate the bug, write a regression test, apply a minimal fix, and confirm the result with tests [5]; **(2) Test-first new feature development** – leveraging Claude to generate tests from specs, then implement code to make them pass (TDD) [4] [5]; **(3) Large-scale refactoring under guardrails** – where Claude plans the steps, executes small scoped refactors with frequent commits, and runs verifications at each step [28] [29]; **(4) Incident response & debugging** – feeding Claude logs and stack traces to pinpoint issues, using subagents or its search tools to trace through the codebase, and even generating a patch to fix the incident; **(5) Legacy-to-modern porting** – a structured approach in which Claude helps extract legacy system behavior and ports it to a new platform in stages, with side-by-side parity tests and explicit documentation of any intentional differences. Each workflow comes with an *agent brief*, a planning phase, execution steps, verification and testing, Git integration, and a human review checklist to ensure trustworthiness.
- **Git & PR Integration for AI-Assisted Code:** Using Claude means adapting your source control practices. Winning teams treat Claude as a junior dev: they create separate branches for Claude's work, enforce small, self-contained commits, and require thorough PR reviews. Claude can draft excellent commit messages and PR descriptions – for example, it will summarize code changes in context [30] [31] – but humans must curate them. It's an emerging best practice to **label AI-generated contributions** clearly (e.g. include "AI-assisted: Claude" in the PR template or commit trailer) [32] [33] to alert reviewers to be extra vigilant. PRs should be kept modest in size; if Claude's output is too large, break it into multiple PRs to ease review. Tools like GitHub's CLI (`gh`) can be installed so Claude opens PRs and issues for you [34] – but always require human approval before merge. A *Claude-first Git playbook* (Section G) provides branch naming conventions, commit message guidelines, AI-specific PR checklist items, and advice on managing AI-authored code (including rollback strategies and how to avoid long-running branches diverging from main).
- **Project Configuration via `.claude/` Directory:** Claude Code is highly configurable. In a mature setup, the repository's `.claude/` directory serves as the command center for AI settings, customizations, and extensions. Key files include: `settings.json` – the project-wide config (checked into Git) controlling permissions, tool access, environment setup, and hooks (e.g. you can encode that Claude should always deny reading `.env` files here) [21]; `settings.local.json` – per-user overrides (not checked in) for local API keys or personal preferences; `commands/` – a library of custom slash-commands (predefined prompt templates for common tasks) [35]; `skills/` – project-specific Agent Skills packaged as folders with `SKILL.md` files that give Claude specialized knowledge or abilities (these load automatically when relevant, teaching Claude project-specific rules or workflows) [36] [37]; `agents/` – definitions of custom sub-agents with isolated contexts or permissions for delegated tasks (e.g. an agent that handles only database migration logic, with access to a DB tool but no write access

to app code); `rules/` – modular instruction files to enforce policies or styles (breaking up what might otherwise be a monolithic `CLAUDE.md` into maintainable pieces). Additionally, `CLAUDE.md` files can live at the repo root and in subdirectories – these provide scoped context that Claude automatically loads (e.g. the root `CLAUDE.md` might contain overall project guidelines, while `frontend/CLAUDE.md` contains UI-specific conventions) [13] [38]. Section H offers best practices for each of these, common pitfalls (e.g. avoid putting secrets in team settings, use `skills/` for complex guidance instead of bloating `CLAUDE.md`), and an example `.claude/` structure for a large project.

- **Commands vs Skills vs Agents – Decision Matrix:** In Claude Code, *slash commands*, *skills*, and *sub-agents* are complementary tools – choosing the right one is part of vibe engineering. **Commands** are manually invoked and best for deterministic, frequently used prompts or sequences (think of them as saved macros a developer triggers, like `/deploy staging` or `/port-module X`) – they give direct control to the human [39]. **Skills** are automatically triggered by Claude when certain queries arise; they inject specialized expertise or standards without the user explicitly asking (for example, a "Review PR" skill that *always* applies the company's code review checklist whenever Claude is asked to review code) [36] [37]. **Agents (Sub-agents)** are like spawning a separate AI pair for a subtask – they run in parallel or isolation, with their own context and tool permissions, and are explicitly invoked when needed (or suggested by Claude for complex problems) [40] [41]. Overusing one mechanism can cause issues – e.g. too many sub-agents might fragment context or complicate coordination, while too many auto-skills could make Claude's behavior unpredictable if it's firing off a dozen "hidden" rules. The playbook (Section I) includes a clear table and guidance to help decide which mechanism to use for a given scenario, ensuring a balanced approach that maximizes Claude's help while keeping the process understandable and controllable.
- **Continuous Evaluation and Metrics:** Adopting Claude Code at scale requires measuring its impact and adjusting. Section J proposes a lightweight evaluation framework: track **speed** (lead time per task, from spec to merged PR, and how many prompt iterations Claude needed), **quality** (defect rates, test pass rates, complexity of code changes, adherence to style), **review effort** (time reviewers spend, number of review comments on AI-generated code – initially this may spike as reviewers catch AI quirks [42] [43], but should improve as prompts/skills are refined), **reliability** (frequency of Claude errors or needing human re-dos, tool permission denials, etc.), and **safety** (any incidents like attempted unauthorized actions or omissions of required checks). A two-week pilot scorecard can be used – e.g. measure baseline metrics for two weeks, then metrics with Claude assisting, and review outcomes with the team. The goal is to ensure Claude is accelerating development *without* increasing bugs or burnout. Common failure modes (like Claude's overconfidence or shallow verification of its own work) are documented with warning signs and mitigation steps in a Failure Modes & Recovery table for quick reference. In sum, by treating AI as an **auditable partner** – with plans, checklists, tests, and reviews – teams can reap significant productivity gains from Claude Code while maintaining the engineering rigor and product stability that professional software development demands.

A. What Actually Works with Claude Code (2025)

Claude Code has been battle-tested in various scenarios by end-2025. **Real teams have converged on a set of workflows that consistently deliver value**, avoiding the pitfalls of naive "let it code everything" approaches. This section outlines proven patterns and how they differ across maintenance, rewrite, and new development, including when to use each, and how to detect and correct issues early.

Proven Claude Code Workflows in Practice

- **Explore → Plan → Code → Commit:** This is a foundational workflow adopted by many Claude users. Instead of diving straight into coding, the human first asks Claude to *explore* or read relevant parts of the codebase, then to come up with a plan before writing any code [1] [2]. For example, you might say: "Read the file that handles logging, but don't fix anything yet." Claude will retrieve the content (using its tools like `Read` or `Grep`) and summarize or analyze it. Next, you explicitly ask for a plan: "Now *think* and draft a step-by-step plan to improve the logging without breaking existing behavior." Using keywords like "think hard" or "ultrathink" can trigger Claude's extended reasoning mode for a thorough plan [2]. Once the plan looks sound (this is a checkpoint – the human reviews or even saves the plan as a file or GitHub issue [44]), you greenlight Claude to implement. Claude then writes code following the plan, often verifying each step as it goes (you can instruct it to double-check assumptions while coding) [45]. Finally, have Claude commit the changes with an explanatory message and even open a Pull Request [46]. This workflow significantly improves reliability on complex tasks: *when Claude jumps straight to coding without research or planning, it may overlook corner cases or design constraints*, but the plan-first approach reduces that risk [3]. **Failure modes:** If Claude's plan is too vague or overly ambitious ("Refactor the entire module in one go"), that's an early warning – the human should intervene and ask for a more incremental plan [28]. If Claude starts coding without adhering to the plan, pause and reiterate the agreed steps. **Recovery:** You can always roll back (e.g. use `git revert` on a commit) and go back to the planning stage if needed, clarifying any missed requirements. Saving the plan externally (in an issue or file) also means if implementation goes awry, you can reset the conversation and tell Claude to reconsider the plan with adjustments, without starting from zero.

- **Test-Driven Development (TDD) with Claude:** Anthropic engineers highlight a *"write tests, then code"* workflow as especially effective [4] [5]. Here, you first ask Claude to generate unit tests or integration tests for the desired behavior before any implementation is written. For instance: "Create unit tests for the UserRegistration service covering X, Y, Z conditions. We'll do TDD, *do not write the service code yet*." Claude will produce tests (it's important to explicitly mention that the functionality might not exist yet, so it doesn't try to stub it) [47]. You then run those tests (Claude can execute them if allowed) and confirm they fail – validating that the tests indeed capture missing functionality or a bug. Once you have failing tests, instruct Claude to implement the code to make them pass, with a rule that it should **not** modify the tests [5]. Claude will iteratively write code and run tests, refining until green. This leverages Claude's strength in iterative improvement: it has a clear success criterion (all tests passing) and will keep tweaking its solution until it meets it [48]. Anthropic notes this yields higher-quality fixes and features, as the AI is essentially forced to consider edge cases captured in tests [49]. **Failure modes:** If tests are too broad or not precise, Claude might "overfit" the implementation – i.e. code that technically passes the tests but isn't logically sound for other cases [50]. An early warning is if Claude's code handles only the exact inputs of the tests and nothing more. Mitigate by asking Claude (or a *verification subagent*) to double-check the implementation for generality beyond the tests [50]. Another failure mode is if Claude struggles to satisfy a complex test scenario; it might loop or produce lengthy attempts. If so, consider simplifying the task: maybe break the feature into smaller pieces and do TDD for each, or provide a partial implementation hint. **Recovery:** You can always manually adjust a test or provide Claude with a hint ("Notice that the calculation should use formula X in edge case Y") and let it continue. The TDD workflow shines especially in legacy parity work – you can write tests reflecting legacy behavior, then have Claude implement the new system to match, greatly reducing regressions.

- **Visual Feedback Loop (UI/UX workflows):** For front-end changes or any task with a visual component, Claude can work with images. A popular pattern is to provide Claude with a *design mock or screenshot*, have it implement code, then produce a screenshot of the result, compare it to the target, and repeat until it matches [51] [52]. For example, if porting a UI component from WinForms to a web UI, you might supply a screenshot of the legacy UI and say "replicate this in the new Electron app." Using an MCP server like Puppeteer (to control a headless browser) Claude can render the new component and screenshot it [53]; if not, you can manually give it screenshots. Claude will refine the UI code and layout in iterations [52]. **When it works:** This "see and adjust" loop yields rapid convergence to pixel-perfect results, as Claude can visually detect discrepancies just like a human (given the right tools) and correct them [54]. It's especially useful in modern feature dev when you have a design to implement. **Failure modes:** Without visual tools, Claude is limited to reasoning about UI by code and might miss subtle styling issues. If you notice Claude making repeated small guesses ("Try adjusting padding... try adjusting again"), it may be groping without real feedback. Early warning is lots of iterative CSS tweaks in output. **Recovery:** invest in the tooling – e.g. set up a Puppeteer MCP plugin or have Claude run the app locally and snapshot it [53]. In absence of that, you might step in and manually describe the remaining differences to Claude ("The button is 10px too low and colored red instead of blue in your output"). Also, enforce a cutoff of iterations – if after 2-3 tries it's not right, regroup and identify if a different approach or more context (like the CSS framework documentation) is needed.

- **"Safe YOLO" Mode for Bulk Tasks:** Claude Code has a headless autonomous mode (`--dangerously-skip-permissions`) which some call **YOLO mode** [7]. This lets Claude run without pausing for any confirmations – essentially doing a task start-to-finish on its own. Teams use this for low-risk, repetitive tasks: e.g. apply a formatting fix across hundreds of files, or update license headers in every source file. **When it works:** In a controlled sandbox (e.g. a disposable branch or a Docker container), this can be a huge time-saver – Claude will churn through the task quickly [7]. Anthropic even provides a reference Docker setup for safe YOLO runs [55]. It's critical to only use this mode for tasks that are easy to review after (like lint fixes or boilerplate generation) [7]. **Failure modes:** The obvious risk is Claude doing something destructive or off-track when unchecked [56]. If you run it on your real environment, a prompt injection or a misunderstood command could cause damage (hence *"dangerous"* skip permissions). Early warning of trouble is hard in YOLO mode (since by design it won't ask), but you might monitor system changes (files changed unexpectedly, etc.). **Recovery:** The best recovery is prevention: run YOLO only in isolated environments and with version control so you can discard unwanted changes [56]. If Claude's unattended run produced a bad result, simply reset to the last good commit. In practice, many developers mitigate risk by limiting scope (e.g. run YOLO only on a specific folder) and by inspecting diff output periodically. For example, run it to generate boilerplate, then *not* merge those commits until a human has skimmed through them. This mode is never used for complex logic changes – only mechanical, easily verifiable tasks (and often with tests afterward to confirm nothing broke).

- **Codebase Q&A (Interactive Code Comprehension):** Many teams find Claude invaluable for understanding a large or unfamiliar codebase – essentially as an AI pair programmer you can ask questions. You can ask, for instance, "How does logging work in this repository?" or "Where is the user permission check implemented for deleting records?" and Claude will **agentically search** the code (using `Grep`, `Read`, etc.) and assemble an answer [57] [58]. Anthropic engineers report that onboarding new hires via Claude Q&A can significantly cut down ramp-up time [58]. This works by leveraging Claude's 100k-token context to pull in relevant snippets on the fly, rather than expecting any one person to know all the pieces. **When it works:** This is extremely effective for legacy maintenance scenarios, where understanding "why is this like

that?" is 80% of the battle. Claude can sift commit history (`git blame` queries), read design docs if pointed to them, and cross-reference usage of functions. It's like a smarter grep: instead of just matching text, it actually interprets the code to answer conceptual questions. **Failure modes:** If the codebase is very large or loosely connected, Claude's search might miss things (users note that Claude's built-in search is essentially a grep and can fail if queries aren't precise [8]). A warning sign is Claude answering confidently but incorrectly about code – it might have only read part of the context. Another is if it frequently says "I didn't find anything" – the query might need refining. **Recovery:** In such cases, you should guide Claude: provide file names or broader hints ("Search for 'permission' in the `services/` directory"). If answers seem off, double-check by manually opening the referenced files or asking follow-ups ("Are you sure? Could it be in module X?"). Splitting the query among subagents can help too – e.g. one subagent maps the code structure while another looks for a specific pattern. Overall, treat Claude's answers as a helpful first pass – verify critical details from the source. When used well, this Q&A approach offloads a lot of tedious code reading while still keeping the engineer in charge of confirmation.

- **Git Automation and Integration:** Claude Code can interface with Git very effectively, turning natural language into version control operations. Proven uses include: having Claude **compose commit messages** (it will look at the `git diff` and recent commit history to draft a message that follows team conventions [30] [59]), performing interactive rebases or merges, and even creating PRs with descriptions. Many experienced users let Claude handle "90%+ of git interactions" by simply describing the intent [60] [61] – e.g. "Commit these changes with message 'fix: address null pointer in payment processing'" or "Open a PR for branch `feature/123` against `main` with title 'Feature 123: new payment flow'". Claude knows the shorthand ("pr" means create pull request) and uses the GitHub CLI or API if configured [62]. **When it works:** This is very useful in modern development where process (commits, PRs, resolving merge conflicts) can eat time. Claude is quite adept at understanding merge conflict markers and can resolve conflicts if you tell it which side to prioritize. It's also great for digging through git history: e.g. "Why was function X changed in v2.1?" and Claude will search commit messages/diffs [63]. **Failure modes:** One risk is giving Claude too much leeway – e.g. blindly letting it commit and push changes that haven't been reviewed. Another is that if multiple complex git operations are needed (like rebasing a branch with many conflicts), Claude might get confused or do partial fixes. Early warning signs are Claude pausing to ask for clarification on a conflict, or if it posts an overly long diff in the chat (maybe misinterpreting how to resolve multiple files). **Recovery:** Best practice is to tackle one git operation at a time. If a rebase is non-trivial, you can do it stepwise: ask Claude to list conflicting files, fix them one by one, then continue. Always double-check the `git log` or GitHub PR after Claude's actions. If something went wrong (e.g. commit message is incorrect or commit was made to wrong branch), you can fix it manually or instruct Claude to do so (like "Oops, amend the last commit's message to ..."). In summary, letting Claude handle routine git tasks is a big productivity win, as long as you keep an eye on the outcomes – treat it like a junior dev using git and verify it didn't do something wild (Claude generally will not, especially if guardrails are set to prevent, say, direct pushes to protected branches).

Legacy Maintenance vs. Rewrite Parity vs. New Feature Development

Different project modes call for different "vibes" in vibe engineering. Claude Code workflows are tuned slightly differently in each context:

- **Legacy Maintenance:** This involves working in an older, possibly messy codebase where the goal is to *fix bugs or make incremental improvements without breaking things*. What works here is emphasizing **comprehension and caution**. Users often lean heavily on Codebase Q&A and the

plan-first workflow. For example, a proven pattern is: *have Claude analyze a legacy module and generate a* `how-it-works.md` *before changing anything* [64]. This is effectively using Claude to document the legacy code's intent and quirks (including function signatures, global state, etc.) [64]. Maintenance tasks benefit from *small scope guardrails*: explicitly tell Claude which file or function to modify, and nothing beyond. If you ask Claude to fix a bug, you might say "Apply the minimal fix only in this function – do not refactor other parts of the module." Maintaining behavior is paramount, so tests are your safety net – it's common to augment the legacy code with new tests (Claude can help write them) to pin down existing behavior before any change. Another successful strategy is using **subagents for deep dives**: as one engineer noted, they spawn a subagent to do a "full analysis" of a problematic component (which may be thousands of lines), then only bring back a summary of potential issues to the main session [17] [18]. This way, the main context isn't flooded, and you have a distilled view (e.g. "Subagent reports that function `computeTax()` has side effects that might affect global state"). **Early warning signals in maintenance mode:** Claude proposing changes that alter function signatures or data models is a red flag – it might break other parts (the user should have told it not to, or it's hallucinating an optimization). Another warning is if Claude's bugfix suggestion comes too quick without analyzing the code – it might be a *surface-level fix* that ignores root cause (as one engineer experienced, the AI suggested a null-check to fix a crash, missing the deeper issue of an unexpected API response [65] [66]). **Recovery:** When in doubt, revert and force Claude (or yourself) to do more analysis. You can ask Claude "Could there be side effects? How does this module interact with others?" or use git blame through Claude to see history of that code (maybe it was written a certain way for a reason). In maintenance, it's often worth spending an extra hour on Claude-assisted reading and test-writing to avoid a regression that costs days later. In summary, for legacy maintenance Claude is used as a powerful reading and reasoning assistant, and a careful code changer under tight supervision.

- **Rewrite Parity Work:** Here, you have a legacy system and a new system in parallel, and you're gradually porting features to the new system (the reference scenario given, e.g. rewriting a 1M LOC C# app into Electron, running side-by-side). The key is *behavioral parity* – the new code must match the old system's behavior unless a divergence is explicitly intended. **What works:** Start by importing as much context about legacy behavior as possible into Claude's realm. Teams create "legacy facts" documents (or populate `CLAUDE.md`) with things like: important business rules in the old system, known quirks that must persist, and boundaries not to cross (e.g. "core logic remains in the C# service, do not replicate it in JS"). Claude can help by reading legacy code and summarizing business logic that needs to be carried over [67] [68]. A recommended approach is to use **skills or subagents specialized for parity checking**. For example, one might build a `/parity-check` command or a skill that, given a function in legacy and in the new code, runs both with sample inputs and compares outputs (using a test harness) – Claude can automate such comparisons. In planning a port, Claude Code's *planning mode* is extremely useful: you can ask "Analyze the legacy module X and propose how to implement it in the new system" [69] [70]. It will consider architecture differences and often suggest a stepwise strategy (e.g. first extract a pure logic core, then implement that core in new environment, then gradually replace calls). **When it works:** Claude's large context lets it consider large swaths of legacy code (even if not all at once, it can chunk through via subagents) and reason about equivalences. It's adept at translating between languages – e.g., converting a .NET/C# concept to an Electron/JavaScript analog – while keeping function input/output behavior consistent [71] [72]. This translation ability (with awareness of idioms in each language) is cited as a Claude strength in modernization [73] [74]. **Failure modes:** The biggest risk is *architectural drift* – the new implementation unintentionally diverges from the legacy system's proven design. Claude might, for instance, "improve" something (like simplifying a complex validation logic) not realizing that complexity handled a corner case. Early warnings of drift include Claude suggesting different data flows ("In

the new app we don't need this check") or using new libraries that weren't in the plan. Another issue is *context overload*: trying to have Claude port too much in one go. If Claude's outputs start getting inconsistent or it forgets earlier parts of the plan (signs of context window limits), it means the chunk is too big. **Recovery:** Enforce guardrails that **parity is king** – e.g. instruct "The new implementation must produce identical results given the same inputs (except where specified). If unsure of any legacy behavior, pause and ask." Also leverage tests: for parity, one strategy is to use **record-replay testing** – run the legacy system on a set of inputs to record outputs, then have Claude generate a test suite in the new system using those input-output pairs to validate parity. If tests reveal differences, those are either bugs in the new code or intentional differences that need sign-off and documentation. Document any intentional divergence (preferably automatically: Claude can append a section in a changelog or migration doc stating "Differences: In new system, we round interest rate to 5 decimals whereas legacy did 4, per new requirement."). In code reviews for ported features, humans should double-check that no hidden assumptions were lost – a good practice is to involve a *domain expert* on the legacy system to review the AI-authored code for subtle deviations. In short, Claude can accelerate parity work by handling rote translation and even identifying where behaviors live in legacy code, but the human must continuously verify that "no user will notice a difference" unless it was planned.

- **Modern Feature Development:** In a greenfield or modern portion of the project, where you're adding entirely new features (with less legacy baggage), Claude's role shifts to more of a creative partner and rapid executor. **What works:** In these cases, *generation and iteration speed* can be prioritized a bit more. Developers often use the test-first or plan-and-build workflows to quickly scaffold new modules. Since the new code doesn't have to mirror old code, Claude has more freedom to propose implementations – you still give it architectural guidelines ("use our standard MVC structure" etc.), but you might lean on its suggestions for how to design something. It's known that Claude and similar models shine at producing boilerplate and common patterns quickly [75]. For example, "Claude, create a new REST endpoint for updating user profiles" might result in a near-ready implementation following the project's conventions (especially if those conventions were noted in `CLAUDE.md`). Another common pattern is using Claude to handle *peripheral code* while you handle core logic – e.g. you write the tricky algorithm, and ask Claude to write all the mundane wiring (controllers, DTO classes, etc.). **Failure modes:** The danger in new dev is *over-reliance on AI where design thinking is needed*. If a team just accepts whatever structure Claude generates, it might not align with the intended design or could introduce technical debt. Early signs include Claude making design decisions unprompted ("I created a base class for these" when that wasn't requested). Also, without legacy constraints, Claude might be more likely to wander from best practices if not guided – e.g. using an outdated library it saw in training data. **Recovery:** Keep the human-led design process: use Claude for options brainstorming ("Give me 3 design approaches for feature X with pros/cons"), but then the team chooses and instructs Claude accordingly. Modern development with Claude should still include guardrails like code review, but reviewers will focus on ensuring the new code meets acceptance criteria and is maintainable. Because no legacy behavior tells you if it's correct, *you must rely on thorough testing and code review*. Fortunately, Claude can also assist in writing those tests and even reviewing its own code (one practice: after Claude writes the feature, start a fresh Claude instance and have it do a "code review" of the diff as if it were a reviewer – it often catches its own lapses or suggests improvements in clarity). In summary, for new features Claude is a turbocharged implementer that can save time on boilerplate and provide ideas, but the engineering team must still steer the architecture and ensure quality, treating Claude's output as a first draft to refine.

Early Warning Signals & Recovery Strategies

Across all contexts, being attuned to early warning signs from Claude's behavior can save you from costly mistakes. Some cross-cutting signals:

- **Silent Assumptions:** If Claude's response includes functionality or decisions that were never discussed (e.g., it adds a new config setting "for future extensibility" or assumes a business rule), this is a sign the model is filling in blanks that you, as the engineer, should confirm or reject. Recovery: immediately ask Claude *why* it did that – often, it will explain its reasoning. Then either update the instruction ("That assumption is wrong, the rule is actually…") or adjust the plan to explicitly cover that scenario. Encouraging Claude to be **direct and honest, and to ask rather than guess** is something you can bake into `CLAUDE.md` as a rule [76] [20].

- **Over-Broad Edits:** If a diff shows Claude modifying areas you didn't intend (e.g. it refactors a helper function that *wasn't* part of the request), that's a guardrail breach. It might indicate Claude is trying to handle something proactively (sometimes useful, sometimes not). Recovery: Use *scope guardrails* – remind Claude to limit changes to the specified scope, and perhaps use the `/undo` or git revert on the extraneous changes. In future, you might tighten permissions (e.g. deny writes outside a certain directory during that task).

- **Shallow Verification:** Claude says "All done, tests passed" but maybe you notice it only ran one test or it didn't consider an edge case. If Claude ever outputs "I think this is good to go" very quickly, be cautious – verify by running a full test suite yourself or instructing it to run extended checks (e.g. integration tests, lint, etc.). Using a second instance of Claude for verification (as described in the multi-Claude workflow [77]) is a powerful technique – one Claude generates code, another independently assesses it. If that second opinion flags issues, that's an early catch and you can cycle back to fix them.

- **Context Saturation:** As conversations go long, Claude might start forgetting details or confusing names (context dilution). If responses get less coherent or begin re-asking things you told it before, it's time to clear context or split into sub-tasks. One strategy: periodically summarize the state in a scratchpad file and use `/clear` to reset Claude with just that summary, thus wiping out clutter. This keeps the session fresh while retaining essential info.

- **Model Overconfidence:** Claude might sometimes insist on a solution that you suspect is wrong (LLMs can be confidently incorrect). If you sense this – e.g. it's not listening to a correction or it's glossing over a failing test by saying "the failure is trivial" – slow down the process. Ask it to explain its solution step by step. Often the act of explaining will surface the flaw (or at least show you its logic so you can pinpoint the error). If not, it may be time to re-plan from scratch, as the conversation might have gone in circles. Always remember that *the human is the ultimate backstop* – if something doesn't smell right, you have the full freedom to stop Claude and take over or re-prompt.

Section 4 (Failure Modes & Recovery) condenses these and other common issues into a handy table for quick reference.

Grounding these insights in real usage: official Anthropic guidance and experienced engineers' reports all emphasize **the human-in-the-loop approach**. Claude Code is incredibly powerful, but it doesn't eliminate the need for senior engineering judgment [78] [79]. The practices that "actually work" are those that treat Claude as a junior partner who can think fast and execute diligently, *under the close guidance*

of a human expert. By following the proven workflows above and being alert to early warning signs, teams in 2025 have been successful in using Claude Code to dramatically accelerate development while maintaining (or improving) software quality.

B. Context Management (Core Claude Competency)

Claude Code's ability to handle large codebases hinges on managing context wisely. With a ~100K token window, Claude can theoretically "read" a ton of code – but feeding it everything is inefficient and can even confuse it. **Effective context management** means providing just the right information at the right time, structuring how Claude sees the project, and keeping irrelevant or sensitive data out of its view. This section details Claude-optimized context strategies, including defining a working set, progressive disclosure, context budgeting (what to include vs exclude), structured handoffs between phases, and preventing context poisoning in long sessions. We also provide a *Claude Code Pre-Flight Context Checklist* as a deliverable to use before kicking off any significant task with Claude.

Repository Mapping and Defining the "Working Set"

Don't load the entire monorepo into Claude's head. Instead, define a *working set*: the files, modules, or sections of the codebase relevant to your current task. Claude Code is designed to help with this – it has tools like `Glob` and `Grep` to find and pull in files on demand [80] [81]. Rather than preemptively giving Claude all code, you can instruct it in natural language: "Open the file that handles user login" or "Find references to `calculateTax` across the project," and it will fetch those as needed [82]. In practice, a human often has an idea where the relevant bits are (if not, they might do a quick manual search or ask Claude Code's global search). By iteratively building the working set – starting from a couple of key files, then expanding if needed – you avoid overloading Claude with unrelated context. **Why is this important?** Because the context window, while large, is still finite. If you stuff it with everything, you leave less room for Claude's own thought and the conversation, and you risk diluting focus (the signal-to-noise ratio drops).

For example, if debugging a payment calculation issue, load the `PaymentService` code and perhaps the `TaxCalculator` module – don't also load UI code or config files unless the bug might relate to them. You can always bring in more later if needed. This targeted approach mimics how a human works: you don't read the entire codebase top-to-bottom for one bug, you pinpoint relevant areas. Claude can assist by doing broad searches itself, but *you often guide those searches*. One Reddit user noted: *"I've found it easier to guide it to the relevant files myself"* for large, loosely connected codebases [83]. This ensures Claude isn't blindly grepping thousands of files, which is time-consuming and can incur large token costs.

Checklist item: Identify and enumerate key files/folders for the task at hand. Use phrasing like "We will be working on `module X` and `module Y` primarily – focus context on those." This hints Claude to pay extra attention there. You can also explicitly tell Claude *not* to wander: e.g. "Don't modify or read unrelated modules while doing this" (which addresses both context focus and guardrails).

Progressive Disclosure (Tight → Expand)

Claude Code (and the Agent Skills system in particular) embraces *progressive disclosure* – providing information in layers of detail as needed [10] [11]. You can apply this concept generally in prompts. Start with a **tight context**: give Claude a summary or high-level description of the problem, maybe a couple of crucial code snippets, and pose the question or task. Let Claude identify if it needs more info. It might come back with: "I need to see how function X is implemented" or it might guess – if it guesses, and you

suspect it needs more, you then feed the next chunk. Essentially, you're simulating a Socratic process: *broad question → request for detail → more detail provided → so on*. This way, you don't upfront dump 50 files "just in case." Instead, you disclose progressively: first general info, then specifics as Claude asks or as the plan requires.

Claude's built-in behavior with Skills exemplifies this: it starts with just skill names and descriptions loaded (to know if they might be relevant) and only loads full skill content when needed [84] [11]. You can mirror that manually. For instance, when porting a feature, you might initially just tell Claude: "We have a legacy method that does X. Plan how to implement it in the new system." If Claude's plan seems off or it asks for it, then you provide the actual legacy code (perhaps a link or by using `Read(file)` yourself). This keeps the conversation efficient and relevant. It's also good for cost management – token usage stays lower, which in long rewrite projects (potentially months long) can save significant usage.

Checklist item: *Don't throw the kitchen sink in.* Start each session or task with the minimum necessary context. You can say: "Claude, I will gradually give you info. First, here's the overview…". Use the `#` system command to add important notes to `CLAUDE.md` for persistence (like reminding it of a core principle, see below), but resist loading full files unless needed. Claude's pretty good at telling you when it lacks detail. If it hallucinates or makes assumptions, that's a prompt to give it the specific snippet it's missing (progressively expanding context).

Context Budgets: What *Never* Goes In (or Rarely)

Some things should almost never be put in Claude's context, either because they waste budget or risk contamination of the model's output:

- **Generated or Derived Files:** This includes minified assets, compiled binaries, machine-generated code, lockfiles with thousands of lines of dependencies, etc. They are rarely helpful for reasoning, and including them could even cause Claude to treat them as source (leading to nonsense edits on generated code). If Claude needs to regenerate something like that, it's better to run a tool (for example, don't feed a large `yarn.lock` – instead run `yarn install` outside or let Claude run it if truly needed). **Never feed Claude large diffs or artifacts just for the sake of it.** Use `permissions.deny` or ignore patterns to keep these out; e.g. instruct it not to open anything in `node_modules/` or `dist/` directories.

- **Secrets and Credentials:** This might seem obvious, but ensure environment keys, password files, etc., are excluded. Claude Code supports a deny list in settings – by default `.env` and similar should be off-limits [21]. Even if not for security, presenting a .env with dozens of config vars can distract the model ("Does the AI need to know the AWS key? Probably not."). If your conversation requires referencing a setting, just mention it abstractly ("the DB connection string") rather than providing the actual secret.

- **Entire Large Files Unless Focused:** If a file is thousands of lines, decide if you need the entire thing or just relevant parts. Often you might say "open file X" and Claude will dutifully load the whole thing into context, which might eat a chunk of your window. If the task is to refactor one function in that file, consider using Claude's search to pull just that function (Claude can `grep` by function name if it's well-defined, or you can copy-paste that segment yourself). There is a trade-off – sometimes surrounding context in the file matters, but you can gauge it. A neat trick is using **scratchpad or checklist files**: have Claude write key excerpts or a summary of a giant file into a Markdown list (thus condensing what's needed) [85] [86].

- **Previous Conversation Fluff:** If you've been working with Claude in a session for a while, earlier parts of the conversation might become irrelevant (or even a liability if they contain outdated info from before you changed direction). Using `/clear` resets Claude's short-term memory [14]. You typically do this between distinct tasks or whenever you feel context is getting "stale." One should *not* feel obliged to carry the whole conversational history; unlike a chat with a human, it doesn't offend Claude if you wipe its memory of something and re-provide only the relevant facts. In long-running rewrite projects, engineers often break the work into epochs: e.g., "Phase 1 complete. Now new session for Phase 2." They then feed in only the high-level outcomes of Phase 1 (like an updated design doc or updated `CLAUDE.md`) rather than the entire verbose trail of how Phase 1 was done.

In summary, allocate your context tokens like a budget: spend them on *instructions and code that matter*. Keep out anything that is noise or could mislead. For instance, including a large log dump might be necessary for debugging, but then *exclude it* when moving on to coding the fix – don't let that log consume context in later steps.

Structured Handoffs: Briefs, Plans, Diffs, Test Output, Logs

A hallmark of vibe engineering is structuring the interaction into phases, and passing the right information to Claude at each phase in a deliberate format:

- **Briefs:** Start tasks with a *structured brief*. This could be a bullet list or numbered list outlining the task, requirements, constraints, and success criteria. Rather than a rambling paragraph, a list is easier for Claude to systematically address. For example: "1. We need to add a new field `X` to the report. 2. Must also update the API to return this field. 3. The field is derived as … 4. Do not break existing fields." This acts like a mini-spec in the prompt. Claude will often follow the numbering in its response.

- **Plans:** As discussed, get Claude to output a plan before coding. Treat this plan as a handoff between *"design mode"* and *"implementation mode."* You can even have Claude output the plan into a file (like `plan.md`) which you then review. If you like, you reset context (clear conversation) and then input: "Based on the approved plan below, implement the solution" and paste in the plan content. This ensures extraneous context from earlier Q&A or brainstorming doesn't carry over – you're giving it a clean slate plus the plan. This is a powerful technique to *modularize the conversation*.

- **Diffs:** When Claude makes changes, prefer it to output diffs or at least summaries of changes, rather than full rewritten files (except for new files). The CLI by default has it show a diff [87]. This structured output is easier to review and keeps context tight (a diff inherently has only the changed lines plus some context). Encourage diff view: e.g. "Show the changes as a unified diff." Reviewing diffs is a natural human skill – you'll catch if it edited something wildly outside scope.

- **Test Output and Logs:** If you have Claude run tests or some analysis, it's helpful to direct that output somewhere structured. For example, "Claude, run the test suite and put any failures into `test_failures.md`" (Claude can use shell to run tests, then capture output). If you just let it spit test logs into the conversation, that could be hundreds of lines of noise possibly overwhelming context or truncating important parts. By using files or summarizing logs, you hand off only what's needed. An approach: *"Summarize the log – what were the key errors?"* rather than pasting the entire log. Claude can grep logs for "ERROR" and return a concise view [88] [89].

Similarly, for a failing test, you might ask Claude to output just the assertion message and stack trace relevant, not every test case result.

- **Generated Files & Vendor Data:** Sometimes Claude might need data from generated files (e.g. a compiled output to verify something) – prefer to have it run a script to get the info rather than reading the file. For vendor libraries, ask it to use documentation or skip reading library code unless the task is to debug the library. If it does read a huge vendor file, that's likely context poisoning (the model might latch onto irrelevant details). Recognize those situations and steer away ("That's library code, Claude. Don't delve into that – assume it works as documented.")

- **Checklists & Scratchpads:** The Anthropic best practices suggest using Markdown checklists as a sort of scratchpad for large, multi-step tasks [85]. For instance, if you have 20 lint errors to fix, have Claude write them all out as a checklist first, then tackle them one by one, checking off each item as it's resolved [90] [86]. This keeps progress clear and context focused on the immediate sub-task. It also serves as documentation of what's been done (the checked-off list can be attached to the PR or kept for records). Encourage Claude to maintain such scratch files for complex workflows; these files themselves become part of context (in a helpful way) but can be trimmed or removed when not needed.

Handling Long-Running Sessions & Preventing Context Poisoning

In long sessions (e.g. an hours-long debug or multi-day coding with the same session open), a lot of extraneous stuff can accumulate in the conversation: previous code versions, back-and-forth dialog, irrelevant paths explored and abandoned. This can *"poison"* context, meaning the model might draw on now-incorrect intermediate conclusions or waste time rehashing old points.

The remedy is to regularly prune and refresh context. Two main tools: the `/clear` command and use of multiple Claude instances or subagents.

- **Frequent /clear:** It might feel counterintuitive, but clearing the conversation history while retaining the essential info in `CLAUDE.md` or notes leads to better performance on very long tasks [14]. As Anthropic advises: *"Use `/clear` frequently between tasks to reset the context window."* [14] If you just solved one sub-problem (say you fixed part A of a bug, and now need to fix part B which is related but somewhat separate), that's a great time to clear. Before clearing, maybe recap: "Summary: we fixed A by doing … Now we will address B." Put that summary in the prompt after clear, so Claude knows where to start. Clearing ensures the next exchanges aren't weighed down by the full history of solving A.

- **Isolated Subagents:** Instead of one marathon conversation, spawn subagents for distinct threads. Szymon Pacholski's example illustrated this: the main thread held the high-level context, but a subagent did a 50K-token deep analysis and returned only a 2K summary [17] [18]. This not only saved tokens but kept the main context *clean*. You can have multiple subagents – e.g. one subagent per major component of a rewrite – each with their own `CLAUDE.md` or specialized skills, and your main conversation just orchestrates between them. This compartmentalization prevents, say, the context about how the UI is implemented from crowding out context needed for the database migration logic.

- **Memory Files vs. Conversation Memory:** Prefer externalizing important info to files (`CLAUDE.md`, notes, etc.) rather than relying on conversation memory for hours. If Claude has deduced a crucial fact ("Function X must return null in scenario Y"), put that in a persistent place

(e.g. add it as a comment in code or an entry in a rules file). Then if you clear context or start tomorrow in a new session, that fact isn't lost. Conversely, anything that is *no longer true* (like a bug that's been fixed) – clear it out of the context by not carrying those messages forward.

Context poisoning can also refer to inadvertently biasing Claude with irrelevant data. For instance, if earlier in the conversation someone joked about something, that could still subtly influence later answers. Keeping a professional, focused tone and content scope avoids that. Also, be mindful of *order*: Claude tends to weigh recent messages more. So after a long series of interactions, if some outdated info from an hour ago is still in the log, even if contradicted later, it might confuse things. Clearing or summarizing effectively "forgets" the wrong turns.

Pre-Flight Context Checklist (Deliverable)

Before engaging Claude on a significant task (especially in a large codebase), run through this checklist to set up an optimal context:

1. **Define Task Scope:** Clearly identify the feature/bug and the parts of the codebase involved. Jot down the key modules, files, or functions that you believe are relevant. *(Example: "Fixing null pointer in OrderProcessing – likely involves OrderService.cs and PaymentGateway.cs.")*

2. **Update** `CLAUDE.md`**:** Ensure any high-level project info is up to date here. Add any new conventions or decisions relevant to the task. If this is a rewrite task, document legacy vs new mappings or important invariants here. Keep it concise [15] [91]. This file will be auto-loaded, giving Claude baseline knowledge.

3. **Set Guardrail Reminders:** In your initial prompt or via `CLAUDE.md`, restate guardrails that affect context. For example, "IMPORTANT: Do not read or write to any files in `/config/` (contains secrets) [21]. Focus only on `src/`." Also, "If something is unclear, ask me – do not assume undocumented behavior." These will help filter context usage.

4. **Select Interaction Mode:** Decide if you will use planning mode, a custom command, or just chat. If planning, perhaps use the `think` keyword to get Claude to deep-dive [2]. If using a slash command (like `/port-module` that you wrote), that command might automatically set context like which files to consider.

5. **Provide a Structured Brief:** Begin the session with a bullet list of the task requirements and context. Mention known relevant files (from step 1). Mention tests that should pass or issues to resolve. This brief orients Claude and acts as a mini-index of what's to come in context.

6. **Progressive Info Loading:** Feed only the top-level info first (e.g., "The issue is user ID is sometimes null. It occurs in OrderService."). Wait for Claude to request or identify further info needed. Then allow it to open files or you explicitly provide code snippets. Essentially, plan to *iteratively* load context, not dump all at once.

7. **Exclude Noise:** Before you let Claude loose searching, exclude patterns for known irrelevant files (you can add to `settings.json` `permissions.deny` for reads on certain globs like `build/**` or `docs/**` if not needed). Verify that no large binary or log files are in the working directory where Claude might accidentally stumble on them.

8. **Tooling Prep:** If logs or data will need analysis, consider preprocessing them. For example, if you have a 5MB log, maybe filter it to the last 100 lines that seem relevant or ask Claude to use `grep` on it for "ERROR". This keeps context lean.

9. **Subagent Setup (if needed):** Anticipate if a sub-problem might need isolation. For instance, if part of the task might require a deep static analysis, you could prepare a subagent (`.claude/agents/analysis.md`) in advance with limited tools (just read/search) to handle that. Not every task needs this, but for very complex ones, be ready to delegate to subagents to save context [92] [93].

10. **Double-Check Recency of Context:** If using Claude over multiple days, ensure you reload any files that changed since last session. Claude doesn't automatically know about external changes unless it reads them fresh. So use `Read` on a file to refresh its view. Outdated context (yesterday's version of a file) can be worse than none.

By following this checklist, you tee up Claude for success: it will be focused on the right things, not distracted by noise or clutter, and empowered to fetch details as needed. This approach takes a bit of upfront effort, but it pays off in more relevant answers and fewer missteps. As one user summarized, *"The context gathering consumes time and tokens, but you can optimize it through environment tuning."* [94] Setting context deliberately is exactly that tuning.

C. Memory & Persistence (Long-Running Rewrites)

When working on a long-running rewrite or any extended project, you inevitably accumulate knowledge, decisions, and patterns that the AI should remember from session to session. Unlike a human engineer who retains context over weeks, a new Claude Code session starts fresh – *unless* you explicitly persist information. Claude Code provides ways to persist memory (like `CLAUDE.md` files and other config), but it's crucial to decide **what goes into persistent memory and what stays ephemeral**. This section presents a Claude-compatible "memory model" for long projects, outlining what to persist (and how), what never to persist, the concept of "golden files," version-controlling AI memory, and strategies for resetting/forgetting between phases to avoid carrying stale or harmful context.

What to Persist in Claude's Memory

Think of persistent memory as the project's *collective brain* that Claude can tap into every time it starts. Good candidates for persistence include:

- **Core Architecture and Design Principles:** Document the high-level architecture (e.g., "Electron UI calls into C# service via gRPC, all business logic remains in service, UI only handles presentation"). If Claude knows these rules from the get-go, it will adhere to them. Put them in the root `CLAUDE.md` as bullet points or a brief section [15]. Similarly, any design patterns or important frameworks: e.g., "We use Repository pattern for data access. Services should not query the database directly." These guide Claude's solutions every time.

- **Coding Conventions and Standards:** This includes naming conventions, formatting, error handling patterns, etc. Claude can follow style guides if you provide them. For instance, include in `CLAUDE.md` that "All public methods must have XML doc comments" or "Use PascalCase for class names (C#) and camelCase for variables." These become second nature to Claude if

persisted [95] [96]. One team even emphasizes including examples ("Example: function names should be verbs like `CalculateTotal`") – concrete examples help the model.

- **Project-specific Commands and Workflows:** If your team has specific git workflow (e.g., "Feature branches should be named feature/xxx and we use Conventional Commits for messages"), note that. Claude can automatically generate commit messages following Conventional Commit format if it knows to [30] [31]. If there are common build or test commands, list those (Claude will then know to run `./gradlew test` versus `npm test`, etc.). At Anthropic, they put common commands in CLAUDE.md (e.g., "npm run build: Build the project") [95], and you can do the same for your project.

- **Known Bugs or Oddities to Work Around:** Every legacy system has quirks ("Module X must be initialized before Y or everything breaks" or "Ignore table Z – it's deprecated"). These are gold to persist. If Claude knows these up front, it won't inadvertently do the wrong thing. For instance, an entry: "IMPORTANT: The `LegacyAuth` class has known issues – do not modify it; changes are handled in the new AuthService." Without this, Claude might attempt a "fix" in LegacyAuth when porting auth logic, which you want to avoid. One can put a section "Legacy Warnings" in CLAUDE.md listing such things [97].

- **"Golden" Examples and Invariants:** This is worth elaboration. **Golden files or golden tests** are those that represent the correct, canonical behavior or format that must always be preserved. For example, you might have a golden JSON output of an API for a given input that must remain unchanged (like for regression tests). Or a snippet of code that implements a critical algorithm correctly. Persisting these in memory means Claude can reference them as needed. Perhaps you include a simplified version of a crucial algorithm in CLAUDE.md with a note "This is the correct approach for calculating interest – use this as reference whenever touching interest calculations." Then, even if months later you or Claude revisit that area, it has the reference. Golden tests can be stored as part of the test suite of course, but you could also instruct Claude to run those golden tests whenever relevant (semi-automating it via a skill or hook). In memory terms, you might list key invariants: "The output of Legacy and New for feature X must remain byte-for-byte identical for all existing use cases" – that's a high-level invariant to always remember.

- **Terminology and Domain Knowledge:** If your project has specific domain language (e.g., "policy" vs "contract", or certain acronyms), list them with explanations. Claude then uses the correct terms consistently and doesn't ask repeatedly what they mean.

These persisted pieces of information essentially form a *project knowledge base*. They should be concise, factual, and rarely change. By persisting them, you ensure Claude always starts with context that would take a human new joiner weeks to build – but be careful to keep it accurate and up-to-date.

What Must Never Persist

Equally important is deciding what **not** to carry over across sessions. Some examples:

- **Transient Debugging State:** If in one session you discovered, say, a certain value was null due to a specific dataset, that's not necessarily a general rule. Don't bake "Value X is always null" into memory – it might not hold later. In the heat of debugging, you might drop a note in conversation like "(we saw orderId was null for those test orders)". That should not go into

CLAUDE.md. Persist only conclusions that are permanently relevant (e.g., "orderId can be null if not set – we should handle it"), and even then phrase them as general rules or resolved issues.

- **Outdated Plans or Partially Correct Analyses:** If Claude made an assumption or a plan that turned out partly wrong, definitely don't persist it. For instance, maybe initially you thought module A called module B, but later found out it doesn't. Don't let the initial (wrong) assumption live on in memory. Update or remove it. This is why reviewing memory files is so important – you might have added something early on that's no longer true, and if you leave it, Claude might be misled next time. Treat persistent memory as you would documentation: it must be maintained to reflect reality.

- **Sensitive Information:** Even if accidentally seen by Claude in one session, don't propagate it. That means not writing secrets into CLAUDE.md or skill files. Also, anything your company policies forbid sharing with an AI should of course not be persisted (preferably not even given in the first place, but if it was ephemeral in one run, make sure it doesn't get saved in config or logs that Claude might load later).

- **Model's Own "Thoughts" or Chain-of-Thought:** Sometimes Claude might outline its reasoning or internal plans in a session (especially if you use the `think harder` mode). You generally *don't* want to carry over these internal reflections verbatim. They might contain speculative or context-specific info that doesn't generalize. For instance, Claude might say in a plan, "Probably we can drop support for X (not used anymore)." Unless you confirm and decide that as a fact, don't put that in memory. Use memory for confirmed decisions and facts, not maybes.

- **Code Artifacts That Could Drift:** If you persist a large chunk of code as "memory" (like including a whole interface definition in CLAUDE.md for reference), you risk divergence if the code changes in the repository but you forget to update CLAUDE.md. Then you have stale memory. It's safer to persist references or key aspects, but not long code blocks that are likely to change often. If you do include some code as example, keep an eye on it and update when the source changes (or better, include a note "see actual code in file X for latest" so you remind future self to have Claude read the latest file rather than trust the memory snippet).

In short, **persist conservatively**. Memory should be the stable bedrock of knowledge, not a dumping ground for every discovery. When in doubt, leave it out – you can always find it in git or wherever if needed, or ask Claude anew. The cost of a wrong persistent memory is confusion and misdirection later.

Golden Files and Invariants

We touched on golden files – here's how to leverage them with Claude:

- Identify critical files that should not change or whose content is the reference for correctness. For example, a canonical JSON output, a reference implementation of an algorithm, a set of official test vectors. Keep these in a known location (like a `tests/golden/` directory or a `docs/invariants.md`).

- You can instruct Claude in CLAUDE.md or a skill: "We have golden outputs in `tests/golden/`. Always run those tests or compare outputs to them when relevant." Or even create a slash command or hook that triggers automatically running golden tests after each major change. For example, a hook could be: after any `Edit` action in certain modules, execute a script that runs

`npm run test:goldens` and have Claude capture the result, alerting if a golden test failed. This automates persistence of invariants into the workflow.

- For coding style or patterns, "golden examples" might be as simple as including a perfect exemplar of how a certain pattern is implemented. If you have one beautifully written module that all others should mimic, consider linking it in memory (like: "Follow the style of `PaymentProcessor.cs` as an example of good implementation."). Claude can refer to it or you can explicitly say "open PaymentProcessor.cs for reference" when it's working on a similar module.

- **Version these golden artifacts.** Just as code is versioned, your memory of what's golden is too. If a requirement changes that invalidates a golden output, update the stored output and note the change in memory (e.g., in CLAUDE.md, change the invariant description). If an architectural rule changes ("we now allow Service A to call DB directly, after all"), remove or edit that rule in memory. Essentially, treat the memory files like living documentation and maintain them.

One strategy is to have a dedicated memory file for *invariants* or *rules*. For instance, `.claude/rules/invariants.md` that lists all these key things (with rationale). Because it's in `.claude/rules/`, Claude will load it as needed (or you can manually ensure it's referenced). This separates them from more general info and makes it easier to review them at a glance.

Versioning and Reviewing Memory

As emphasized, **memory is part of your project and should be managed with the same care as code**. That means:

- **Check memory files into source control** (except personal local bits). The `CLAUDE.md`, `.claude/rules/*.md`, `.claude/commands/*.md`, etc., that define behavior should be in the repo so everyone uses the same guidelines and so changes are tracked. Anthropic recommends checking in the main `CLAUDE.md` for team sharing [13]. Do the same for any shared memory constructs.

- **Pull Requests for memory changes:** If someone (whether human or AI) updates CLAUDE.md or adds a new skill or command, have those changes go through PR review like any code change. This ensures the team is aware and agrees. For example, if Claude Code itself, at your prompting, writes "Added rule: always use caching for API calls" to CLAUDE.md, that should be reviewed – maybe that's not always true, or maybe it needs refinement. Don't just accept memory changes blindly. Some teams label such PRs as "docs" or "ai-config" and treat them seriously because they will affect AI behavior broadly.

- **Periodic Memory Audits:** Over a long rewrite (say months), schedule a periodic check (maybe each sprint) where you or someone on the team reads through the persistent Claude memory documents to ensure they are still correct and relevant. Clean out stuff that's no longer needed (to keep context lean) and add any new lessons learned that should persist. This is analogous to a knowledge base grooming.

- **Syncing with Team Knowledge:** If architecture changes are made in design docs or Confluence pages outside of Claude, reflect them in the Claude memory too. Conversely, if Claude memory has details that would benefit new engineers, consider syncing that back to human-readable

> docs. Eventually, the CLAUDE.md might become a valuable artifact on its own – at Anthropic, engineers often include the CLAUDE.md changes in commits so others benefit [98].

By version-controlling and reviewing memory, you avoid "prompt drift" – where over time the instructions to the AI become contradictory or messy. Instead, you have an evolving but coherent set of instructions.

Reset/Forget Strategies Between Phases

During a long rewrite, there are natural phase boundaries: maybe you finished migrating module A, and next you tackle module B, which is quite separate. It can be wise at those junctures to **reset Claude's context entirely** – effectively treating the next phase like a fresh project start (except of course you have all your persistent memory loaded to bring it up to speed).

What does a reset entail? - Certainly using `/clear` or just restarting the CLI so you have a new session. - Possibly unloading or disabling certain skills or rules if they were only relevant to the previous phase. For example, if you had a skill specifically to help port module A, you might now remove or disable it so it doesn't accidentally trigger during module B work. - Flushing any conversation-specific context that was carried (some might be in `.claude/memory.json` or similar if at all; by default Claude Code doesn't persist conversation history between runs unless you saved it). - Ensuring that any decisions made in phase A that affect phase B *are* captured in persistent form. E.g., if in phase A you decided on a new data model that phase B will use, put that in CLAUDE.md or an architecture notes file so the new session knows.

Another scenario for resets is if things go awry. Let's say the conversation got confused or Claude is repeatedly getting something wrong likely due to earlier context tangling. Don't hesitate to break the cycle by wiping context (and maybe copying the essential instructions into a fresh prompt). Claude Code even allows starting multiple instances so you can do this without closing the old one – e.g., "Claude1" got stuck, so you spin up "Claude2" in another terminal from scratch with the fresh approach, maybe referencing some outputs from Claude1 as needed but not everything.

Finally, consider using the *headless mode in CI* for final validation rather than relying on the interactive session memory. For instance, before concluding a phase, run a Claude headless command that reads the key docs and verifies certain constraints (like scanning code for forbidden patterns). Since headless mode doesn't carry your interactive chat history, it's a fresh check. If that headless run finds issues (meaning something that wasn't in memory or was considered fine earlier now flags), you know there's a discrepancy in assumptions that needs addressing.

In summary, treat each major rewrite segment as a new story: plan it with fresh eyes, but armed with the documented knowledge from previous segments. This avoids snowballing errors. The ability to forget is a feature, not a bug – *use it deliberately*.

Deliverable: Claude Memory Template + Governance Rules

Below is a template structure for a Claude memory file (CLAUDE.md) and some governance rules for maintaining it:

<details><summary>CLAUDE.md Template (Project Memory)</summary>

```
# Project Overview
- **System Architecture:** Electron front-end + .NET 6 C# backend.
Communicate via gRPC.
- **Core Principle:** Front-end contains no business logic; all business
rules in backend [71].
- **Data Flow:** UI -> Backend Service -> Database. Use repository pattern in
Service layer.

# Coding Standards
- **Languages/Frameworks:** C# for backend (follow .NET conventions),
TypeScript/React for Electron UI.
- **Style:** CamelCase for local vars, PascalCase for public members (C#). 2-
space indentation JS/TS.
- **Error Handling:** Use `Result<T>` pattern instead of exceptions for
expected errors.
- **Logging:** Use `ILogger` in backend; on UI, log to console for debug
only.

# Testing & Quality
- **Testing:** All new features require unit tests. Prefer integration tests
for service logic.
- **Golden Tests:** Located in `tests/Golden/`. **Must** pass after any
related changes.
- **Linting:** Run `npm run lint` (UI) and `dotnet build` (which includes
analyzers) on backend before commits.

# Repository Etiquette
- **Git Branches:** Use `feature/{JIRA-ticket}` format for new work. Claude
should never commit to `main` directly.
- **Commit Messages:** Follow Conventional Commits (e.g., `fix(auth): ...`)
[31]. Include issue ID.
- **PRs:** Claude can open PRs but a human reviews and merges. Include
summary of changes in PR description.

# Known Legacy Behaviors (for parity)
- Legacy system calculates interest with 4 decimal precision. **Do not alter
precision** in new system without approval.
- Legacy `OrderProcessor` has a quirk: it allows null customer ID for guest
orders. New implementation must handle same.
- Old system sometimes returns error "ERR42" which is actually non-fatal; new
system should mimic this behavior for now.

# Important Invariants
- **User IDs**: Format must remain the same (`USER-XXXX`). Do not change ID
formats.
- **Auth**: Authentication flow must remain backwards-compatible with legacy
tokens.
- **No Regression**: Any behavior in legacy marked as "intentional" must be
preserved or explicitly documented if changed.
```

```
# Tools & Commands
- **Build backend**: `dotnet build`
- **Run all tests**: `dotnet test` for backend, `npm test` for UI. Claude can
run these to verify its changes.
- **Run app**: `npm start` (Electron) and backend runs via `bin/Debug/
Backend.exe`.
- **gh CLI**: Available. Claude can use `gh pr create` to open PRs (ensure
branch is pushed).

# Claude Interaction Preferences
- Be direct and honest; if unsure, ask for clarification rather than assume
[20].
- Explain rationale for significant changes in commit messages or comments.
- Use sub-agents for heavy analysis: e.g., large static analysis = use
analysis agent (to avoid context overflow).
- **STOP conditions:**
  - If about to delete or overwrite large sections of code without
instruction – stop and ask.
  - If encountering an unknown migration scenario – pause for human guidance.
```

</details>

Governance rules for this memory could be: - **Ownership:** The TL (tech lead) owns the content of CLAUDE.md. Any team member (or Claude via suggestion) proposing changes must get TL approval via PR. - **Update Frequency:** CLAUDE.md is updated at the end of each sprint (at minimum) to incorporate new decisions or lessons. - **Accuracy:** Everything in persistent memory must be **true**. If a rule is deprecated or changed, update it immediately in the next commit. - **Brevity:** Keep memory files under, say, 200 lines. If growing beyond that, consider splitting into multiple files (e.g., `/rules/architecture.md`, `/rules/coding-style.md`) using `.claude/rules/*` so Claude loads relevant ones on demand. This prevents context bloat. - **Security:** Never include secrets or credentials. If a rule references a secret (e.g., "uses API key for service X"), phrase it without the actual key. - **Review:** During code reviews, also review any changes to the `.claude/` folder with equal rigor. If a code change seems to contradict an existing CLAUDE.md rule, either the code or the rule is wrong – resolve that discrepancy. - **Backups:** Since `.claude/` guides AI behavior, ensure it's not accidentally deleted or ignored. It should be treated as critical project config. (Consider adding it to protected files list in repository if you have such a mechanism.)

By applying such governance, the persistent memory for Claude becomes a reliable asset rather than a potential source of bugs. It's essentially having a continuously updated spec that your AI agent follows – and just like a spec, it must be managed or it'll fall out of sync with reality.

In conclusion, a sound memory and persistence strategy gives you the *best of both worlds*: Claude remembers the important stuff across sessions, but doesn't carry over the junk. It "forgets" appropriately and "remembers" what you've curated for it. This makes long rewrites feasible without the AI getting lost after day 2. As one Reddit user noted, *"you can use local .md files as a memory system,"* but the straightforward approach alone was lacking until structured techniques (like splitting memory into components) were used [99] – now we have those techniques in this playbook.

D. Guardrails (Hard Constraints, Not Suggestions)

In any AI-assisted development, **guardrails are the non-negotiables** – constraints that the AI must respect at all times. Unlike soft guidance ("prefer this style"), guardrails are hard limits ("never do this"). Claude Code is designed to allow such guardrails through settings, prompts, and permission controls. This section defines Claude-specific guardrails in key categories – scope, behavior, safety, quality, architecture, and explicit stop conditions – and provides deliverables: a one-page spec of enforceable guardrails and an expanded guardrails document template that mature teams use. These guardrails ensure Claude's powerful autonomy doesn't go astray, keeping the AI's contributions safe, correct, and aligned with team policy.

Types of Claude Guardrails

- **Scope Guardrails (Project & Code Scope):** These constrain *where* Claude can make changes or run actions. For example:
- *Directory Whitelist/Blacklist:* E.g., "Claude is allowed to edit files under `/src/new_app/**` but not under `/src/legacy/` unless explicitly told." This prevents accidental modifications to untouched legacy code.
- *File Type Restrictions:* "Do not modify binary files or images." If a change needs to an image (say update an icon), a human will handle it – Claude should not, for instance, try to edit an image's binary content (which it might try if not told – obviously that would be gibberish).
- *Data Scope:* "Don't change database schema without permission." This could be critical in a rewrite: migrating data structures is risky, so you might guardrail that out. Claude should instead produce a migration script and await approval.
- Claude Code's permissions system can enforce many scope guardrails at the tool level. For instance, deny `Write` on certain paths as we saw [100]. Use that to your advantage: set up `.claude/settings.json` with rules like:

```
"permissions": {
  "deny": ["Write(./legacy_app/**)", "Write(./**/*.bin)"]
}
```

so even if a prompt accidentally suggests editing something out-of-bounds, Claude will be blocked and will ask for override (which you presumably won't give).

- **Blast Radius Freeze:** A concept Anthropic suggests: explicitly tell Claude what not to touch in a refactor to limit blast radius [19]. E.g., "In this refactor, do not modify public APIs or change any externally visible behaviors" is a spoken guardrail. Usually, guardrails like that go into the prompt at plan time.

- **Behavioral Guardrails:** These govern *how Claude should behave in conversation and generation*:

- *No Guessing Requirement Details:* Claude should ask for clarification if something is ambiguous. This can be instilled via CLAUDE.md ("if requirements are unclear, ask instead of assuming" [20]) and by habitually phrasing tasks as "if you're unsure, pause and ask me." A skill or rule could also enforce this by scanning Claude's own output; for example, a skill that triggers if Claude produces an output with phrases like "I assume that..." and stops it, reminding it to ask instead.
- *Honesty and Warnings:* If something is outside its capability or risky, Claude should warn or stop. E.g., "Claude, if you are not confident about a destructive action, you must alert the user clearly." This could prevent silent failure modes.

- *No Overselling:* Possibly instruct Claude not to be overly optimistic ("Done, all good!") without evidence. Instead, maybe a rule: "After completing a change, report test results or explicitly state any uncertainties. Don't just say it's done if not sure." This ensures it behaves more cautiously.

- *Ask for Review on Major Changes:* You could set a guardrail that if a diff is huge (say > 500 lines), Claude should automatically suggest splitting into parts or ask for confirmation to proceed. This can be a cultural guardrail (the team expects it) rather than a technical one, but you can encode hints in prompts like "If the diff is very large, verify with me before committing."

- **Safety Guardrails (Security & Dependency):** These ensure Claude doesn't do anything that compromises security or violates policy:

- *No secret exposure:* We already cover not reading sensitive files via permissions. Also instruct it not to print secrets if it happens to see one (e.g., if part of a config file slipped through, it shouldn't regurgitate it in chat).
- *Network Safety:* If Claude has access to network (through an MCP plugin or such), enforce rules like "no external network calls except to allowed domains." Many setups run Claude Code offline or only with vetted APIs (like calling an internal documentation server). If your environment permits some internet access (which is rare in enterprise scenarios), heavily sandbox it. Usually, though, easiest guardrail is: **no internet access** – which is a default unless you explicitly give it.
- *Dependency Management:* If Claude suggests adding a new library dependency, require that it flags this ("Note: adding library X"). A guardrail could be: "Claude must not add any new dependency without user approval." That way it doesn't secretly slip in a random npm package that might be malicious or unapproved. You can catch this in review, but it's nice to instruct from start.

- *No System Damage:* Use sandbox mode for anything risky. If you fear Claude running `rm -rf` or something, keep the permissions tight. The `--dangerously-skip-permissions` exists but with that very name to remind you it's dangerous [7]. Only use it in containers as recommended. In normal operation, keep the default that Claude asks for permission for any unknown command or file write. That prompt to the user acts as a safety check.

- **Quality Guardrails:** These ensure a baseline quality of output:

- *Tests Mandatory:* A rule could be: "If code is written, corresponding tests should be written (or existing tests updated)." Or at least, "Claude must run tests after making changes and confirm they pass." We saw in workflows that instructing it to run tests and iterate is beneficial [48]. Make this a rule rather than optional. Tools: in `.claude/commands/`, you might have a command for running tests, and instruct Claude in CLAUDE.md to always use it after code changes.
- *Linting & Static Analysis:* Guardrail example: "All code must pass linting. Do not introduce new linter errors." If you have a pretty standard linter, Claude often will avoid basic lint issues if it knows (like if `CLAUDE.md` says "use 2 spaces, no unused vars"), but making it explicit helps. Perhaps you even set up a hook: after every file edit, run a formatter or linter via a Claude hook [101] to auto-correct or flag issues (Claude hooks allow running scripts pre/post tool usage, as shown with formatting example [101]).
- *Self-Review:* You might require Claude to do a self-check: e.g., after it provides a solution, ask it "Are there any edge cases not handled? Any potential performance issues?" – This can be formalized: a skill triggers when code is written to have Claude review it if possible (like the multi-Claude verify approach or a skill that encapsulates a checklist).

- *Consistency:* If multiple files are changed, they should be consistent. For instance, if adding a parameter to a function, it should update all calls. This is more of an expectation, but you can

remind Claude as a guardrail: “Don’t make half-changes; propagate changes through all relevant code or inform me if you’re unsure about some usage.”

- **Architectural Guardrails:** These protect the system’s fundamental design and boundaries:

- *Layer Boundaries:* For example: “UI must not call database directly. Backend must not make HTTP calls to third-party except via the designated gateway.” Claude should know these. If it inadvertently suggests violating them, that’s a failure. Put these rules in memory (they rarely change).
- *Tech Stack Decisions:* “We use Tech X for this; do not introduce Tech Y.” E.g., “Must use SQLAlchemy for ORM, do not directly write raw SQL except in migrations.” If Claude tries to do something else, that’s a no-go.
- *Performance Constraints:* If there’s an architectural performance requirement (like “no N+1 database queries; any new code must batch queries or use caching as per our patterns”), state it. Claude can then avoid writing obviously inefficient code or at least call it out.
- *Invariants:* Similar to golden invariants but at arch level: e.g., “All data must pass through validation layer before persistence.” So Claude should not bypass that in new code (like it shouldn’t directly save data without validation).

- *Compatibility:* “Maintain API compatibility with the legacy system unless explicitly changing.” If you’re rewriting a module but it still interacts with others, ensure it doesn’t break those interfaces. This can be tested, but as a guardrail, remind Claude to *not* rename public methods or change message formats.

- **Explicit STOP Conditions:** It’s useful to define certain triggers where Claude should stop what it’s doing and await human input. Possibly:

- *Ambiguous Instruction Detected:* “If you (Claude) are unsure about the requirement or see multiple possible interpretations, stop and ask.”
- *Potential Data Loss:* “If an operation might delete or overwrite significant data, stop.” E.g., migrating a schema might drop a column – Claude should flag that.
- *External Event Needed:* “If completing this task requires approval or information (like a credentials rotation or a big design decision), stop and output a note to that effect.”
- *Scale of Change Exceeded:* e.g., “If more than 10 files need changes for this request, do not proceed further without confirmation.” This prevents an accidental massive refactor from a small prompt.
- *Test Failure:* Possibly encode that if tests remain failing after two attempts, Claude should stop rather than guess further, and hand control back to user. This avoids infinite loops or wasted tokens on flailing.

Claude Code doesn’t have an in-built notion of “stop conditions” beyond needing user permission for certain actions (except if you code that logic in a custom skill or script). But you can simulate it by instructing it clearly and it will usually follow that style. For instance, if you always respond “Stop now” when a guardrail triggers, Claude will learn to anticipate that and might self-regulate by asking proactively.

One-Page Enforceable Guardrails Spec (Deliverable)

This is a concise list of guardrails that could be posted in your repo (maybe as `GUARDRAILS.md` or in CLAUDE.md). It’s phrased in imperative terms as rules for Claude (and the humans guiding it):

<details><summary>Claude Code Guardrails – 1-Page Summary</summary>

Scope Guardrails
- **Allowed Edit Scope:** *Only* modify files within the `new_app/` directory (unless explicitly instructed otherwise). Do **not** alter anything under `legacy_app/` or global config files [19].
- **File Types:** Do not edit binary files, images, or auto-generated files (e.g., `.designer.cs` Windows Forms files).
- **DB Schema:** Do not change database schemas or migrations without explicit approval.

Behavioral Guardrails
- **No Assumptions:** If a requirement or code behavior is unclear, ask for clarification – do not fill in the blanks on your own [20].
- **Permission Checks:** Always seek confirmation for any destructive or irreversible action (e.g., deleting files, force-overwriting code).
- **Rollback Readiness:** For any major change, ensure it could be reverted (e.g., via commit). If uncertain about an approach, pause implementation to confirm direction.

Safety Guardrails
- **Secrets:** Never output or store secret keys, passwords, or sensitive info. Do not read files like `.env` or `credentials.json` [21].
- **External Calls:** No external network calls allowed (no downloading libraries or accessing URLs) unless using approved tools (GitHub API via `gh`, etc.).
- **Dependencies:** Do not introduce new third-party dependencies without approval. Use existing libraries and frameworks as much as possible.

Quality Guardrails
- **Testing:** After making code changes, run the relevant test suites. Ensure all tests pass before considering a task done [5] [48]. New features must include new tests.
- **Linting:** Adhere to coding style guidelines (format, naming, etc.). Fix or report any linter/static analysis warnings in code you produce.
- **Completeness:** Changes should cover all relevant parts of the code (e.g., update all function calls if function signature changes). No half-finished refactors.

Architectural Guardrails
- **Layer Boundaries:** Respect the architecture – e.g., UI code must not directly access the database, backend services must be the only ones containing business logic. No cross-layer shortcuts.
- **API Contracts:** Do not change public API interfaces (function signatures, JSON response formats, etc.) without explicit instruction. Maintain backward compatibility between legacy and new systems.
- **Performance:** Avoid any implementation that would obviously degrade performance (e.g., N+1 DB queries, unbounded loops on large datasets). Flag any potential performance concern.

STOP Conditions (when to Halt and Ask):
- If you are unsure about a requirement or encounter ambiguous instructions – **STOP** and ask for guidance.
- If a proposed change could have wide impact (modify >10 files or a core module) – **PAUSE** and get confirmation before proceeding.
- If test failures persist after 2 attempts to fix – **STOP**; do not guess further without human input.
- If a required action is outside your allowed tools/scope (e.g., need to migrate data or update config) – **STOP** and signal for human handling.

</details>

These guardrails would be visible to any engineer and can be given to Claude as part of the system/project documentation. Many of them can also be enforced through Claude's configuration (permissions) and through careful prompt design.

Expanded Guardrails Document (for Mature Teams)

A mature team might maintain a detailed `AI_GUARDRAILS.md` or similar, which explains each rule, the rationale, and examples. It might be several pages long. For example:

- Under "Scope Guardrails", it could list each directory and what the policy is (editable vs frozen). Perhaps a table of components vs allowed modifications. E.g., "`/common/legacy/` – read-only, do not modify; `/common/utils/` – modifiable but must run full regression tests as these are widely used," etc.
- Under "Behavioral," it might describe scenario-specific guidelines: "During incident response, prefer gathering info over making changes. Always confirm before applying a hotfix in production code," etc.
- For "Safety," it might tie into company security policies (like referencing OWASP top 10 for code contributions, etc., and instructing the AI to avoid patterns that violate those).
- For "Quality," the document might include the standard Definition of Done from the team's process, effectively telling the AI: a task isn't done until code is documented, tests are written, etc.
- It might also include an escalation procedure: "If Claude repeatedly hits a guardrail (e.g., tries to do disallowed action 3 times), the session should be terminated and reviewed by a human."
- Possibly it references compliance requirements: e.g., for regulated industries, "All AI-generated code must be reviewed by at least one senior dev – no exceptions."
- The expanded doc could also instruct how to update the guardrails: i.e., if a rule needs to change, the team must approve it; this avoids someone just telling Claude to ignore a guardrail on a whim.

In practice, you would integrate guardrails in multiple ways: policy docs for the team, CLAUDE.md for the AI, and actual config (permissions, hooks). By having both human-facing and AI-facing guardrail definitions, you ensure alignment – everyone knows what the AI is supposed to not do.

Key point: Guardrails are only effective if consistently enforced. With Claude, that means using the technical features (like always-run tests, permission prompts) in tandem with instructive prompts. The combination of clear instructions ("don't do X") plus mechanized enforcement (Claude literally *cannot* do X because of settings) covers your bases.

The above guardrails, when properly implemented, convert Claude from a sometimes unpredictable agent into a disciplined assistant that operates within clearly marked lanes – essentially *making vibe engineering safe and reliable*.

E. Tool & Skill Exposure (Least Privilege)

Claude Code is powerful partly because it can use tools: your shell commands, code compilers, test runners, linters, custom skills, etc. However, giving an AI free rein with tools can be risky. The principle of **least privilege** applies: give Claude only the tools and access it truly needs to do the job, and nothing more. This section outlines which tools Claude should have access to (and why), the benefits each

provides, their risk profiles, and how to configure them safely. We'll cover safe shell usage, git operations, code search/navigation, build and test automation, linters/static analysis, security scanning, and structured tool APIs (MCP servers). Finally, we present a *Claude Code Tool-Access Matrix* deliverable summarizing recommended privileges and safeguards for each tool.

Exposing Shell Commands (Safely)

Why Shell: The shell lets Claude compile code, run tests, search files, etc., similar to how a developer uses a terminal. It's fundamental for a CLI agent. By default, Claude Code can run shell commands but will ask for permission for anything potentially side-effecting (which is good).

What to Allow: You want Claude to run read-only or safe commands freely, to speed up its autonomy: - File system reads: `ls`, `cat`, `grep`, `find` – these help it navigate and gather context. These have no side effects (reading only) and are low risk, so they can usually be always allowed. - Build commands: compiling code (e.g., `dotnet build`, `npm run build`) – moderate risk (can produce output files but not destructive). Usually safe to allow since at worst it consumes CPU or disk but not harming logic. Benefit: Claude can catch compile errors and type issues by itself. - Test commands: `npm test`, `dotnet test`, etc. Also safe in general (they run code but presumably in a controlled environment). These should definitely be allowed since verifying code is core to quality [5]. - Simple utilities: `echo`, `awk` for formatting, etc. – safe. - Custom project scripts that are read-only or environment setup (like launching a dev server maybe, or running a local docs generator) – as needed.

What to Restrict or Require Confirmation:
- File writes/edits: These are key operations (Claude editing files directly). By default, Claude will prompt "Allow Edit?" each time. That can slow things down if editing many files, so teams often add `Edit` to the allowlist once they trust Claude [102]. But if you do, keep an eye. You can allow edits but maybe not in certain directories (as per scope guardrails above). - Dangerous shell commands: `rm`, `mv`, `chmod`, etc. If not needed, don't allow them at all. If needed (e.g., cleaning build artifacts), restrict specifically (like allow `rm -f ./build/temp/*` but not any rm). - Network commands: `curl`, `wget` – likely deny by default (unless needed for specific tasks like fetching an internal resource). - Installing software: `apt-get`, `pip install` – generally avoid letting AI do environment changes. Pre-install what it needs or manage by you. It could inadvertently install something malicious or break environment. So keep environment static for Claude sessions. - Superuser commands: definitely no `sudo` for the AI.

Configuration: In `.claude/settings.json`, use `allowedTools` and `deniedTools`. For example:

```
"permissions": {
  "allow": ["Edit", "Read", "Grep", "Glob", "Write", "Bash(python *)",
"Bash(npm *)", "Bash(dotnet *)"],
  "deny": ["Bash(rm *)", "Bash(sudo *)", "Bash(curl *)"]
}
```

This says: allow edits, reads, searching, writing new files, running any python/npm/dotnet command (with any args) without asking, but deny any rm, sudo, or curl command execution [102] [21]. The syntax `Bash(git commit:*)` can allow all `git commit` commands, etc. [24]. Fine-tune this list to your comfort.

Least Privilege Thinking: Only add something to allow list when you see Claude legitimately needs it frequently and it's safe. Otherwise, it can always ask, and you approve case-by-case.

Logging & Audit: Keep your terminal scrollback or logs, or use the JSON streaming output to capture all executed commands. That way, if something odd happens, you can trace what Claude ran. Possibly integrate with your CI logging if using headless mode.

Git Access (Read vs Write)

Why Git: Claude using Git can automate many tasks: checking history for context, making commits and branches, merging, etc. It's a huge productivity boost, as many Anthropic engineers found (they let Claude handle the majority of git ops) [60] [59].

Git Read Operations (Safe to Allow):
- `git log`, `git blame`, `git show`, `git diff`: These just read history. Let Claude use these freely. It can answer questions like "who changed this line and why" by itself [103] [104]. - `git status`: safe, to see changes. - `git branch -v` or other queries. All safe.

Git Write Operations (Handle Carefully):
- `git add/commit`: You may allow this, as committing to a local branch is easy to undo (just don't push automatically). Many allow `Bash(git commit:*)` in settings [102] so that Claude can commit changes with messages. Benefit: saves time and documents changes with context [59] [31]. - Risk: poor commit messages or committing incomplete work. But you can review commits afterward. - Mitigation: Possibly have a commit template or require Claude to include certain info. (Some advanced use: commit message skill to format it a certain way.) - `git push`: Potentially risky if pushing to remote without review. You might not allow direct push to main at all (branch protections). Allow push to branches but maybe not by default. Could do manual: You might prefer to manually push after reviewing local commits. - Alternatively, allow `git push` but to a fork or a specific test repo. - `git merge/rebase`: These can alter history or cause conflicts. Probably best not to allow automatically. If needed, have Claude do a dry-run or ask permission. - `gh pr create`: This uses GitHub CLI to open a PR. This is quite useful to allow [34] [62]. Risk is minimal (it just opens PR on your repo). As long as branch was properly pushed, this is fine. The PR is subject to human review and CI anyway. - Other destructive: `git reset --hard`, `git cherry-pick` – avoid letting AI do these unprompted. It might inadvertently drop changes or pick wrong commits. Those should be human decisions (or at least require confirm). - `git revert`: Actually could be allowed, since reverting a commit is usually safe if something went wrong – a nice self-healing step the AI could attempt. But you might keep that manual to avoid confusion.

Configure:

```
"permissions": {
  "allow": ["Bash(git status)", "Bash(git diff *)", "Bash(git log *)",
"Bash(git blame *)",
            "Bash(git commit -m *)", "Bash(git push origin HEAD)"],
  "deny": ["Bash(git push origin main)", "Bash(git merge *)",
"Bash(git reset *)"]
}
```

This example allows reading history and committing, allows pushing current branch, but denies pushing to main (if main is protected, AI couldn't anyway) and denies merges/resets. We allow a generic commit with any message for convenience.

Credential Handling: If you integrate Claude in CI or on a developer machine, ensure it has needed git credentials set up (like a GitHub token if using `gh`). But also ensure those credentials are limited in scope (least privilege for an AI user). Perhaps use a bot account or a personal access token with limited repo access rather than full account.

Audit: Every commit by Claude should be identifiable. Encourage including something like `Co-authored-by: Claude <[email protected]>` in commit trailers or a label on PR like "AI-generated". This provides traceability [32] [33]. Also, commit messages often indicate they were AI-generated by style (but better to tag explicitly). Some orgs require that per policy [32] [105].

Rollback Safety: Typically, keep AI work in its branch. If bad, you just don't merge. Or revert as needed. If Claude could push to a branch, make sure main branch is protected (most teams do that anyway). That way, it can't accidentally cause a direct production deploy or something.

Search/Navigation Tools

Why Tools like Grep/Glob: These allow Claude to find relevant code by itself. For example, if you ask "Where is function X used?", Claude can `grep` the codebase for `X(` and show results. This extends its context beyond what's loaded. It's essential for large code understanding [106] [107].

Allowing Search: `Grep`, `Glob` are read-only and should be fully allowed (they are by default no-permission tools in Claude) [80]. - `grep -R "something" .` – fine, though in huge repos might be slow. But presumably okay. - `find` or `fd` to list files. These just help it figure out project structure (it might do `find . -maxdepth 2` to see top-level dirs). - `ctags` or language server: Currently not integrated, but if you have an MCP for code navigation, you might use that. However, out-of-the-box, grep is the main one.

Recency of Index: Note that grep is live – it reads current files. So if code changes, grep sees it. No stale index problems (unlike some static indexes). - But `grep` might return too many results or miss things if pattern is tricky. It's still effective.

Risks: Minimal – maybe reading large files fully which could fill context if not careful. But that's not a security risk, more a token/cost one. If the repo has confidential text in code comments or something, well if it's in repo presumably allowed. If secrets accidentally in repo, we already told Claude not to open those specific patterns.

No Need to Deny: Except perhaps extremely large binary files could choke grep – but grep wouldn't print binary anyway. If you want, you could exclude `*.min.js` or large data files via permission denies (like in previous examples with `.env.*` etc.) to avoid even reading them.

Encourage Use: Put in CLAUDE.md: "You can use Grep to find code. For example, use `grep -R "term" src/` to locate usages." This reminds it. Actually, Claude knows to do this (since it's how it works often), but it doesn't hurt to mention any project-specific search nuance (like if code is split in unusual places, or if using `git grep` with certain flags would be better).

No specialized config needed beyond ensuring grep and find are in PATH and working. And as said, maybe incorporate search results better. Possibly in the future you might integrate semantic search (embedding-based) as an MCP tool, but that's advanced.

Logging: Grep queries will appear in the command execution log. Keep an eye if Claude does something like `grep -R "password"` – if it's scanning for potential issues, interesting. Usually it greps relevant function names or error codes, which is fine.

Build/Test Tooling

Why Build Tools: Letting Claude compile and run tests is like giving it a superpower to verify its output. It can catch syntax errors, type errors, failing tests and then fix them, which addresses the shallow verification problem. This was highlighted as an Anthropic favorite approach [48].

Allowing Builds: - For compiled languages: `javac`, `dotnet build`, `msbuild`, `gcc`, etc. If your project is in such a language, allow those compilers to run. They generally just create binaries in local directory. Risk is low (assuming no hazardous compiler bug). - If building requires some environment (like specific DB instance up), maybe skip integration tests that require environment if running just build – but that's more test config than tool permission issue. - For interpreted languages: no build, but maybe allow launching a dev server (with caution if it has side effects on environment) for integration tests, etc. - If build scripts generate code or files, consider whether that confuses Claude. Possibly not; it will usually ignore build artifacts.

Allowing Tests: Absolutely allow test running: - `npm test`, `dotnet test`, `pytest`, etc. Running tests is read-only on code (though tests themselves might write to a test DB or files, but hopefully in a controlled way). - If tests require external services (maybe integration tests hitting a staging DB), maybe configure environment to use a local docker or stub services to avoid any external harm. But that's more environment design. At least ensure tests run in a non-prod environment (should anyway). - The value: Claude sees failing tests output and often will parse it to pinpoint what broke and fix it [48].

Risks: - If tests are destructive (clear DB tables, etc.), you might want to ensure test config uses a local SQLite or something, not a real database. That's just good practice in general, not specific to AI usage. - Performance: if test suite is huge, you might not want to run all tests frequently. You could instruct Claude to run a subset relevant to its changes or run faster static analysis first. But mostly, speed is a convenience factor. If it starts running 1000 tests every minor change, that can slow the feedback loop. Possibly direct it: "Prefer running targeted tests unless full suite needed" [108] (Anthropic's CLAUDE.md example suggests that to human devs, which could apply to AI too). - No additional security risk beyond what tests do (unless tests themselves do something wild like drop a schema – which they shouldn't).

Continuous Integration Mode: There is a headless mode for CI (with JSON output) [109] – you could incorporate Claude into CI to, say, automatically fix lint issues or even attempt to fix failing tests in a PR. But that's advanced usage: an MCP or external script calls Claude to do tasks on CI events [110]. If you go that route, definitely heavily sandbox and require approvals for changes.

Logging: Test output might be large. Possibly instruct Claude to summarise or filter it (like only show failing tests). But also, it's fine to see failures – that's how it knows what to fix.

Linters and Static Analysis

Why Linters: They catch style issues, common bugs, etc. If Claude can run them, it can proactively fix code style and some errors, reducing reviewer load.

Examples: `eslint` for JS/TS, `flake8` or `pylint` for Python, `golangci-lint` for Go, SonarQube scanners, etc.

Allowing Linters: - They're read-only in sense they just read code and output warnings. So safe to allow (they may create a report file but that's fine). - If a linter can auto-fix (like `eslint --fix`), that's a write operation. You might or might not allow the AI to run that automatically. Possibly allow it, since auto-fix is deterministic formatting typically. Or have it just run check, then fix via Edit commands so it's visible. - Static analysis tools (like type checkers, or security analyzers) similarly can be run.

Benefits: Claude could incorporate linter output into its own review. E.g., after writing code, it runs `npm run lint` and sees "unused variable" warning, then it knows to remove that. Or run a security scan to ensure no obvious vulnerability introduced [104].

Risks: - If linter is slow or memory heavy, repeated runs could bog machine. Possibly instruct AI not to run a full heavy static analysis on each iteration, maybe only on final or on demand. But if automated in a hook, it will run after each file edit. That might be too slow. So might allow manual run with a command. - Some linters may require config that the AI doesn't automatically know. Ensure `CLAUDE.md` notes how to run them properly (e.g., "use `npm run lint` which runs ESLint with our config"). - Make sure the AI doesn't get confused by linter output if it's lengthy. Usually it can parse it, but you might see it try to fix things that are style preferences. That's fine though.

Allow static type check: e.g., `mypy` for Python, `tsc` for TS. Great to allow, helps catch type mismatches. **Allow static security scans**: e.g., `npm audit` or `bandit` for Python, or custom scripts. Possibly heavier, but could be done as final check.

Audit: You can enforce, via memory or even a skill, that code must pass linter (like a skill that triggers on `Skill: code-review` and it includes "Check for style issues and mention any."). But simpler: integrate in CI as well. So AI tries, but if any slip, CI fails and you then know to instruct fix.

Access: Linters usually just read files, so no special privileges beyond reading. Running them often is just an npm script or pip module.

Security & Dependency Scanning

Why: To catch vulnerabilities or license issues in new code or dependencies. If Claude adds a dependency (with permission), a `npm audit` or `safety` check could catch a known vuln. Or scanning code for common flaws.

Expose Tools: - **Dependency scan:** `npm audit`, `yarn audit` for JS; `pip audit` for Python; `ossindex` or others. They fetch vulnerability DB info, so might require internet (which violates our no internet rule). But many can run offline with cached DB or internal mirror. If not, maybe skip dependency scanning in AI loop; do it in CI. - **Static security linters:** e.g., `semgrep` (which has offline rulesets) for code patterns. Or specific checkers (like SQL injection scanner). - **Secret scan**: ensure Claude didn't accidentally include a secret. A tool like `git secrets` or `truffleHog` could run on diff. But if you prevented reading secrets in first place, this likely not needed.

Risks: These tools sometimes call out. `npm audit` will attempt to fetch advisories. If offline, that fails. Perhaps allow certain network call or have a locally stored advisory DB. Or instruct Claude not to run audit if offline. Up to environment.

Alternatively: If no direct scanning, rely on guardrails and code review for security. But defense-in-depth says if you can incorporate some automated scanning, do it.

Configuration: - Might not explicitly allow these by default (since they might phone home). If you do, ensure it's known. Or only run in CI with internet, not by Claude locally.

Use-case: If the project is sensitive, you might be more conservative and not let AI manage dependencies at all (human does it). But if AI is allowed, then definitely have an automated check after the PR to flag if any dependency is problematic.

Audit: If Claude adds dependency, ensure commit message or PR call it out. Then perhaps have a policy that any new dependency triggers manual security review. That could be a human process or maybe a skill that lists new deps in PR description (Claude can do that, as it sees diff including package.json changes, could mention "Added library X version Y").

Structured Tool Servers (MCP-style)

What are MCP Servers: *Model Context Protocol (MCP)* connectors let Claude interface with external services or APIs in a structured way, beyond shell commands [111]. For example: - A Puppeteer MCP server for controlling a headless browser (for visual tests or web UI manipulation) [53]. - A Jira or issue tracker MCP that lets Claude query ticket info or update tickets. - A database query MCP where Claude can run safe read-only SQL queries to an internal DB for info. - Any other structured environment (some folks integrate Slack or doc search as skills via MCP).

Why Use MCP: They provide capabilities that shell either doesn't have or would be dangerous. E.g., browsing web or screenshots: via MCP you can enforce allowed domains and scrub output.

Least Privilege with MCP: - Only enable the servers that are needed. The `.mcp.json` file lists what Claude can connect to [112]. For each one, limit its scope (most MCP servers have their own permission design too). - E.g., the Puppeteer MCP should maybe be pointed to only your local app's URL, not arbitrary internet. - If you have an internal "Knowledge base" MCP, ensure it only serves non-sensitive data. - Each MCP server runs as separate process usually, with logs. Monitor those logs to see how Claude uses it.

Benefits: These can greatly extend tasks: - Visual diffs with Puppeteer (Claude can take screenshot and compare to an image target) [53]. - Multi-step flows (like controlling a simulator for iOS app testing). - Querying large docs (like AskDev in the cognition blog – though that's more a skill than MCP, but similar principle).

Risks: - Complexity: more moving parts that can break or mis-communicate. Debugging an MCP server issue might be non-trivial. - If an MCP has powerful actions (like one that can modify external state), treat it like granting a specialized permission. E.g., an AWS MCP that can spin up resources would be high risk. - Keep MCP servers as read-only or testing-only if possible. E.g., a Sentry MCP to fetch error logs is read-only and safe [113]. - There is mention in sources of using Sentry or others read-only [113], that is fine.

Tool Access Matrix (Deliverable)

We can summarize recommended access for each category:

Tool / Category	Why Claude Benefits	Risk Profile	Least-Privilege Setup	Logging / Audit
Shell – Read Ops (ls, cat, grep, etc.)	Allows Claude to gather code context, search for usage, read files on demand [106].	Low risk (read-only).	Whitelist common read commands. No confirmation needed. Deny none (read) or very minimal (maybe large binary reads).	Terminal logs show commands; minimal need beyond that.
Shell – Write Ops (file edits, moving files)	Enables Claude to actually apply changes (code edits, file creation).	Moderate – could overwrite or delete content.	Use `Edit` tool with allow if confident [24]; otherwise require confirmation per edit. Deny dangerous patterns (no wildcard deletes). Restrict to project dir (no editing /etc etc.).	Each edit is logged (Claude output diff). Possibly backup files via VCS pre-edit (since using git).
Build Tools (compiler, bundler)	Helps catch compile errors, ensures code builds. Saves human time debugging syntax/type issues.	Low – just generates artifacts; potential heavy resource use.	Allow compilers (e.g., `javac`, `tsc`, `dotnet build`). Ensure they run in project scope (no writing outside). Limit frequency if needed (maybe instruct not to rebuild constantly).	Capture build output. If builds are long, monitor to ensure not stuck. Logs will show any errors found.
Test Runners (unit/ integration tests)	Validates functionality; Claude can iterate until tests pass [48]. Improves code reliability.	Low direct risk – tests might alter test DB or files but in controlled env.	Allow running test commands. Use test config that targets local/dev resources (no production). Possibly sandbox external calls in tests.	Log test results. Consider saving a summary of test outcomes after each run (Claude can do so). Use CI to double-check.

Tool / Category	Why Claude Benefits	Risk Profile	Least-Privilege Setup	Logging / Audit
Git – Read (log, diff, blame)	Provides historical context: why code changed, who owns code, etc. [103]. Useful for troubleshooting and documentation.	Low risk (just reads repo metadata).	Allow all read operations fully. (No need for permission prompts on these.)	Commands will appear in logs. No special audit beyond that.
Git – Write (commit, branch, push)	Automates committing code and opening PRs, maintaining workflow continuity [59] [31].	Moderate – could commit wrong stuff or push to wrong branch. But reversible (commits) and gated (PR review).	Allow local commits on feature branches (possibly auto-allowed commits with message). Restrict push: allow push to current branch, but **not** to protected branches. Require human merge of PRs. Possibly require confirmation for destructive git ops (reset, force push) – or just deny them.	All commits carry Claude's signature or mention. Use branch protection on main. Monitor repo for any direct commits to main by AI (shouldn't happen if guardrails). Review commit diffs in PR as usual.
Search Tools (grep, find)	Allows Claude to find references and files quickly beyond loaded context [107]. Aids understanding large codebases.	Low risk – read-only. Could be heavy on large codebase (perf).	Fully allow. Possibly instruct to limit scope (like grep in `src/` only, exclude huge data dirs).	Not much needed; grep queries show in logs. If performance becomes an issue, consider providing an index or pre-limited search command.
Linters / Formatters (eslint, black, etc.)	Ensures code meets style and basic quality guidelines without manual intervention [42] [43]. Reduces reviewer comments on style.	Low – read-only or trivial file writes for formatting.	Allow running lint checks freely. Optionally allow auto-fix commands (they will edit files; treat like normal file edits). Or have Claude fix issues by reading lint output and applying via Edit (slower but visible).	Lint output logged. Ideally, have CI also run lint to double-confirm nothing slipped.

Tool / Category	Why Claude Benefits	Risk Profile	Least-Privilege Setup	Logging / Audit
Static Analyzers (types, security scans)	Catches deeper issues (type errors, potential bugs or vulns) for Claude to fix proactively [104].	Low for type check (just reading code). Security scans that need internet (moderate – could ping external API).	Allow local type checking (e.g., `mypy`, `tsc`). For security, if offline dataset available, allow the scan. If it requires net, either skip in AI loop or provide sanitized offline advisory DB. Possibly run heavy scans only on demand (or in CI rather than live).	Type checker output logged. Security scan results logged (and likely flagged in PR if CI runs them). Human to inspect any flagged security issues anyway.
Safe DB/ Network Tools (local DB queries, internal API calls via MCP)	Lets Claude fetch info from dev database or internal services to inform decisions (e.g., query how many legacy records exist of a type before deciding migration approach). Could also simulate user actions via API.	Moderate – read-only queries fine, but any write or external API call could have unintended effects. Must be constrained.	If using, set up read-only credentials for DB (maybe a replica or sanitized subset). Expose a custom command like `/query "SELECT ..."` that Claude can use, which internally is restricted. For internal APIs, if allowed, use a sandbox environment (e.g., a staging server, not production). Ideally route through an MCP that restricts what can be done (like only GET requests to certain endpoints).	Log all queries/ calls made (the MCP server should log requests). Review their usage to ensure no sensitive data is fetched or logged in AI conversation if not allowed.

Tool / Category	Why Claude Benefits	Risk Profile	Least-Privilege Setup	Logging / Audit
MCP Servers (special) (Puppeteer, etc.)	Enables complex tasks: e.g., taking screenshots for UI validation [53], interacting with other subsystems programmatically. Extends AI's reach in controlled fashion.	Varies by server: Puppeteer (moderate – could navigate anywhere on web if not restricted; but mainly local usage = safe). Others like file system MCP (similar to direct shell). Each needs individual assessment.	Only enable MCP connectors you need. Configure each: e.g., Puppeteer allowed URLs = your app only. For others, ensure permissions (like an MCP for Jira might only allow read or create comment, not mass close issues). Require explicit enabling via project settings for each (Claude won't use unless listed).	Each MCP typically logs its usage (requests made). Keep those logs (e.g., HTTP calls, browser actions). If something odd appears (AI tried to navigate to external site), adjust restrictions. Possibly add a skill to oversee MCP usage (like to confirm navigation if not whitelisted).

This matrix covers main categories. The key is: *only give Claude what it needs, and tailor each permission.* As a result, Claude becomes a very capable assistant within a sandbox. If it tries something out-of-bounds, the sandbox stops it and you're alerted to approve or deny.

For instance, if Claude attempts `rm -rf /tmp/dir`, your deny list intercepts (or at least asks). Or if it tries to push to `main`, it either can't (due to branch protect) or it triggers a deny. Meanwhile, it can freely do things like run tests and commit to a feature branch, which speeds up development a lot but with minimal risk.

By following this least privilege approach, any damage Claude could do on its own is limited. It operates with roughly the same permissions as a junior developer on a dev machine – and often even less (most junior devs can accidentally do `git push main`; if we prevent Claude from that, it's arguably safer!). Combined with monitoring and human checkpoints, this keeps the engineering process both efficient and under control.

F. Vibe Engineering Workflows (Claude-First)

Now we bring it all together in concrete, end-to-end workflows. These are step-by-step playbooks for common software development tasks, optimized for a Claude Code agent working in tandem with a human. Each workflow covers: preparing an agent brief, having Claude plan, executing the plan (with tools & guardrails), verification steps (tests, etc.), integration with Git/PR, and a human review checklist at the end. We will cover the five mandatory scenarios:

1. **Bugfix in Legacy Code** – how to diagnose and fix a bug in an older system module using Claude.

2. **New Feature (Tests First)** – using Claude to implement a new capability starting from test definitions.
3. **Large Refactor Under Guardrails** – performing a major code refactoring or migration with Claude while ensuring scope control and safety.
4. **Incident/Debug Workflow (Logs & Traces)** – using Claude to assist in investigating and resolving a production incident by analyzing logs/traces.
5. **Legacy → Modern Porting Workflow (Parallel Systems)** – guiding Claude to port functionality from a legacy system to a modern one, maintaining parity and documenting differences.

Each will illustrate Claude-first techniques: how you as the human lead with context and guardrails, and how Claude carries out substantive coding under those guidelines. Let's dive in.

Workflow 1: Bugfix in Legacy Code

Scenario: We have a legacy C# application running alongside a new system. A bug is reported: e.g., "In legacy app, when processing an order with a guest user, it throws a null reference exception." We need to fix this in the legacy code (since legacy is still in use for that part).

Agent Brief (Initial Prompt): Start by giving Claude a clear, structured brief. For example:

```
[User to Claude]
Bug Report: Legacy OrderProcessor throws a NullReferenceException when
processing guest checkout orders (i.e., orders with no logged-in user).

**Details:**
- The error occurs in legacy system only, new system unaffected.
- Stack trace points to `OrderProcessor.ProcessOrder()` at a line involving
`customer.Profile` access.
- Likely cause: `customer` is null for guest orders, not handled properly.

**Task:**
1. Find where in the legacy code this null is not handled.
2. Propose a safe fix to allow guest orders without throwing an exception.
3. Ensure not to break anything for logged-in users or other order paths.
4. Write a regression test in legacy code (if possible) to cover guest order
scenario.
5. Explain any assumptions and ensure legacy behavior parity (guest orders
should perhaps just skip profile-related steps).

Constraints: This is in legacy code (C#). Limit changes to the legacy
`OrderProcessor` or related classes; do not modify global logic. Follow
existing code style. Confirm with tests that fix works.
```

This brief does a few things: - Summarizes the bug and context (so Claude has a target and some clues). - Enumerates steps (1-5) so Claude can follow methodically [1]. - Mentions constraints (don't do broad changes). - Asks for test and explanation, reinforcing good practice (like TDD style) [4].

Planning Phase (Claude): Ideally, ask Claude for a plan: "Before coding, outline your approach." It might respond with: - It will search for `customer` usage in `OrderProcessor`. - Identify likely null

cause. - Suggest adding a null-check or conditional branch. - Mention updating or adding a unit test for the guest scenario. - Possibly mention verifying no other part is affected.

Check plan for sanity: - If it suggests something weird (like “refactor entire order system”), intervene. We want a minimal fix. - If plan looks good, proceed.

Execution Phase:

1. **Code Navigation:** Claude might start with: `grep -R "OrderProcessor" .` or open `OrderProcessor.cs`. It will read it.
2. It finds line `if (customer.Profile.DiscountEligible) ...` which can null-ref if `customer` is null.
3. It might also grep for where `customer` is set, etc. This is fine [106].
4. **Implement Fix:** Claude will propose code changes. E.g., adding:

   ```
   if (customer == null) {
       // handle guest user logic
       Log.Info("Processing guest order (no customer profile)");
       // perhaps skip profile-related steps
   } else {
       // existing logic using customer.Profile
       ...
   }
   ```

   It should output a diff or code block. Use guardrails: make sure it only edits `OrderProcessor` and maybe related minor changes like not touching other modules (the prompt implicitly guided that).
5. If it tries to do more (like modify `Profile` class to allow null), that’s too broad – remind to keep the fix local.
6. **Testing (Claude):** It writes or updates a test:
7. Maybe create a new test `OrderProcessorTests` with a method for guest user scenario. Or if there's an existing test class, add a case.
8. It could simulate a guest user by passing `customer = null` or a dummy order with no customer, and then asserting no exception.
9. Run tests: In CLI, you'd have Claude run `dotnet test LegacyAppTests.dll -Tests OrderProcessorTests.GuestOrder` (if using NUnit/XUnit etc).
10. It sees results. If fails (maybe forgot something), it will adjust code.
11. Possibly the code compiled and test passed. If needed, run full suite to ensure nothing else broke (Claude can do that too).
12. **Verification (Human):** At this point, you (the human) should:
13. Read the diff. Does it make sense? Did it properly handle guest case? E.g., ensure that skipping profile doesn't miss something crucial (maybe guest orders should still have some default profile logic).
14. If anything questionable, ask Claude to explain: “Why is it okay to skip profile entirely? Should we assign a default profile for guests?” etc.
15. Claude may explain that in legacy, likely guest had no profile so skipping is intended.
16. Possibly ask Claude to run the application (if feasible in dev) for a quick integration test, or at least confirm log messages or outputs.

17. **Git Integration:**

18. Have Claude commit the changes: “Commit with message `fix(order): handle guest user in OrderProcessor to avoid null ref (legacy parity)` [31] .”
19. Because we allowed `git commit`, it does so, maybe including the test it added.
20. Maybe instruct: “Open a pull request for this bugfix” – Claude will `gh pr create` with title and description summarizing the fix [103] [104] .
21. The PR description, ideally, should mention the bug, what was changed, and that tests are added.

Human Review Checklist for Bugfix: - Did Claude correctly identify the root cause in code, or just patch symptoms? (Review stack trace vs fix to ensure alignment). - Are all code paths for that function covered? E.g., if there are multiple places `customer` could be null, did it address them? - Is the fix consistent with how similar cases are handled in legacy? (Perhaps there's a pattern for guest users – maybe there’s a `IsGuest` flag it could have used). - Does the added test indeed fail without the fix and pass with the fix? (You can run the test on previous commit to be sure). - Check no side effects: Does skipping profile logic for guests have any negative effect? Maybe cross-verify with new system’s behavior for guests. - Ensure logging or alternative flows are acceptable (Claude added a log; confirm that is formatted and appropriate). - Confirm it didn’t break something else: glance at related tests or any changed behavior (if uncertain, maybe do additional testing). - Verify compliance with guardrails: e.g., function signature unchanged, didn't propagate `null` further where it's not expected, etc. - Are commit message and PR descriptive enough (and possibly tag the issue ID)?

After that, merge PR with confidence (or request Claude/human changes if not satisfied). The bugfix workflow shows Claude acting as a smart debugger and patcher, but the human still validates the logic and ensures it fits the bigger picture.

Workflow 2: New Feature (Tests First)

Scenario: Implement a new feature in the modern system – for example, adding a new API endpoint to the Electron app’s backend to support coupon codes on orders. We will practice tests-first TDD style development with Claude.

Agent Brief:

```
We are adding a new feature: **Apply Coupon Code to Order** in the new
system.

Requirements:
- New API endpoint: POST /api/orders/{orderId}/apply-coupon (with coupon code
in body).
- If coupon code is valid, it should apply discount to the order total.
- If invalid or expired, return an error message, order remains unchanged.
- Coupons come from legacy system, but we have a service to validate them
(`CouponService.Validate(code)` returns discount or null).

Task:
1. Write tests (or specs) for applying coupon:
   - applying a valid coupon reduces order total appropriately.
```

```
   - invalid coupon returns 400 error.
   - cannot apply multiple coupons (if already applied, maybe it replaces?
For now, one at a time).
2. Implement the endpoint and logic to make tests pass.
3. Ensure not to break other order endpoints.
4. Use existing patterns (controller -> service -> domain).
5. Provide any necessary data model updates (maybe add `AppliedCouponCode`
field to Order?).

Let's follow TDD: tests first, then code.
```

Planning (Claude): It may outline: - Identify where to put tests (likely in the backend API tests). - Outline test cases per above. - Plan to add a field to Order for applied discount or coupon code, update calculations. - Plan to implement controller method, calling existing CouponService. - Consider edge cases (coupon already applied? probably replace or disallow). - Good plan will mention updating maybe an `OrderService.ApplyCoupon(orderId, code)` method.

Execution Phase:

1. **Write Tests:**
2. Claude creates or updates a test file, e.g., `OrderControllerTests` or `OrderServiceTests`.
3. It writes tests:
   - `TestApplyCoupon_Valid()`:
   - Setup: create an order with total $100.
   - Use a stub or fake for CouponService to return e.g. 10% off or $10 off.
   - Call the API or service to apply coupon "SAVE10".
   - Assert response is success (HTTP 200).
   - Assert order total now is $90.
   - Assert order has coupon code recorded (maybe).
   - `TestApplyCoupon_Invalid()`:
   - Setup similar, but have CouponService return null.
   - Call apply with "BADCODE".
   - Expect HTTP 400 (or specific error code).
   - Assert order total unchanged.
   - Perhaps check error message content.
   - Potentially `TestApplyCoupon_ReplaceExisting()` if you decide on behavior for multiple calls (maybe last one wins or reject second).
4. The tests likely will be integration tests hitting the controller through an in-memory web server (depending on project structure), or unit tests directly on OrderService with stubbed dependencies. It's up to what context Claude infers. If we want unit, we might have to instruct it specifically to use service level with stubs.
5. Let’s assume integrated approach for simplicity. Claude might need to scaffold a dummy `CouponService` interface or mock – if none existed, it might create one or instruct using a known library for mocking.
6. **Run Tests (expect failure):** In TDD, after writing tests, run them to ensure they fail (and for right reason) [114].
7. Claude runs `dotnet test` or equivalent. The new tests fail (likely NotImplemented exceptions or null references because code isn't there).
8. It confirms they fail as expected (maybe output).

9. It's possible some scaffolding must be done so tests even compile (like adding a stub method to avoid compile error).
10. If needed, Claude might add an empty `ApplyCoupon` method returning not implemented to get past compile to failing test.
11. **Implement Feature Code:**
12. Claude adds necessary pieces:
    - Add field to `Order` model: e.g., `public string? AppliedCouponCode` and `public decimal DiscountAmount` or similar. Or maybe just adjust total in memory.
    - Add method in `OrderService` or similar: `ApplyCoupon(orderId, code)`:
    - fetch order, call CouponService.Validate(code).
    - If result (discount) is not null, apply it (like set order.Total = order.Total - discount, also set coupon code).
    - If null, throw some `InvalidCouponException` or return an error status.
    - Update `OrderController` to have an endpoint mapping to OrderService.ApplyCoupon.
    - Ensure transactions or saving order if needed (depending on system).
    - Possibly update any audit logs or events for coupon applied.
13. Use tools: likely just edits multiple files via `Edit` tool for model, service, controller, plus maybe dependency injection (ensure CouponService is available).
14. It might run `dotnet build` to ensure compile.
15. Then run `dotnet test` again. Ideally, tests now pass [5]:
    - The valid coupon test should now see the updated total, etc.
    - The invalid test should get the expected error (depending on how it's implemented, e.g., maybe returns HTTP 400; ensure that is done correctly via whatever framework).
16. If tests still failing, Claude debugs: maybe miscalculated discount or forgot to handle something. It will refine code.
17. Possibly run entire test suite to ensure no regression on other features (should be none if code is isolated).
18. **Verification (Human):**
19. Review test code: do tests cover likely scenarios? Are they readable and truly testing contract (e.g., if discount logic, maybe test boundary like exactly 100% off code, or how multiple apply calls behave)? We might add a note to test one more thing if we know of any.
20. Review implementation:
    - Check that applying a coupon doesn't break something like tax calculation or finalization flow if done later (we should consider how this fits in order lifecycle).
    - Confirm that coupon validation uses the right service (Claude assumed a `CouponService`, which we said exists, maybe stubbed to always return some fixed discount for known codes).
    - Look at error handling: did it return a proper error object or just a generic? Ensure it aligns with existing API patterns (maybe there's a standardized error response format).
    - Check guardrails: architecture wise, did it follow layering? (Likely yes, if we had a service).
    - Did it avoid changing things like how orders are saved or anything off-scope?
21. If something is off (maybe the code directly updated the database from controller skipping service), instruct changes to align with arch.
22. Possibly test manually if possible (maybe spin up dev server and call endpoint via HTTP).
23. Confirm naming and style: e.g., `ApplyCoupon` vs `ApplyDiscountCode` – ensure consistency with naming conventions (maybe no style issues if it adhered to CLAUDE.md).
24. **Git & PR:**
25. Ask Claude to commit: possibly separate commits for tests and implementation could be nice (TDD flow often sees commit failing tests then commit fix), but since it's one session, maybe just commit all with a combined message.

26. For clarity, might do one commit “test: add failing tests for apply coupon feature” then another “feat: implement apply coupon to orders (passes tests)”. Claude can do sequential commits if you prompt accordingly (committing staged changes).
27. Create PR with summary:
    - Title: `feat(order): support coupon code application`.
    - Body might list scenarios and that tests were added (Claude likely says "Added tests for valid/invalid coupon cases, and implemented OrderService.ApplyCoupon using CouponService. All tests passing.").
28. As human, edit PR description if needed to add context or link to a user story/ticket.

Human Review Checklist for New Feature: - **Requirements Coverage:** Do tests and implementation meet all bullet points of requirement? (Valid code works, invalid yields error, one coupon at a time logic). - **No Regressions:** Did it inadvertently affect existing order logic? (e.g., when calculating total elsewhere, is it aware of coupon discount? If not, maybe okay if coupon just directly adjusts total at apply time). - **Code Quality:** Are names clear (function, variables)? If internal code, less critical for external docs, but still maintainable. - **Architecture & Style:** Did it use correct layers (controller calls service, service uses repository, etc.)? Check that exceptions or errors are handled in typical way (maybe returning a specific error DTO). - **Tests Quality:** Will these tests remain robust? e.g., if coupon logic changes, tests might need update, but they seem straightforward. Are tests self-contained (no external dependency calls unless stubbed)? Possibly see if they properly stubbed CouponService (maybe using a fake or injecting a test double). - **Edge cases:** Think of any not in tests: e.g., what if coupon code is empty string? Or if order already completed (should coupon apply be allowed on completed order)? Possibly out of scope, but mention if something stands out. - **Docs/Comments:** Should any documentation or changelog be updated? Perhaps mention new API in API docs. Might instruct Claude to add a line in `CLAUDE.md` or `README` if relevant ("Added /apply-coupon endpoint – requires passing coupon code, returns updated order"). - **Security:** Could any malicious use of coupon endpoint cause issues? Probably not beyond trying random codes, which is fine and handled. But consider if there's a risk: like infinite use of coupon? (If legacy allows one per order, ensure multiple calls either override or blocked accordingly). - If satisfied, approve and merge PR.

This new feature workflow shows Claude basically doing TDD: writing tests it expects to fail, then writing code to satisfy them [47] [48]. The human ensures those tests align with actual requirements and that code is integrated properly. The final outcome is a feature with tests and presumably minimal bugs.

Workflow 3: Large Refactor Under Guardrails

Scenario: We need to refactor a large piece of the code (legacy or new) without changing external behavior. For example, in the new system, we want to restructure the authentication module – say extract authentication responsibilities out of `UserService` into a new `AuthService` for better separation, without altering functionality.

This is a complex, multi-step refactor where guardrails are essential to avoid breaking things or drifting from intended design.

Agent Brief:

```
Refactoring Goal: Separate authentication logic from `UserService` into a new
`AuthService` module.

**Context:**
```

```
- Currently, `UserService` handles user CRUD and authentication (login/
password verify, token generation).
- We want an `AuthService` solely for auth concerns (login, logout, token
mgmt), and keep `UserService` for profile management.
- The API endpoints /api/login, /api/logout currently call `UserService`.
- We must move that to AuthService *without changing the API outputs or
behavior*.
- Also ensure existing tests for login/logout still pass, and no other part
of system breaks.

**Plan Constraints:**
- Maintain public method signatures for now so that other modules calling
UserService.Auth methods still work (maybe make them call through to
AuthService internally).
- Eventually update controllers to call AuthService directly.
- Do *not* change any business logic (e.g., password hashing stays same,
token TTL stays same).
- Use guardrails: no changes to database schema, no changes to API responses.

**Steps:**
1. Plan the extraction: identify auth-related methods in UserService
(`Login(user,pass)`, `Logout(token)`, etc.).
2. Create AuthService with equivalent methods (initially maybe calling
underlying same code or copy logic).
3. Gradually migrate internals: e.g., make UserService.Login call
AuthService.Login and then return result.
4. Update controllers to use AuthService (once tested).
5. Run all tests (unit, integration) to confirm parity.
6. Clean up: remove auth code from UserService once everything else uses
AuthService.

We should do this in small commits if possible. Let's proceed carefully.
```

This outlines a phased refactor (we might not literally do multiple commits in one go with AI, but we can simulate phases).

Planning (Claude): It will likely break down: - List auth-related functionalities in UserService (maybe `AuthenticateUser`, `GenerateToken`, etc). - Suggest creating new AuthService class, copying those methods there. - Use deprecation or wrapper: e.g., UserService.Authenticate now just calls AuthService.Authenticate. - Then later steps to redirect controllers and remove duplicates. - Emphasize running tests at each step.

Execution Phase:

Phase 1: Create AuthService with duplicate logic 1. Claude creates `AuthService.cs` with methods: `Login(username,pwd)`, `Logout(token)`, etc. It likely copies logic from UserService for verifying password and issuing token. - It must ensure using same dependencies (like the UserRepository for retrieving user, same TokenProvider). - Might also create an interface IAuthService and implement if following patterns. Possibly update DI config to register AuthService. 2. In `UserService`, for each auth function, it either calls AuthService or is prepared to be called by controllers: - Perhaps keep

`UserService.Login` but inside it calls `_authService.Login` and returns that result. - Or depending on design, maybe controller will be changed first. Let's do one step at a time. 3. Ensure compilation: run build & tests. - Possibly tests fail if expecting some internals. But ideally not, since we haven't changed outward behavior; just added new code and forwarded calls, so all should still pass [28]. - If any test was directly calling internal method (like some test calls `UserService.VerifyPassword` which we moved to AuthService), that test needs update to call AuthService or we keep a stub in UserService. - We'll see. Claude addresses test failures by adapting tests or bridging code. 4. *Phase 2: Switch controllers to AuthService directly* - Update `AuthController` or `LoginEndpoint` to use AuthService injection instead of UserService for auth actions. - This should not change API output at all, just from where it gets results. Possibly update DI for controller. - Run tests again (especially integration tests for login/out) – should still pass if AuthService is working. - If any test fails, fix accordingly. 5. *Phase 3: Remove redundant code from UserService* - Now that controllers use AuthService, we can remove the auth methods from UserService (or keep them but mark obsolete if others might still call). - Possibly some old calls in code still call UserService.Auth methods; if so, we should find & replace them to use AuthService. (Use grep to find where `UserService.Login` was used, ensure replaced or forwarded). - Ideally find all references via grep and update to AuthService (except the one inside UserService if still present). - Remove any duplicated logic or fields. Eg, if UserService had a TokenGenerator field and AuthService now has one, ensure not two sources of truth; maybe centralize in AuthService and remove from UserService. - At this stage, we must be careful not to break anything subtle. Good time to run full test suite and maybe do a smoke test of user flows. 6. *Phase 4: Tests & Verification* - Run all tests. If pass, likely refactor succeeded with parity. - Code review (by human, next step) is critical to confirm no drift. - Possibly add new tests if something was untested (though we try not to change behavior, so not needed unless we fear a gap). - Ensure naming etc. (AuthService might have new tests to add similar to what UserService tests were). - Clean up references: e.g., documentation or comments that said "UserService does login" now might say AuthService. 7. *Phase 5: Commit in pieces (optional)* - We might have multiple commits: - commit 1: Add AuthService (duplicate code), forward calls in UserService. (All tests should still pass since essentially nothing externally changed). - commit 2: Update controllers to use AuthService. (Should still pass tests, no behavior change). - commit 3: Remove auth code from UserService and finalize separation. (Pass tests). - Possibly let Claude do one commit per step. Or just one big commit if that's simpler, but best practices prefer incremental. We can do incremental by instructing commit now, then continue.

Using Guardrails: - We explicitly told it “no changes to DB schema, no API changes.” So we check it adhered: - It shouldn’t alter database fields or migration. If it tried to add something to user table or token storage, that’s out of scope – ensure it did not. - API responses remain same (like login still returns token as before, maybe via AuthService but final JSON unchanged). - “Without altering functionality” guardrail ensures it doesn’t do sneaky improvements. Watch if it tries to change password hashing or fix a bug as it sees – we likely want identical behavior. If it does any such, correct it to stick to original behavior (for now). - Scope: limited to user/auth modules. Check it didn’t touch unrelated modules. - Frequent testing acts as a guardrail too (ensuring parity). - If any test fail, that’s an early warning something changed incorrectly [115].

Verification (Human) After Execution: - Review differences: basically ensure that for every place something changed, output remains same. E.g., if previously login error message was "Invalid credentials", check it's still exactly that, not changed by slight code restructure. - Look at commit diffs if done in steps: - First commit: see that it’s largely adding new files and calling them. No logic changes in old methods beyond forwarding. - Second commit: confirm controllers now call new service properly. Check DI config changes (if any). - Third: confirm removal of code in UserService was indeed duplicates and nothing else. - Special attention: Did we inadvertently open any security hole or degrade performance? - E.g., previously maybe an in-memory cache of tokens was in UserService; did we replicate that in AuthService or accidentally drop it? Make sure any such detail carried over. - Or

concurrency aspects: if login had a lock, ensure new service also locks if needed. The AI might not catch subtle concurrency, so human thinking is needed. - Architecture consistency: Did we put AuthService in the right layer? Possibly the architecture expects an interface IAuthService, and usage via that. If Claude just made a concrete class and used directly, consider if that fits your patterns. Maybe fine. - No drift: The risk is minimal since we copied logic, but maybe the AI might optimize something (like “we can simplify this check”); ensure it didn’t. - Tests passing is good assurance. If tests coverage is high around login, we trust the parity. - After thorough review, it looks good.

Git & PR: - If multiple commits, likely one PR containing them. Provide context: - Title: `refactor: extract AuthService from UserService (no behavior change)`. - PR body: "This PR separates auth concerns into AuthService. No user-facing changes. All tests pass. Steps: ... (Claude might list some summary)." - Code review (our current step) essentially done above.

Human Review Checklist for Large Refactor: - **All Tests Pass:** Confirm continuous test green after each phase. (It's implied but double-check). - **Behavior Parity:** For critical flows (login, logout, token refresh maybe), do a quick manual test or at least trust tests. If any missing scenario, maybe test manually or add test. Ensure logs remain same if that matters (maybe not). - **Performance Parity:** Did we introduce extra DB calls or similar? e.g., if UserService login used to do 1 query, AuthService might do 1 – likely same. Just consider if splitting might have doubled something inadvertently (maybe not). - **Clean Code:** Did we remove all redundant code? (No stray duplicate functions left in UserService). - **Backward Compatibility:** Are any other modules or external scripts calling the old methods? If public API, maybe out-of-scope because we kept API same. But if some other code uses `UserService.Login`, check if it's now calling through or needs ref update. - **Follow-up Tasks:** Document if next step is to eventually remove the userService stubs or further improvements. Maybe add a comment "TODO: remove UserService.Login once all call sites updated." Possibly the AI already removed them if none left, which we did. - **Adherence to guardrails:** Confirm no schema changes or external changes. - **Roll-back Plan:** If something does break in production, since no logic changed, low risk. But if needed, can revert PR or temporarily call old flows if still present.

After satisfied, merge PR.

This workflow shows how to leverage Claude for heavy lifting in refactor while the human ensures constraints and subtle details remain correct [116] [28]. The key is splitting into manageable pieces with test verification at each. Claude's planning and methodical execution (with our guidance) can save tons of time on such multi-file operations, especially by updating references and ensuring consistency. Guardrails (test suite, explicit "do not change X") keep it from straying, and human oversight catches anything tests didn't.

Workflow 4: Incident/Debug Workflow (Logs & Traces)

Scenario: A production incident occurred – say the service experienced a spike of errors where orders were not processed due to a null pointer exception on a specific input. We have logs from production and possibly a stack trace. We need to diagnose the root cause and ideally provide a fix or at least a patch, using Claude to speed up analysis of logs and trace.

Agent Brief:

```
We have a production incident:
- Many errors around 2025-12-01 14:30: "NullReferenceException in
OrderPipeline at ProcessPayment()".
```

```
- It appears when a certain payment provider is used (perhaps PayPal).
- We have a log file excerpt (attached) from that time.
- Goal: Identify root cause and suggest a fix or mitigation.

**Data:**
Attached logs from 14:29 to 14:32 (with relevant errors).
Stack trace:
```

NullReferenceException: Object reference not set... at OrderPipeline.ProcessPayment(Order order) in OrderPipeline.cs:line 220 at OrderPipeline.Run(Order order) in OrderPipeline.cs:line 100 ...

```
- Notably, "paymentProvider" was null in logs for those orders.

**Task:**
1. Parse the logs to find patterns (which orders/payment types fail?).
2. Identify code path causing the null (maybe PaymentProvider is not
initialized in certain scenario).
3. Propose a concrete code fix or at least a safe guard (e.g., null-check or
initialization).
4. The fix must maintain behavior for all other cases.

**Context & Guardrails:**
- Legacy code is involved (OrderPipeline is in legacy system).
- We cannot rewrite whole module now, just fix this bug.
- Ensure to update tests or add a test for this scenario after diagnosing.
- This is urgent (incident), so a patch fix is fine as long as it stops error
(we can improve later if needed).
```

We provide logs to Claude. Likely by either copy-pasting relevant parts or using `cat incident.log | claude` to feed (since logs could be large, but let's assume a moderate size).

Using Tools for Logs: - Claude can search logs for "NullReferenceException" or correlating info. - It could parse that at time X, with order ID Y, PaymentProvider is null in context of PayPal maybe. - Possibly we see in logs like `INFO Starting payment via PayPal...` then error.

Execution: 1. **Log Analysis (Claude):** - Might use grep in log: find lines around the error. - Summarize: e.g., "It seems every time PaymentType=PayPal, PaymentProvider remains null and triggers NRE at line 220." or "paymentProvider is not set for PayPal orders, likely a missing assignment when using PayPal." - If logs show differences (maybe for credit card orders, log says "Using Stripe provider" and no error, but PayPal path says "Using PayPal" then error). - Identify code: - Open `OrderPipeline.cs` and look at ProcessPayment method around line 220. (Claude can do that if we give it access). - It finds something like:

```
if(order.PaymentType == PaymentType.CreditCard) {
    provider = new StripeProvider(apiKey);
}
provider.Process(order);  // line 220
```

Possibly missing an `else if` for PayPal, so provider stays null for PayPal and then calling provider.Process leads to NRE. - Confirm by reading code: yes, likely scenario. 2. **Claude's Diagnosis:** "The code does not handle PaymentType.PayPal: `provider` remains null, causing NRE. Root cause: missing branch for PayPal in ProcessPayment." - It might also consider maybe PayPal handled elsewhere or supposed to be configured but isn't. - But logs confirm PayPal used and no provider set. 3. **Proposed Fix:** - Quick patch: add a branch to handle PayPal. Possibly call `provider = new PayPalProvider(...)`. - If no PayPalProvider class exists (maybe not implemented?), could handle differently – maybe treat PayPal orders differently or fail gracefully. - At least, at minimum, add a guard:

```
if(provider == null) {
    Log.Error("No provider for payment type " + order.PaymentType);
    return Result.Failure("Payment type not
supported"); // gracefully handle instead of NRE
}
```

That stops the NRE. - But better to implement PayPal properly if possible. If timeline doesn't allow, maybe just handle gracefully. - Let's assume we have some PayPal integration but maybe forgot to integrate here. Perhaps a `PayPalProvider` class exists but they forgot to plug it in. - Check codebase via search: `grep -R "PayPalProvider" .` – see if one exists. - If yes, likely fix is to instantiate it here. - If not, maybe it's not implemented; then best to catch it and fail gracefully. 4. **Implement Patch (Claude):** According to what's available: - If `PayPalProvider` exists: add

```
else if(order.PaymentType == PaymentType.PayPal) {
    provider = new PayPalProvider(config);
}
```

above the provider.Process call. - If none, then maybe log error and throw a friendly exception. - Add test: either an integration test simulating PayPal PaymentType order and expecting either a handled error or if now implemented, expecting success if possible. - If we implement new provider class last-minute, maybe too big a change. Possibly they had one and just forgot to use it. - For incident resolution, likely just prevent crash and handle it as unsupported type with error. - So code fix with guard as above plus test. - Possibly also add a unit test in `OrderPipelineTests` for PaymentType.PayPal: - Prepare an Order with PaymentType.PayPal. - Call pipeline.Run. - Expect it returns failure (with a message "Payment type not supported" or so) but not exception. - Or if implementing properly, test that it now processes with PayPal provider if configured (but let's assume not enough time, do safe fallback). 5. **Verify:** - Run tests, especially the new one, and full regression. - Should pass (except maybe if some code expected PayPal to throw? Unlikely). - If they had no tests for PayPal flow (which they didn't because it crashed, so likely not tested), now we added one. All others remain unaffected by the small fix if done correctly. 6. **Conclusion (Claude):** Provide summary of fix: - "Added missing PayPal handling in OrderPipeline.ProcessPayment to avoid null provider [19]. Now if PayPal, either instantiate provider or gracefully error instead of null reference. All tests pass, bug should be resolved." 7. **Human Review:** - Confirm that's indeed root cause. (It aligns with log, code). - Is the fix correct and minimal? (Yes, one branch added). - Does it introduce any new risk? - If we chose to error out, now PayPal orders fail gracefully. That might be acceptable short-term (better than crash). - If PayPal is critical, perhaps they'd implement provider properly. But if not ready, at least no crash. - Are we happy with logging and error message? Possibly refine the message to be clear (like "PayPal not supported in this version"). - Because it's production, might also consider if any orders were lost or need reprocessing due to this. That's outside code fix; maybe note down. - Add a comment in code `// TODO: Implement PayPal provider` if needed. - Check guardrails: We didn't break anything

else, one-liner fix in legacy, fine. - Confirm with team if acceptable to not actually process PayPal for now (maybe they didn't realize PayPal was an option with no support). - All tests (particularly integration around Payment pipeline) still pass, and new test is good. 8. **Deployment Consideration:** Because it's incident, they'd want to hotfix. If branch/PR, expedite review/merge and deploy to production.

Git & PR: - commit message like `fix(payment): handle missing PayPal provider to prevent NRE` [19]. - PR description: outlines cause and fix. Possibly mention "temporary fallback, doesn't process PayPal fully but prevents crash" if that's the route.

Human Review Checklist for Incident Fix: - **Root cause verified:** Not just treat symptom. It was indeed cause, we didn't mis-diagnose something else. E.g., ensure that provider truly null because of code path, not that PayPal provider exists but returned null. If the latter, fix might be different. - **Scope of fix:** Minimal risk, localized to one function. Good for a quick patch. - **Future action:** If this was a not-implemented feature, note to properly implement later. If quick patch is acceptable to business short-term (likely yes to stop errors). - **No side-effects:** The fix might cause PayPal orders to not go through (but previously they crashed, so they weren't anyway). Now they fail gracefully, which might be considered improvement. Check with product if this is okay or they'd rather block PayPal option entirely until working. If so, maybe also disable PayPal selection, but that's more UI side. At least no crash. - **Logging/Monitoring:** We added a log for unsupported payment (in code or within the thrown error). Ensure that logs are adequate for monitoring next occurrence or verifying fix in prod. - **Testing in staging**: Possibly simulate a PayPal order in test environment to see new behavior (no crash, proper error). - After verifying all, okay to deploy.

This shows using Claude to parse logs quickly and pinpoint code issues, which it excels at due to pattern matching and context integration [88] [117]. The human adds judgement about whether to implement fully or fail gracefully, etc., and ensures the patch aligns with incident response best practices (fix forward vs hotfix, communication, etc. which are beyond code).

Workflow 5: Legacy → Modern Porting Workflow (Parallel Systems)

Scenario: We have a legacy C# service that calculates something (e.g., tax calculation). In the modern Electron app, initially we still call the legacy for tax calculation. Now we want to port that logic into the new system so both run in parallel and eventually retire legacy. The constraint: legacy and new should produce identical tax results for same inputs. We'll have them in parallel for some time to compare.

This is a complex scenario with multi-step: - Extract domain behavior from legacy. - Implement in new system (maybe in Node or maybe also in C# if new core is C#, but let's assume new core could be TS or same C#, either way). - Document assumptions differences if any. - Validation: we likely create tests or use real data fixtures to ensure parity.

Agent Brief:

```
Port Legacy Tax Calculation to Modern System (Parallel run):

**Legacy**:
- Class `TaxCalculator` in legacy (C#) computes tax on an order given
location, items.
- It supports various rules (different rates per state, exemptions for
certain categories, etc.).
```

```
- New system (Electron app) currently calls legacy via API for tax. We want
to implement a new `TaxCalculator` in the Node backend for independence.

**Requirements:**
- The new implementation must produce the exact same results as legacy for
all existing scenarios (parity).
- We will keep legacy and new running in parallel temporarily to verify
results match on live data.
- No changes to how legacy works (do not modify legacy, just reimplement).
- Document if any intentional differences (e.g., if we find a legacy bug and
choose to fix in new).
- Provide tests using sample orders to confirm parity between old and new
calculation.

**Steps (Plan):**
1. Analyze legacy TaxCalculator logic. Identify all factors (rates, rounding,
etc.).
2. Write unit tests capturing representative cases (perhaps using examples
from legacy docs or create some).
3. Implement new `TaxCalculator` (likely in Node/TypeScript).
4. Run tests to ensure outputs match legacy for all cases (we can call legacy
function from tests too to compare).
5. If any discrepancy, investigate and resolve (or note if it's a legacy bug
that's acceptable to differ, but by default aim for identical).
6. Integrate new TaxCalculator into modern system behind a feature flag or
parallel mode (so both systems can output tax, maybe log differences).
7. Ensure performance is similar or acceptable (tax calc is not heavy, so
fine).

**Guardrails:**
- Do not change any business rule without explicit decision.
- Use the same data inputs/outputs structure (so that external behavior is
unchanged).
- Place new code in appropriate module (likely a new service or util in Node
backend).
- Make sure to handle all special cases legacy handles (e.g., maybe if item
is gift card, tax is 0, etc).
```

Plan (Claude): It may create: - List of things to check in legacy code (like rates table, which states considered, how rounding done). - Suggest copying any static data (tax rates per state if hardcoded or config). - Outline making tests: could call legacy code via some harness (maybe if we can compile legacy code into test context, or easier: take known outputs from legacy and hardcode expected values). - Possibly suggest to run a few orders through legacy to generate expected results for tests. If legacy code accessible to tests, do that. If not, use predetermined examples.

Execution:

1. **Reverse Engineer Legacy Logic:**
2. Claude opens `TaxCalculator.cs` (with permission we gave).
3. Reads through, might find logic:

```
decimal rate = stateRates[order.ShippingState];
if(order.IsExempt) return 0;
decimal tax = rate * order.SubTotal;
if(order.HasLuxuryItem) tax += luxurySurtax(order);
return RoundToCents(tax);
```

4. There's probably a bunch of conditions: categories exemptions, caps, etc.
5. It notes them down.
6. If the logic is complex, progressive disclosure: maybe break it by reading related functions (like `luxurySurtax` function).
7. Summaries: "Tax = base rate * subtotal, unless exempt categories or certain item triggers extra 5% luxury tax on portion above $500, etc..."
8. **Create Test Cases:**
9. For parity, one approach: incorporate legacy function in the test, if environment allows. Possibly, if the new system is Node, we can't easily call C# code. But we can gather expected values from legacy by either:
    - Running a small C# program for known inputs (maybe not easily done within Node tests).
    - Or manually, or use documentation. Possibly there's known examples in docs.
10. If none, we might rely on some sample orders from production or create hypothetical ones:
    - Example 1: Basic order $100 in CA (rate 7.25%) no special items -> expected tax $7.25.
    - Example 2: Order $600 including a luxury item in NY (e.g., base rate 4%, luxury above $500 taxed an extra 2%?), etc, expected calculation.
11. Might be easier: run legacy logic within a unit test if we wrote one in the legacy codebase and got outputs.
12. But since parallel, let's assume we glean enough to craft expected outcomes.
13. Write tests in new system testing new TaxCalculator outputs against hardcoded expected from old logic.
14. For more assurance, possibly ask user to supply a couple of real anonymized examples (but let's proceed with hypothetical known).
15. **Implement New TaxCalculator (Node):**
16. Write TypeScript class `TaxCalculator` replicating logic:
    - Possibly store a state->rate map (maybe get from config or code).
    - Implement exemption rule (if order flag Exempt or category "food" -> 0 tax).
    - Implement luxury rule (if any item category = luxury and price> threshold, additional tax).
    - Use similar rounding strategy (legacy RoundToCents likely round half up).
17. Pay attention to any subtlety (like decimal vs double differences; in Node, use either Number or a decimal library for currency accuracy).
18. Aim to mimic exactly (could even incorporate exact rate values and thresholds).
19. **Run Tests:**
20. See if all test pass. If not, see differences:
    - If differences, determine why. Possibly a rounding difference (maybe Node rounding default differs slightly).
    - Adjust accordingly (like ensure using same rounding method).
    - Or maybe misinterpreted a rule. If unclear, possibly run more logging or ask for clarity in doc or code comments.
21. Eventually align results.
22. **Parallel integration:**
23. Maybe not fully done by AI, but we want to integrate new in a way to verify parallel:
    - Possibly in new codebase, when calculating tax, call both old via API and new, and log if discrepancy. This could be a step to run in test mode or internal only.

    - But a simpler approach: initially, not turning off legacy. Perhaps behind a flag `UseNewTaxCalc` that defaults false, and we will test internally with it on.
24. Claude can add code to do both:
    - e.g., in OrderProcessing pipeline (new system),

```
if(config.useNewTax) {
   let newTax = TaxCalculator.calc(order);
   // optionally compare with order.tax from legacy (if still
getting it)
   console.log(`Legacy tax: ${order.legacyTax}, New tax: ${newTax}
`);
   order.tax = newTax; // if we decide to override
}
```

    - Or more systematically, depend on strategy (Team might not want both concurrently, but often they do hidden).
25. Up to us, but instruct maybe to implement but not fully activate by default. Possibly out-of-scope of code— maybe they'd manually compare offline. We'll skip heavy integration details.
26. **Documentation:**
27. Ensure to comment or note any decision.
28. E.g., if found a minor bug in legacy (say legacy didn't exempt a category it should, and we decide to keep that bug for parity or fix now?), mention it.
29. Let's assume no intentional change, so no difference doc needed besides an internal note "logic is identical to legacy including known quirks".
30. **Verify comprehensively:**
31. Might consider building a quick harness to test multiple random orders with both calc and see difference (if we could call legacy easily). Possibly too advanced for automated test.
32. But manual: maybe test a typical scenario with known expected outcomes (some tax table references).
33. If truly worried, one might run new calc on a dump of recent orders and compare with tax stored (that would be robust, but outside AI coding, more data processing).
34. At least, trust the tests we've done and careful review.

Human Review: - Compare code side-by-side with legacy if possible to ensure all branches accounted for: - Check all if/else in legacy present in new. - Check initializations (like if legacy had default rate for unknown state, new does too). - Rounding differences (floating vs decimal). - Did we inadvertently change something? Possibly in Node the representation might cause micro differences (like .NET decimal vs JS Number might have slight fraction difference on some calculations). - If so, consider using a library or higher precision. If no evidence of issue, maybe fine given typical currency amounts. - Are the tests sufficiently covering corner cases? Possibly add more, e.g., an exempt scenario test, a luxury scenario test if not already, a zero amount test, etc. - Performance: likely fine (simple math). - Integration path: decide with team how to test new vs old in production, but that's beyond code. Maybe mention in PR description suggestion to run dual for a period. - Mark legacy code region or make note to remove reliance after thorough testing of new (maybe in a future story).

Git & PR: - Possibly multiple commits: 1. "feat: implement new TaxCalculator with parity tests" (this adds new code and tests). 2. "chore: add flag for new tax calc (not enabled by default)" (if such integration done). 3. Could also include removing legacy call, but that likely not now, maybe later after confidence. - PR title: `feat: Port tax calculation from legacy (parity achieved)` - PR body: explain done to reach parity, mention tests verifying identical output on X sample scenarios, no customer-

facing change yet (flag off or just internal). - Maybe attach a small comparison of outputs in description if available (like "For 10 tested orders, results match to cent").

Human Review Checklist for Porting: - **Completeness:** All features of legacy implemented? (Check code and any documentation for legacy). - **Parity Evidence:** Are we confident in parity? If possible, maybe run a quick check on 1-2 real orders using both systems if accessible. Could run legacy function in immediate mode using known example, compare new. If possible, do that. - **No inadvertent improvement/regression:** If we discovered a legacy bug, did we intentionally copy it or fix it? Team might want to preserve behavior first then fix separately. If we fixed it, call it out as intentional divergence with justification. - **Testing thoroughness:** Possibly bring a legacy dev or domain expert to review the new implementation to see if it captures everything (like some obscure tax rule they know). - **Switch plan:** Confirm strategy to flip over and monitoring. Possibly not in PR but as deployment plan. - **Performance**: If large data like tax tables used, ensure not doing something inefficient (like computing full rate table each call - hopefully not). If stateRates map is static, fine.

After all good, likely merge, and plan to gradually route traffic to new calc with monitoring.

This porting workflow highlights structured approach: extract domain logic, meticulously test for parity, incorporate domain knowledge, and use the combination of Claude's brute-force copying ability and human domain oversight to ensure nothing is missed. The code is new, but it's essentially reimplementing old, so creativity is in tests and making sure it's identical. And we used the vibe engineering principle of **explicit testing and parallel run to guard against drift**.

G. Claude-Assisted Git & PR Strategies

When an AI like Claude is contributing code, it changes some dynamics of version control and code review. This section focuses on **Git and PR discipline** tailored for AI-generated changes. We discuss branch strategies, commit sizing and messaging, PR composition and templates, how to review AI diffs, labeling AI contributions, and strategies like frequent rebasing or reverts to manage AI-driven branches. The deliverable is a *Claude-first Git & PR playbook*, which teams can adopt to integrate AI contributions smoothly into their development workflow.

Branch Strategy for AI Work

Use feature branches for Claude's work just like a human's work. It's wise to isolate AI changes in a branch (or multiple smaller branches) rather than committing directly to main. For example: - If you're working on a bugfix with Claude, create `fix/bug-1234-claude` branch. - For a new feature: `feature/coupon-code-ai`. - Having "ai" or the task in the name is optional, but might hint that it's largely AI-generated code inside (for awareness).

Why branches? It allows normal PR review process, CI checks, etc., before merging to main. It also prevents the AI from messing up main if something goes wrong. And if an AI branch goes off track, it's easy to discard or redo without affecting main or other branches.

Keep AI branches short-lived. Aim to complete the task and merge relatively quickly (same day or a few days), to avoid drift from main. AI context is freshest when branch is up-to-date with main: - If branch lingers, main might move and merge conflicts accumulate. Also AI memory might have stale assumptions. - If branch gets outdated, consider rebasing main into it before continuing AI work (to feed AI the updated context). Claude can handle rebases/conflict resolution [103] [104], but you might supervise that.

Parallel AI branches: You can have multiple AI sessions on different branches (like Anthropic example of multi-worktrees) [118] [119]. But coordinate so they don't conflict in same area. If they do, merge one before the other or prepare for conflict resolution.

Use descriptive branch names. Nothing new, but helps track: e.g., `ai/` prefix or suffix could be used if you want to mark AI heavy branch. This is up to team preference. Some might do `experiment/` if trying an AI-led refactor that might not merge if it fails.

Commit Granularity and Conventions

Small, focused commits. Encourage Claude (and yourself when guiding it) to break changes into logical commits. For instance, in the refactor workflow we had it do multiple commits for each stage. This makes review easier: reviewers can see a clean commit implementing X. - While Claude could dump everything in one commit, it's better to say "commit after adding tests" then "commit after implementing code" [45]. - If using CLI, you can stage certain files and instruct commit message accordingly.

Commit Messages from Claude: Claude is quite capable of drafting commit messages that summarize changes [31]. Often it will include context from diff: - E.g., "fix(order): handle null PaymentProvider for PayPal – Added branch to instantiate PayPal provider or error out if none, preventing NRE [19]." - These can be *really good* because Claude remembers the reasoning, which sometimes devs might not articulate. - However, verify accuracy. If commit message says "Added tests for X" but actually tests commit was separate, adjust. It's only as accurate as its understanding.

Adopt a convention for AI commits: Some teams add a trailer or tag. For example: - Append `[AI]` in commit subject or body. - Or use Co-authored-by line: `Co-authored-by: Claude Code <claude@anthropic.com>` [120]. - This explicitly marks AI involvement which can be useful for audit or later analysis (e.g., if a bug is found, knowing AI wrote that code might hint at a certain pattern). - As Addy Osmani noted, labeling AI contributions improves accountability and review focus [32] [121]. - This should not replace the human as author (the human who ran Claude could still be primary author, with AI as co-author). - It's an emerging practice so decide at team level.

No commit directly to main: Even after PR approval, use normal merge or squash. If squashing, preserve commit message clarity.

Commit message guidelines: - If following Conventional Commits (feat, fix, etc.), ensure Claude's messages follow that structure [31]. It often does if it sees examples in history. If not, you can instruct: "Write commit message in Conventional format: `<type>(<scope>): <summary>`". - It's a good idea that commit messages by AI include the "why" if not obvious, since code is not discussed in stand-ups. Claude can even include context like "This resolves bug 123 by adding null checks" which is valuable in log.

PR Size and Content

Limit PR size: AI can churn out a lot quickly; don't let it create a PR with thousands of line changes unless absolutely necessary. As a guideline: - If PR > ~500 lines, ensure it's well-justified and reviewers have time. If not, consider splitting it. - For example, a big refactor might be split into multiple sequential PRs (like create new module, then next PR switch usage, etc.). This parallels how you'd manually do phased refactor.

One logical change per PR: - Keep the PR focused: don't bundle an AI bugfix with an unrelated AI formatting cleanup, etc. It's easy for AI to "fix" multiple things it notices, but restrain it to the request at hand. - Resist temptation to say "Also, Claude, fix all lint in the file while you're there." That can inflate PR diff and confuse the narrative of the change. Do separate commit/PR for mass lint fixes if needed.

PR Templates optimized for AI: - Use PR template to force certain info that AI should fill: - *Summary of Change:* Claude can generate a succinct summary. - *Testing:* If it wrote tests or ran tests, it can note "All existing tests pass; new tests added for scenarios X and Y [48]." - *Impacted areas:* It might list modules or endpoints changed. - *AI involvement:* maybe a checkbox "This PR includes AI-generated code – [X] Yes". - *Review Notes:* We could have a section "Areas of uncertainty or assumptions:" where if Claude had to assume something, it lists it. We can prompt it to fill such. - The template ensures consistency and thoroughness. The human author (with AI's help) should fill it out.

Example PR Description (for the coupon feature):

```
**Summary:**
Implements coupon code application feature. Adds new API endpoint and
underlying logic to apply a valid coupon to an order, including updating
order total and handling invalid codes.

**Changes:**
- New `ApplyCoupon` method in `OrderService` and `AuthController`.
- Added `CouponService` integration to validate coupon codes.
- Unit tests added for applying valid vs invalid coupon.

**Verification:**
- All unit and integration tests pass (including 4 new tests covering this
feature).
- Manual test: applied a sample coupon "SAVE10" to an order in dev, order
total reduced by 10% as expected.

**Notes:**
No changes to existing order flows except when coupon endpoint is used. If an
invalid code is provided, returns HTTP 400 with error message.

Co-authored-by: Claude (AI)
```

Claude can draft much of this and then the human tweaks factual bits.

Reviewing AI-Generated Code (Human Diff Review)

Be extra vigilant: As a reviewer, approach AI code like code from a junior developer who might have misunderstood parts. Common patterns to watch: - Over-generalization or under-generalization (did it handle all edge cases?). - Silent assumptions (maybe it assumed a variable is always non-null because it didn't see null usage). - Inconsistent style or patterns (if AI didn't fully align with project idioms). - Security pitfalls (AI might not realize a certain input needs sanitization or that an operation is sensitive). - Performance issues (maybe fine 90% of time, but check if any unusual algorithm choices). - Dead code or over-engineering (AI sometimes adds an unused method or parameter anticipating something).

Use tools: Run the tests yourself; use linters or static analyzers on the diff to catch obvious no-nos. If your CI shows any new warnings for that PR, double-check them - AI might introduce subtle warnings.

Focus on intent compliance: Ensure the code does what the task intended and nothing more. AI may add minor features not asked for (like logging or additional fields) that might not be desired. E.g., if it spontaneously logs sensitive info, mark that in review.

Check complexity: Sometimes AI solutions can be more complex than needed. If you spot simpler, suggest refactor. It's like you'd review a human's code – propose a cleaner approach if viable. But weigh if it's necessary now or not (maybe accept as is if correct and not too bad, to avoid re-looping AI for minor aesthetics, unless codebase quality would suffer).

Diff Size: If PR is large, consider reviewing commit by commit if AI did logical commits. That can clarify the progression. Also rely on the PR description (which AI wrote) to guide you.

Encourage team knowledge sharing: If reviewer sees something odd, ask for clarification (via comments). The "author" (human driving AI) should answer, possibly using AI to clarify if needed. This ensures not merging code no one understands. It's fine if code is complex as long as it's correct and documented.

Labeling and Tracking AI Code: - If your org or repo tracks AI contributions, ensure proper tagging (like label the PR "ai-generated" or mention in commit trailer). This is more for later analytics or compliance (some companies might require noting AI usage). - This also cues reviewers to maybe spend a bit more attention, as suggested by industry voices [32] [121].

Reviewer Mindset: Approach it collaboratively: the AI is like a pair programmer that did a chunk; ask questions, and possibly you can even feed that back to Claude in review stage. For example, if reviewer wonders "what if coupon is applied twice?" and it's not handled, you could go back to Claude to update code for that scenario. Then update PR. - This iterative refine in review is actually smoother than with humans sometimes, as the AI can produce the patch quickly.

No rubber-stamp: Avoid assumption "AI wrote it so it must be right" or conversely "AI wrote it, it's suspect." Do a normal thorough review. Over time, you'll gauge how reliable the output is and where it tends to slip up.

Labeling and Tracking AI Contributions

Commit and PR Labels as mentioned: - Could have a commit footer like `AI-Authored: true` or just rely on Co-authored-by. - PR label "AI" or "gpt" or such to filter later. Some regulated projects might require an annotation in code comments that "This function created by AI on 2025-12-01", but that might clutter code. Better keep metadata in commit messages or PR notes.

Repository Tracking: - Possibly maintain a document or system listing AI involvement. This might help in post-mortems ("this bug was in code originally generated by AI on version X"). It's arguable if that matters, but some orgs want the provenance for legal or quality. - Tools might emerge to automatically detect AI code, but until then, manual labeling helps.

Developer Attitude and Transparency: - Encourage a culture where using AI is not hidden. Developers shouldn't try to pass off AI code as solely their own if they're not fully confident in it. Tagging helps remove stigma and increases vigilance. - As Addy Osmani suggests, psychological safety for devs to

disclose AI use is crucial [122] [120]. So incorporate that: "This PR uses AI assistance" is a neutral statement, not a negative.

Rollback and Revert Strategies

If an AI-generated change goes wrong (post-merge bug or performance issue): - **Revert quickly** if needed. It's often safe to revert since presumably it was somewhat isolated. Then re-assess with either a new AI attempt or manual fix. - Because of commit labeling, it's easy to find which commit was AI. But treat it same as any commit: if broken, revert. (Don't hesitate thinking "but maybe it's doing something smart" – if it's breaking things, trust your pipeline). - Maintain normal version control discipline: small commits are easier to revert individually if something in a big PR was bad. If everything got squashed, you revert whole PR or cherry-pick out problematic part. - With AI, sometimes the failure might be subtle or discovered later. Use git blame (with Co-authored info, you might see "Claude" in blame if co-author line isn't accounted by blame tools – so maybe not, better to rely on commit message or code comments). - If an AI commit is reverted, maybe re-run a new session with the knowledge of what went wrong to generate a better solution.

Preventing Branch Drift: - As said, merge AI branches promptly. If an AI branch is expected to be stale (maybe waiting for some reason), at least periodically rebase it on main and run tests again, possibly re-engage Claude to fix merge conflicts or update context if main introduced changes in same area. - Alternatively, break work into smaller PRs so branch doesn't diverge much. - For long-running AI tasks (like huge refactor that could take days to complete with verifying steps), consider splitting into sub-tasks or multi-phase PRs to integrate partial progress gradually rather than one long branch.

The Playbook Deliverable:

Claude-First Git & PR Playbook:

- *Branch Naming & Scope:* Create a new branch for each Claude-assisted task (e.g., `feature/<name>-ai`). Limit branch scope to a specific feature or fix. Avoid letting AI work directly on main or dev branches.
- *Regular Rebase:* If the AI branch exists for more than a day, rebase it onto the latest main to incorporate new changes and reduce merge conflicts. Re-run tests after rebasing. Use Claude to help resolve any conflicts safely, but review conflict resolutions manually.
- *Small Commits:* Instruct Claude to make logical, incremental commits during development. Each commit should ideally compile and pass tests. This yields a clear history and easier troubleshooting. (e.g., separate commits for "Add failing tests", "Implement function", "Remove old code").
- *Commit Messages:* Ensure commit messages are descriptive [31]. Use conventional format if adopted (e.g., `fix(auth): ...`). Include context or reasoning especially if code is non-trivial ("Null-check X to prevent Y"). Include `Co-authored-by: Claude` in commits that are heavily AI-generated to maintain provenance [120].
- *PR Preparation:* Open a Pull Request as usual. Use a PR template that prompts for a summary, testing evidence, and any AI-related notes. Have Claude draft the PR description summarizing what it did [123], then the human verifies and edits if needed (ensuring accuracy of descriptions and no oversights).
- *PR Labeling:* Apply a label (e.g., "AI-generated") to the PR for transparency. In the PR description, consider a brief note like "This change was implemented with assistance from Claude Code." to inform reviewers (which can prompt them to review carefully) [32].

- *Review Process:* Treat AI-written code with the same scrutiny as human-written. Reviewers should focus on correctness, adherence to requirements, and maintainability. Use the PR description and commit messages (written by Claude) to understand the rationale. If unclear, ask questions in comments – the human author can get clarification from Claude if needed.
- *Testing & CI:* Ensure CI runs all tests and linters on the AI branch. Do not merge until CI is green. If AI introduced new warnings or minor style issues, have Claude fix them in a new commit before merging (or fix manually).
- *Merge Strategy:* Prefer merging via PR (squash or rebase as per team norm). Squash merge will condense AI's multiple commits – if doing so, make sure the final commit message captures key details from all commits (the PR description can help form this).
- *Post-Merge Monitoring:* After merge, watch the next build or release for any anomalies potentially from the AI changes. Be ready to revert quickly if a problem emerges. Because changes are in isolated commits/PRs, reverts should be straightforward.
- *Revert if Needed:* If an AI-generated change causes issues, do not hesitate to revert the commit or rollback the deploy. Investigate with the help of Claude (feed it the situation) and create a follow-up PR with a fix. Document the incident (perhaps noting the AI contribution; this helps refine future AI usage).
- *Knowledge Sharing:* Include AI-authored code in code review discussions or post-mortems as learning opportunities. Ensure multiple team members familiarize themselves with the AI-generated sections (no "black boxes" owned only by AI). Over time, this builds trust and understanding of AI contributions among the team.

By following this playbook, teams can integrate AI contributions systematically: branches isolate risk, commit and PR hygiene facilitate oversight, and labeling ensures accountability and continuous improvement in how we use AI in development.

H. Claude Code Configuration & Project Structure

Claude Code introduces new configuration files and project structures (via the `.claude/` directory) that differ from traditional development setups. Understanding these is crucial for effectively using Claude in a team setting. This section provides a deep dive into configuring Claude Code for a project, including best practices and potential pitfalls for each component:

- `CLAUDE.md` (global and directory-specific)
- `.claude/settings.json` (project-wide config)
- `.claude/settings.local.json` (personal overrides)
- `.claude/commands/` (custom slash commands)
- `.claude/skills/` (reusable agent skills)
- `.claude/agents/` (custom sub-agents)
- `.claude/rules/` (modular instruction sets)
- The relationship between these and older approaches (like one monolithic CLAUDE.md).

We'll explain the purpose of each, how to design them effectively, and note common failure modes if misused. This will culminate in an example `.claude/` layout for a large repo.

`CLAUDE.md` (Root and Subdirectories)

Purpose: `CLAUDE.md` files serve as persistent context and instructions that Claude automatically loads for a given directory context [12] [13]. Think of them as living documentation/guidelines for the AI.

- The root `CLAUDE.md` is loaded in every session started at the repo root (most sessions). Use it for project-wide knowledge: build commands, coding style, key architectural guidelines, definitions of terms, etc [95].
- Subdirectory `CLAUDE.md` files are loaded when Claude is working in that subdir or on files within it [124] [38]. Use them for folder-specific info. For example, in `services/Payments/CLAUDE.md`, detail how payments are structured, relevant domain knowledge (like "Payment statuses: Pending, Settled, etc."), and any rules unique to that module.
- Claude will load parent directory CLAUDE.md as well if you run it deeper in a tree. Also child CLAUDE.md on demand when needed [13] [38]. So structure matters: keep each scoped to what's relevant at that level.

Best Practices: - **Keep them concise and curated:** A CLAUDE.md is part of the prompt, so large ones can consume context. Include only what's truly helpful often. Aim for maybe <100 lines in root if possible. Use bullet points and short sections (Claude reads it often; readability matters). - **Human-readable:** They should be understandable to human devs too (since it's shared in git). It doubles as documentation for team members [125]. - **Update regularly:** Treat like code: when architecture changes or new conventions come, update CLAUDE.md so Claude is aware next time. E.g., if a new naming convention is decided, add to style guide section. - **Examples:** Provide small examples in CLAUDE.md (like a snippet of code illustrating a style) [126]. Claude learns patterns well from examples. - **Emphasize important points:** If certain rules are critical (like "IMPORTANT: All DB queries must go through XYZ layer"), put that in CLAUDE.md maybe with emphasis. Anthropic notes adding "IMPORTANT" or "YOU MUST" can improve adherence [127]. - **Directory scope usage:** Use subdirectory CLAUDE.md to avoid repeating info relevant only to that module. E.g., in `mobile/CLAUDE.md`, mention mobile platform constraints not relevant to backend.

Failure Modes: - *Bloat:* Dumping huge design docs or spec dumps can dilute focus. If CLAUDE.md is too extensive, Claude might ignore parts or run out of context. Use `.claude/rules/` (discussed below) to break into modular pieces if needed. - *Outdated info:* If not maintained, Claude might follow outdated guidance, causing misalignment with current code. E.g., if it says "we use framework X" but team switched to Y. - *Over-specific or biasing examples:* If an example in CLAUDE.md is too specific, Claude might pattern-match too closely. E.g., an example code might cause it to always use the same variable names in outputs. Provide representative but generic examples. - *Conflicts:* If different CLAUDE.md files have conflicting instructions (maybe root says one thing, subdir says another), it could confuse Claude. Make sure instructions are consistent or context-appropriate. If conflict, presumably the more local one (subdir) should override for that scope – so design accordingly.

In summary, `CLAUDE.md` is the quick reference and rulebook for Claude – keep it handy, correct, and to the point.

`.claude/settings.json` (Project-wide Configuration)

Purpose: This file defines how Claude Code behaves for your project at a configuration level (tools allowed, environment variables, hooks, etc.), and is shared with the team via git [128]. It's essentially the "AI agent configuration" for the repo.

Key Configurable Areas: - **Permissions (Allow/Deny Tools):** We discussed in section E how to set allowed and denied tools (e.g., always allow Edit, always deny dangerous commands) [102] [100] . Put those here for team-wide policy. For instance,

```
"permissions": {
  "deny": [
     "Read(./secrets/**)", "Write(./secrets/**)",
     "Bash(rm *)"
   ],
  "allow": ["Edit", "Write", "Bash(git commit:*)"]
}
```

This ensures every engineer's Claude follows these rules without each individually setting it. - **Environment Variables**: If the AI needs certain env vars (like API keys for tools or config paths), you can define them in settings.json so Claude has them each session [129] [130] . But be cautious: do not put actual secrets here in git. Instead, possibly reference placeholders and have actual values in `settings.local.json` . - Example: set `"env": { "API_BASE_URL": "https://api.test.myapp.com" }` if needed for some tool calls. Or keys: better to prompt user to set locally if needed. - **Hooks:** You can define scripts to run at certain events (SessionStart, BeforeEdit, AfterEdit, etc.) [101] . Useful for enforcing guardrails or auto-formatting: - e.g., a SessionStart hook to set up environment or print a banner. - e.g., AfterFileSave hook to run `npm run lint:fix` on the file changed, so code is always formatted (but careful it might confuse Claude if file content changes under it). - **Plugins/MCP:** If using Claude plugins (like connecting to marketplace or internal MCP servers), configure them here (enabledPlugins, extraKnownMarketplaces, etc) [131] [132] . - **Model Choice:** Possibly can set default model in settings. E.g., if you have access to `claude-opus-4` vs `sonnet-4` , etc., you might specify in config if needed (though usually choose on run or skill basis). - **System Prompt adjustments**: There's a "system prompt" concept (Claude's underlying persona). Usually leave default, but settings might allow customizing it or adding policies. Usually not needed if CLAUDE.md covers instructions.

Best Practices: - Keep `.claude/settings.json` in source control as the single source of truth for Claude config for project [128] . - Use it to enforce team decisions: e.g., if team says "Claude should never commit to main", encode that in permissions (deny push to main). - Document within the file via comments (if format allows, it's JSON though so official comments not allowed - maybe mention in README). - Double-check it into git early when setting up Claude for project. Many may forget to share config, leading to inconsistent local settings. The settings.json ensures uniform behavior.

Failure Modes: - *Misconfigured allow/deny:* If you deny too much, Claude becomes hamstrung (e.g., denying Edit means it will ask constantly). If you allow too much (like dangerously-skip), you might have safety incidents. So test the config with a sample session. - *Local overrides overshadowing project:* If devs have their own `~/.claude/settings.json` with conflicting rules, project config should override (project has higher precedence) [133] . But if not careful, might get weird interactions. Understand scope precedence (Enterprise > CLI args > Local > Project > User). - *Forgetting local secrets:* If an env var needed for a tool (like GH CLI needs token), and you don't provide it, Claude might be confused why `gh` isn't working. Provide guidance either via prompt or documentation to devs to set those in local settings or OS env. - *Overriding by accident:* If someone commits a too-open permission (like always allow all bash), they might open risk. Code review changes to settings.json too, treat it as sensitive config (like you would review changes to CI config). - *Version drift:* If different branches might have different config needs (rare, but e.g., you adopt a new plugin in a branch), merging could conflict. Manage changes to this file carefully in PRs to avoid stomping config inadvertently.

`.claude/settings.local.json` (Personal Overrides)

Purpose: This file (in `.claude/` but gitignored by default) is for each developer's personal preferences or sensitive info that should not be shared [134] [135].

For example: - Setting a personal theme or editor preferences if Claude integration with your editor. - Overriding allowedTools for your environment (though better to standardize in project if possible). - Storing personal API keys or tokens (e.g., OpenAI key if needed for some plugin, or GitHub token for GH CLI). We strongly avoid committing those, so local is the spot. - You might also temporarily loosen a guardrail for debugging something without affecting team (though be cautious; ideally guardrails apply to all to avoid accidental dangerous usage).

Best Practices: - Keep local settings minimal. If you find yourself adding something that multiple devs likely need, consider pushing it to project settings (except secrets). - Use it for things that either cannot be in git (like secrets) or that are truly personal (like maybe you want Claude to always use a certain writing style in commit messages, which team doesn't care about). - Document what can go in local. Maybe in a `CLAUDE.md` or contributor guide, say "Put your API tokens in .claude/settings.local.json as `{\"env\": {\"GH_TOKEN\": \"...\"}}` to enable GH CLI usage." This helps onboard new team members using Claude. - The first time `.claude/settings.local.json` is created by Claude or you, ensure it's in .gitignore (Claude Code does that automatically when created) [136].

Failure Modes: - *Configuration drift:* If one dev changes a setting locally (like allows a tool always that project disallowed), their Claude might do something others wouldn't. Could cause non-reproducible outcomes. Encourage adherence to project config and use local only for things that don't affect output (like credentials or UI preferences). - *Accidentally committing local values:* It's ignored by default. But if someone manually renames or misplaces, risk pushing secrets. Unlikely if they follow defaults. Just ensure `.claude/settings.local.json` stays in .gitignore, and instruct not to circumvent it. - *Lack of visibility:* If a dev overrides something critical locally (like turning off some safety), others won't know. This could lead to "works on my machine" issues. Ideally, there should be little need to override safety settings. If they feel the need to, perhaps project config should be adjusted or they should discuss with team.

`.claude/commands/` (Custom Slash Commands)

Purpose: In this directory, you can create pre-defined prompt templates (as Markdown files) that appear as slash-commands in Claude's interface [35]. This is powerful for repeated tasks or prompting patterns.

For example, you might have: - `.claude/commands/review.md` – containing something like:

```
Please review the current changes for clarity, potential bugs, and adherence
to our style guide.
```

Then in chat, by typing `/review`, Claude will apply that prompt. - `.claude/commands/port-module.md` – a more complex multi-step template for migrating a module (embedding your known process). - These commands can take `$ARGUMENTS` placeholder for runtime parameters [137]. E.g., `commands/fix_issue.md`:

```
Please analyze and fix the GitHub issue: $ARGUMENTS.
```

Then triggering `/fix_issue ISSUE-123` plugs that in (like example from Anthropic doc) [138].

Best Practices: - Use commands for things you find yourself typing often to Claude. E.g., environment setup instructions, or "Explain this code" for new devs, etc. - Keep each command focused. It's basically a saved prompt. It can be multi-line too if need steps. - Since commands are just MD files, they can be reviewed by team. So store only non-sensitive info. (No secrets in a command file, as they'd be committed). - Leverage argument feature: It makes commands flexible. If you have a command for analyzing an error log, you could have `$ARGUMENTS` in it and then pass the log file path. - Document the commands available (maybe in README or `CLAUDE.md` add a section listing them) so team knows and can use them.

Example: If the project often needs code style cleanup: `commands/cleanup.md`:

```
$ARGUMENTS
```

(This one might be used by selecting code and running /cleanup to have Claude format or simplify it - though it's blank here, it could instruct "Make the above code idiomatic", etc.)

Failure Modes: - *Stale commands:* If the command text references something outdated ("fix known bug in version 1.2" after version bump, etc.). Review commands periodically. - *Over-reliance or misuse:* If commands are too generic, dev might trigger them without fully understanding them. E.g., a `/optimize` command that tries to micro-optimize code could backfire if used recklessly. So design commands carefully and maybe restrict if needed (permissions apply to actions, not text, so careful what command instructs). - *Not using them:* Team forgets they exist. Mitigate by publicizing and maybe adding help – perhaps a command or note in CLAUDE.md summarizing all available commands.

Commands vs skills: - Commands are user-invoked manually, best for explicit tasks developer chooses (like "generate scaffolding" or "explain code"). - They do not trigger automatically and thus are safer in that sense; they only run when someone thinks to use them.

`.claude/skills/` (Agent Skills)

Purpose: Skills are like stored knowledge or behaviors that Claude can automatically invoke when relevant, without the user explicitly asking for them [36] [139]. Each skill is a directory containing a `SKILL.md` which defines what it does and when to apply it (via description metadata) [140] [141].

Use cases: - "Review PR according to our internal checklist" skill. When the user asks "review this PR", Claude might use that skill to ensure it covers all checklist items [36]. - "Database query expert" skill might contain instructions and scripts to allow Claude to run certain DB analysis when asked. - "Secure coding guidelines" skill that triggers if user asks something that matches security context – to remind or apply rules.

Designing Skills: - Each skill's SKILL.md has YAML frontmatter with at least `name` and `description` that tells Claude when it should use it [142] [143]. - The description is crucial: make it include keywords a user might say that should trigger it. E.g.,

```
name: code-review
description: Provides thorough code review feedback following our team's
guidelines. Use when user asks for a code review or feedback on code quality.
```

Then in SKILL.md body, list the steps to review code, maybe referencing the guidelines doc link or enumerating common checks (naming, complexity, etc). - Scope: Put sensitive or heavy knowledge as progressive disclosure inside the skill directory (like reference files) so they load only if needed [144] [145]. Skills support linking additional files that only load when referenced to keep context slim until needed. - Allowed tools: A skill can also have an `allowed-tools` list in YAML to broaden (or narrow) tool usage when skill is active [146] [147]. E.g., a "read-files" skill might allow `Read` tool extensively. - Testing skills: Skills auto-trigger, so to test, phrasing a request similar to description should make Claude ask "Use skill X?" (It typically asks for confirmation, can auto-confirm if in non-confirm mode). If not triggering, maybe adjust description keywords.

Best Practices: - Only create skills for recurring patterns where automatic help is beneficial. Too many skills might conflict or cause unexpected triggers. - Keep skill instructions clear and focused on that context. They essentially modify Claude's behavior while active, so be specific. In SKILL.md, you can give step-by-step or bullet instructions that Claude then follows when skill invoked. - Use progressive disclosure within skills if they have a lot of reference material (as Anthropic suggests bundling reference.md, examples.md, etc) [144] [148]. That way context is managed smartly. - Limit tools per skill if appropriate (via allowed-tools) to prevent misuse. E.g., a skill that queries a database could explicitly only allow the safe query tool. - Version control: if a skill logic changes (maybe you refine code review checklist), update SKILL.md accordingly so AI uses updated guidelines.

Failure Modes: - *False triggers:* If description is too broad or overlaps, Claude might attempt a skill when not needed. E.g., a "explain code" skill that triggers on any 'how does this work' question might sometimes be not needed if user wanted a simpler answer. Fine-tune descriptions and maybe combine with some require keywords. - *Skill conflicts:* Two skills might both trigger on similar user request. Claude might get confused which to pick. For instance, a "security review skill" and a "general code review skill" both relevant to "review this code for any issues" request. The dev might need to refine or disable one contextually. Claude tends to choose one and ask, but if both match, might pick arbitrarily or less optimal. - *Stale skills:* If a skill references outdated guidelines (like old coding standard after you changed it), it may advise incorrectly. Keep them updated like CLAUDE.md. - *Implicit behavior:* Skills make Claude do things without user explicitly instructing each step. It's powerful but sometimes can surprise the user. Always confirm skill usage when possible (Claude asks confirmation by default, which is good). If the team finds an auto-skill doing unwanted actions, consider disabling (removing from config or adjusting description to be stricter). - *Complex debugging:* If something goes wrong only when a skill triggers, it might be less obvious than a direct prompt. However, Claude does typically output "I have a skill that matches, do you want me to use it?" which is some transparency. If running headless or with auto-confirm, it might do it without telling, which can be harder to trace. So log usage and keep skill logic simple to debug.

`.claude/agents/` (Custom Sub-Agents)

Purpose: This folder is for defining specialized sub-agents. Each sub-agent runs in isolation with possibly a different context or tool set [149]. It's like spawning another Claude instance with narrower focus or permissions. Useful for delegation patterns: - E.g., a sub-agent "DBAnalyzer" that has access to production DB read-only and analyzes something while main agent focuses on code changes. - Or an agent "UITester" that can use a Puppeteer MCP to navigate UI, separate from main code agent.

Each sub-agent is defined by a Markdown file with YAML frontmatter, similar to skills, but for an agent config:

```
name: legacy-analyzer
description: Expert at analyzing legacy code patterns, business logic, and
regs.
tools: Read, Grep, Glob  # restrict tools
---
You are a specialist in legacy code analysis...
```

[150] [151]

Design: - Determine tasks that benefit from an isolated conversation and maybe different allowed tools or expanded context. (Subagents don't share history with main, so they get a fresh context window). - Define their allowed tools in frontmatter (the `tools:` field), and they will only have those instead of full main tools. - Provide them a role prompt in the body (like "Focus on X, ignore Y, output summary only"). - Then in main conversation, you can instruct main Claude to use a sub-agent for a subtask: `Use the legacy-analyzer subagent to analyze BILLING-CALC.COB...` as seen in the tribe AI example [152]. - Or call via code: some commands to spawn agent tasks in Claude's CLI might exist (like a `Task` tool usage spawns the subagent).

Best Practices: - Use sub-agents to reduce context size and not pollute main conversation with details. E.g., let sub-agent read huge file and come back with summary rather than loading that entire file into main context (like Szymon's example of saving tokens) [17] [18]. - Keep sub-agent tasks well-defined. Because coordinating multiple agents can be complex, do it for separable concerns. E.g., one agent per microservice analysis if doing architecture review. - Limit sub-agent permissions to what they need. If one only needs read, don't give it edit or internet. This containment is an advantage (less worry it will do side-effects). - Sub-agent definitions can include skills and rules presumably (they have their own context, likely they will load relevant CLAUDE.md too unless configured not to). - Name them clearly and describe in YAML so main agent knows when to use them or you instruct manually.

Failure Modes: - *Complex orchestration:* Getting results from subagents back to main might be clunky. Typically, the main agent reads the sub-agent's output (which might appear in conversation as separate thread or something). If doing it in CLI, there's a mechanism (like main triggers, sub outputs summary, main reads it) [152] [153]. Ensure that loop closes properly. - *Context isolation pitfalls:* The sub-agent won't know main conv details by default (that's the point). But if needed, you might need to feed it some initial context. Possibly through its invocation (like telling subagent to open certain file and analyze, as the example does). - *Overhead:* Running multiple agents uses more resources (each has its own model context and usage tokens). Use when payoff is worth it (like analyzing tons of code offline then returning 2k summary). - *Agent sprawl:* If you define too many specialized agents, it can be confusing. Start with a few high-value ones. Over time, see if team uses them; prune if not. - *Staleness:* Like others, if an agent's prompt contains guidance (like "comply with architecture X"), update it when architecture changes.

`.claude/rules/` (Modular Instruction Files)

Purpose: This directory is for breaking up instructions or guidelines into pieces that can be discovered recursively by Claude. It's essentially an alternative to cramming everything in one CLAUDE.md. For example: - Instead of a huge CLAUDE.md, you might have `rules/testing.md`, `rules/`

`security.md`, etc. - Claude when working might find and read these as needed (especially if linked or if a tool "Rules" fetches them). - Another usage: if you have distinct sets of guidelines for different contexts, storing them as separate rule files can let you easily include/exclude them by linking.

How Claude finds them: Possibly if you mention in CLAUDE.md "see rules/security.md for details", Claude will then read rules/security.md only when relevant (progressive disclosure concept) [154] [155]. Also, Claude Code might automatically load `.claude/rules/*.md` if it has some logic to do so (less clear if auto unless referenced).

Replacing monolithic CLAUDE.md: The prompt says "replacing monolithic CLAUDE.md files" in favor of rules. The idea: - Instead of one mega file, have a root CLAUDE.md that outlines structure and references specific rules files. - E.g., CLAUDE.md might contain:

```
# Coding Guidelines
This project follows strict guidelines. See [rules/style.md](rules/style.md)
and [rules/performance.md](rules/performance.md) for details.
```

Claude will know those links and can read them when needed (like if code style is relevant). - This modular approach makes maintenance easier (you edit one aspect in its file without wading through huge MD). - Also fosters reuse: e.g., maybe certain rules could be shared across repos by copying files.

Recursive discovery patterns: likely means linking within rules as well (one rule file might link another). - Limit depth though, as anthropic docs caution that deep nested references might partially load unpredictably [156]. So best to link one level deep.

Best Practices: - Organize rule files by domain (style, security, etc.). - Keep each reasonably short and targeted. - In CLAUDE.md (or main rules index), link to them by name so Claude knows to open them when needed. - Use headings in rules file to help Claude navigate (if it opens it fully). - Possibly use an index file in rules/ listing them. Or at least consistent naming.

Failure Modes: - *Claude not reading a rule when needed:* If it doesn't realize a rule is relevant, it might skip it. Possibly mention explicitly in query or ensure triggers. E.g., if asking for a security review, maybe instruct "Follow our security rules (see rules/security.md)". Over time, the model might learn to check those if description hints. - *Rules conflicting or overlapping:* If one rule file says "Never do X" and another says "Sometimes do X", clarifying context needed. Ideally avoid such conflicts by scoping clearly. - *Maintenance overhead:* Could become scattered. If the team is small, sometimes one CLAUDE.md is simpler. But in large codebase, likely beneficial to modularize knowledge.

Example .claude/ Structure:

Finally, let's present an example layout for a hypothetical large monorepo:

```
/CLAUDE.md               # root guidelines (general info, overarching rules)
  (e.g., "Use our style guide (see .claude/rules/style.md). Always run tests
before commit. Important: Domain logic goes in services, not controllers.")
/.claude/
    settings.json        # configure allowed tools, etc.
    settings.local.json  # (gitignored) developer-specific secrets or prefs
```

```
    commands/
       review.md          # /review command to do code review via skill or
instructions
       apply-pattern.md   # e.g., /apply-pattern command for adding typical
code pattern
    skills/
       code-review/
           SKILL.md       # Auto skill to enforce code review standards
           checklist.md   # (referenced by SKILL.md) contains list of items
to check
       sql-query/
           SKILL.md       # Maybe a skill to safely run SQL queries if asked
           reference.md   # e.g., contains schema definitions loaded on
demand
    agents/
       legacy-analyzer.md # sub-agent config for analyzing legacy mainframe
code, with limited tools
       security-audit.md  # sub-agent to do security scans using certain
tools
    rules/
       style.md           # coding style rules (indentation, naming, etc.)
       testing.md         # guidelines for writing tests (prefer TDD, etc.)
       security.md        # secure coding rules (no hardcoded creds, validate
inputs, etc.)
       performance.md     # performance best practices (complexity limits,
etc.)
       architecture.md    # high-level architecture rules (layer boundaries,
module responsibilities)
```

In the above: - The root CLAUDE.md might mention high-level items and link to rules files:

```
# Project Claude Guide
- Follow our [Style rules](.claude/rules/style.md) for all code.
- Ensure security considerations per [security guidelines](.claude/rules/
security.md).
- ...
```

- `settings.json` might contain permission customizations (like allow gh CLI, puppet server, etc. while denying any dangerous ops). - `commands/` holds a couple helpful slash commands. - `skills/` has two skills: one to automatically do code review (so when user says "review code", it triggers structured approach), one to handle SQL queries (maybe if user asks a data question, it triggers skill that safely queries DB). - `agents/` two sub-agents for specialized tasks. - `rules/` multiple focused rule files to keep CLAUDE.md trim.

This structure allows Claude Code to operate with rich context but modularly: - On startup, root CLAUDE.md (with links to rules) loads, giving baseline. - If code style enforcement needed, Claude can open rules/style.md when formatting code. - If doing a security-related change, it might open rules/security.md. - If user invokes `/review`, maybe it leverages the code-review skill or command to apply checklists (the skill might contain references to rules). - If analyzing legacy, user can spawn `legacy-`

`analyzer` sub-agent as defined. - The `security-audit` agent might be used via a slash command to run certain scanning outside main context.

Common Pitfall to note: Make sure the team is aware of this structure. It's new and could be overlooked if someone new comes in. Ideally, include a section in the repository README or contributor docs explaining what `.claude/` is and how to use these features (so they can fully leverage them rather than ignore).

By configuring Claude Code in this structured way, the project encodes a lot of knowledge and guardrails that guide the AI to be a productive, safe partner in development, aligned with the team's practices and architecture.

I. Commands vs Skills vs Agents (Decision Framework)

Claude Code provides three primary extension mechanisms for tailoring AI behavior: - **Commands** (manual slash commands triggered by the user), - **Skills** (auto-triggered instructions based on context), - **Agents** (delegated sub-agents with isolated context/tools).

Choosing which to use for a given need is important to keep the AI assistance effective and maintainable. Let's outline a clear decision matrix and guidelines:

Type: Invocation: Best For: (and what it's not best for)

- **Slash Commands** – *Invoked manually by user via* `/command`.
- **Best For:** Explicit actions that the developer wants to trigger on demand. Great for tasks that don't need to happen automatically but are often repetitive. For example, formatting code, generating boilerplate, running a diagnostic, initiating a particular multi-step prompt sequence.
- It ensures human is in control: nothing happens unless they call it. So it's ideal when you want predictability or are not sure if it should always be done.
- **Examples:** `/create-tests` to generate test stubs for a given code, `/docstring` to add documentation to a function. The user chooses when to apply these.

- **Misuse:** Using commands for something that should be automatic – e.g., if every time you ask for a review you always run a command, a skill might be better. Also, too many specialized commands might overwhelm or be forgotten. Keep the list curated.

- **Skills** – *Triggered automatically by Claude when relevant (semantic match on the user's request), with confirmation.*

- **Best For:** Providing Claude with specialized knowledge or procedures that augment its responses whenever applicable, without the user specifically asking for that skill. This is great for embedding domain expertise or compliance checks. For instance, a "RegulatoryCompliance" skill could automatically trigger when code in a certain module is discussed, injecting rules that must be followed [36]. Or a "CompanyStyleGuide" skill triggers whenever user says "explain this code," ensuring the explanation follows company style.
- Essentially, skills are for things you want the AI to consistently do or consider whenever certain topics arise, without relying on the user to prompt every time. They make the AI smarter in context.
- **Examples:** A skill that automatically formats explanations in Markdown if user is in a Markdown file, or a skill that whenever user asks to "optimize code", it loads known optimization tips.

- **Misuse:** Overusing skills can lead to implicit behaviors that surprise users. If you pack a lot of logic into skills that fire without user realizing, it could make debugging weird outputs harder (like "why did it say that? oh, some skill triggered"). So do not encode complex decision-making as skills unless you're confident it's always correct to do so.

- Another misuse: skills that overlap with each other or with base Claude capabilities might cause duplication or confusion. Also, if a skill triggers too broadly (due to a generic description), it might activate when not needed.

- **Sub-Agents** – *Launched explicitly (by user request or Claude's suggestion) as separate Claude instances with isolated context and tool permissions.*

- **Best For:** Complex or lengthy tasks that can be partitioned, tasks requiring different permissions, or parallel tasks. They shine when you need isolation either to preserve main context or because the subtask is very different in nature. Also great for using different tool sets – e.g., a sub-agent might have access to a production database read replica, which you wouldn't give the main agent normally.
- They are essentially "child processes" for the AI. Use when a task would otherwise blow up the token context or when independence is needed.
- **Examples:** Using a sub-agent to thoroughly analyze a large log file or codebase section and return a summary (so that the huge content doesn't pollute main conversation) [17] [18] . Another: a sub-agent for UI that actually runs a headless browser to verify UI output, while main agent writes code – they communicate via intermediate results.
- **Misuse:** Overusing sub-agents can complicate the workflow. If you spawn sub-agents for trivial things, you fragment context and have overhead. Also, too many sub-agents might be hard to coordinate.
- If not careful, can cause fragmentation of knowledge – one agent might figure out something but not effectively convey it back. So use sub-agents when truly necessary, and ensure they report results clearly for main agent to integrate.
- Also, note that sub-agents currently likely don't share memory unless explicitly passed, so don't expect them to "know" project context beyond what you feed.

Balancing All Three (Vibe Engineering Balance):

- **Commands** give direct control: you use them like tools.
- **Skills** give proactive enhancements: the AI does better outputs spontaneously when relevant.
- **Agents** let you scale or isolate tasks: the AI can multitask or handle specialized duties safely.

A balanced approach (vibe engineering) might be: - Use skills to encode your static knowledge and style (so AI always follows them). Example: a skill with your code style ensures all code suggestions adhere without you prompting every time – raising baseline quality. - Use commands for specific one-off or user-driven flows. Example: after a big change, you run `/run-tests` command to have Claude run all tests and report (if not automatic). Or a command to generate a boilerplate new microservice structure (since you might do that rarely). - Use sub-agents sparingly but powerfully: maybe one sub-agent to monitor something in background or handle a heavy analysis concurrently with code writing (some advanced flows have one agent writing code and another reviewing at same time) [77] [157] .

Common Misuses Recap: - Too many sub-agents: fragmentation, overhead. If one agent could do sequentially, no need for multiple. Sub-agents aren't needed for small scopes (just use main agent with correct context). - Too many skills: can lead to unpredictable or lengthy responses as multiple skills try to inject info. Also debugging which skill said what is hard. Only add skills that clearly benefit many interactions frequently. - Using commands for everything: defeats AI autonomy. If you rely only on

manual commands, you're not leveraging the AI's ability to improve output on its own. E.g., if you always have to type `/format-code`, maybe a skill that auto-formats code suggestions would save a step. - Conversely, letting skills do what user should explicitly decide: e.g., an "auto-commit" skill that triggers after code generation to commit code. That might be dangerous if it commits without user okay. That should be a command instead (so user explicitly triggers commit).

Decision Matrix Summarized:

Scenario/Need	Use Command?	Use Skill?	Use Sub-Agent?
User wants a specific action on demand (one-time or occasional) – e.g., generate docs for this function now.	Yes – manual slash command fits. User can trigger exactly when desired.	No – skill would try to do it automatically every time, unnecessary.	No – sub-agent overkill for a one-off user request.
Enforcing consistent behavior in AI outputs always (style, standards) without user reminding each time.	No – user might forget to call command every time, and it's repetitive.	Yes – a skill that monitors outputs and adjusts or adds info is ideal. It applies whenever context matches.	No – not a separate task, it's a modification of main AI behavior, which skills handle.
Multi-step or lengthy process that you want to encapsulate for reuse (like a known sequence of prompts).	Possibly – a slash command can contain a multi-step prompt template for Claude to follow. If the user explicitly wants to run that process, command is good.	Maybe – if that process should happen whenever certain conditions arise without user asking (rare, usually user knows when to run a multi-step).	Maybe – if the process is so heavy it merits an isolated agent (but if it's sequential prompts, a command might suffice with main agent).
Complex analysis that can run in parallel or separate from main coding context (e.g., analyzing 100k lines of logs).	No – a command can start it, but main agent doing it will blow context. Command alone doesn't isolate context.	No – skill still runs in main context, not ideal for heavy analysis needing separate memory.	Yes – spawn a specialized sub-agent for log analysis, so main agent stays free for other work and context isn't filled with log data. The sub-agent returns summary back.
Needing a different set of permissions or environment for part of a task (e.g., one step requires internet or prod DB access that you don't want to give to main agent).	No – command can invoke something, but main agent still lacking permission means it might fail or be blocked.	No – skill won't override tool permissions in main agent beyond what allowed.	Yes – define a sub-agent with those specific permissions or an MCP to handle that step. Use it just for that task to keep main secure.

Scenario/Need	Use Command?	Use Skill?	Use Sub-Agent?
Team wants AI to automatically apply certain checks or transformations in background (like always optimize queries in suggestions if possible).	No – user can't realistically command this every single suggestion.	Yes – a skill could detect when SQL code is being written and ensure it follows optimization rules by injecting guidance.	No – not separate from main conversation, it's part of it. Skills are appropriate.
Isolating responsibilities to prevent context dilution (e.g., verifying code with a separate agent).	Possibly – one could manually use a command to ask main Claude to verify code, but then main context has to hold code & test outputs.	No – skill would still do it in same context.	Yes – as Anthropic suggests multi-Claude: one agent writes code, another reviews [77]. This is implemented via sub-agents or multiple sessions. Each has separate context so they don't exhaust one context window with both writing and reviewing.

Essentially, use **commands** when you want *explicit control* and one-off invocations, **skills** when you want *implicit enhancement* of AI output *consistently*, and **sub-agents** when you need *isolation or parallelism* for complex tasks.

Balance is key: *vibe engineering* is about disciplined use of these. We want the AI to help in a structured way: - Rely on *skills* to encode your standards so the AI naturally adheres to them (reducing human correction needed). - Use *commands* to direct the AI through well-defined workflows where human oversight at each invocation is beneficial (like initiating tests or generating a scaffold). - Use *agents* when one AI instance should not (or cannot) do it all – to avoid context overload or to segregate duties (like a safety measure, giving a sub-agent limited access to do a risky operation so main never touches it).

One has to avoid going overboard: - If everything is a skill, the AI might become too rigid or do unwanted things automatically – keep core rules as skills, but not every minor preference. - If everything is a sub-agent, then you're constantly juggling outputs – not efficient. Use them for major separations only.

Think of it like this: - *Commands* = user manual triggers (like calling a function in code). - *Skills* = background services or interceptors that run automatically (like an AOP or middleware hooking in). - *Agents* = separate threads or microservices delegated specialized tasks.

A mature use of Claude Code will combine all three: letting skills handle the routine enforcement, commands for explicit dev wishes, and sub-agents for heavy lifting aside from the main flow. The outcome: the AI feels integrated into the dev workflow, not running off wildly (no vibe coding chaos), but also not needing constant babysitting for known patterns (since skills cover those), and capable of scaling to big problems by dividing work (via agents).

J. Evaluation & Metrics

Adopting Claude Code (or any AI pair programmer) should be treated as an experiment that needs evaluation. We want to measure its impact on speed, quality, and other factors in a data-driven way. This section defines a lightweight evaluation framework for a 2-week (or similarly short trial) to assess Claude Code's performance in our team, focusing on: - **Speed** (cycle time, number of iterations to complete tasks), - **Quality** (defects, test outcomes, code review findings), - **Review Burden** (effort required by human reviewers), - **Reliability** (frequency of AI errors or tool issues), - **Safety Incidents** (any misuse or security concerns).

We will outline metrics for each, how to collect them, and present a "trial scorecard" template.

Metrics and How to Measure:

- **Development Speed:**
- *Lead Time per Task:* Compare how long it takes from task start (e.g., ticket open or branch creation) to completion (PR merged) for AI-assisted tasks vs historically without AI. In the 2-week trial, we can sample a few similar tasks done with Claude and note their durations. e.g., "Bug fix 123 took 1 day with Claude, prior similar bug fixes averaged 2 days." [3] [48] .
- *Number of Iterations/Prompts:* Track how many back-and-forth messages with Claude were needed for a given feature/fix. Fewer may indicate efficiency (though sometimes more prompts might catch more issues early, which could be fine). Still, it's good to see if tasks get done in, say, 3 prompt iterations on average or 30 (which would indicate inefficiency).

- Possibly measure "time spent coding vs time spent guiding AI" qualitatively via dev logs or self-report.

- **Quality:**

- *Test Results:* During trial, note any new test failures introduced by AI commits (if CI fails due to AI-generated code) [5] [115] . Also, measure code coverage if available to see if it maintained/increased (AI often writes tests when asked).
- *Bugs/Defects:* Count any bugs found in AI-generated code during review or after merge. E.g., if within the 2 weeks some AI-authored code had to be fixed or rolled back due to a bug, record that.
- *Lint/Static Analysis Warnings:* See if AI code triggers new warnings or not. Ideally, we want minimal new lint issues, or if any, how quickly they were resolved. If AI consistently produces clean code with no warnings, that's a quality win.

- Perhaps track complexity metrics (does AI code tend to be more complex or about same as human code? Might be subjective for now).

- **Review Burden:**

- *Review Time:* Reviewers can log how long they spend reviewing AI PRs vs normal PRs of similar size. Even rough estimates: "Spent 30 min on AI PR, usually such PRs take 15 min" or vice versa. Or count number of review cycles (comments and rework iterations).
- *Number of Comments:* If AI PRs consistently get more review comments or major change requests, that suggests extra burden. Or if they sail through with minor nitpicks, burden is low.
- The Osmani reference indicated AI heavy PRs took ~26% longer to review on average; we can measure something similar in our context.

- *Understanding/Confidence:* Consider a quick survey of reviewers: do they feel less confident in AI code requiring more thorough review (score 1-5)? This is qualitative but useful.

- **Reliability (Tool/Process Reliability):**

- *Claude Errors or Hiccups:* Count any times Claude gave a fundamentally wrong answer or needed multiple attempts (e.g., it hallucinated an API that doesn't exist, or got stuck) and how that impacted flow. Or any technical issues using the CLI (crashes, etc.).
- *Permission Denials or Workarounds:* If the tool repeatedly needed user to override something or do manual steps because it couldn't, note how often and why. That indicates friction.
- *Stability of suggestions:* Did we encounter any unpredictable or inconsistent behavior from Claude that wasted time? For instance, gave one solution then later contradicted it.

- *Integration reliability:* Did commands and skills work reliably? E.g., any skill mis-trigger that caused confusion, etc.

- **Safety/Compliance Incidents:**

- *Policy Violations:* Did Claude ever do something disallowed (use a denied tool, attempt to reveal secret, produce inappropriate content)? Hopefully none due to guardrails, but track if any occurred.
- *Security Issues in Output:* e.g., did it suggest insecure code that had to be caught? If within trial it never introduced an obvious vuln (like SQL injection vulnerability), that's good. If it did and we caught it, mark that.
- *User trust/satisfaction:* Not exactly safety, but see if any dev felt uneasy about certain AI outputs, maybe a sign to adjust rules or clarify instructions.

We should gather data in a simple table or list for each category after 2 weeks. Could use a spreadsheet or just text summary. Because this is short, we may rely on qualitative feedback combined with a few quantitative points.

Example Scorecard for 2-week trial:

Metric	Observation (Claude Code Trial)	Baseline (Before AI or expectation)	Notes
Lead time per feature	Feature X: 2 days (Claude)	Feature X-like historically ~3-4 days	~30% faster, primarily due to faster coding [3].
Lead time per bugfix	Bug Y: 4 hours (Claude)	Similar bug ~8 hours before	1-day saved.
Prompt iterations per task	Feature X: 5 Claude exchanges; Bug Y: 3 exchanges	-	Seems efficient; didn't require extensive back-and-forth.
Test failures introduced	1 minor test failed on first run, Claude fixed in same PR	Typically 0-1 (dev fixes before PR)	On par; it fixed quickly when prompted.
Post-merge bugs	0 observed in trial period from AI code	Baseline maybe 1 bug per 5 tasks (varies)	Good (no regressions so far).

Metric	Observation (Claude Code Trial)	Baseline (Before AI or expectation)	Notes
Lint warnings	2 minor style nits (spacing) in one PR	Baseline human PRs often ~2-3 nits	Similar; easily auto-fixed.
Review time per PR	~1.5x longer on first few AI PRs (reviewers double-checked thoroughly)	- (subjective)	Reviewers cautious initially; time may drop as trust increases.
Review comments	AI PRs: average 4 comments, mostly style or clarifications	Baseline similar-sized PR ~3 comments	Slightly higher, likely due to needing to understand AI decisions.
Claude tool issues	1 instance: Claude stuck in a loop editing same line, had to intervene	-	Possibly due to unclear prompt; resolved by refined instruction.
Permission blocks	None that hindered progress (once needed to run `npm install`, user allowed manually)	-	Minor friction, likely fix by updating allowed tools.
Safety incidents	None – Claude adhered to guardrails (didn't expose any secret or use banned ops)	-	Guardrails effective.
AI suggestions quality	Dev feedback: "90% of Claude's code was production-quality; needed small tweaks"	-	Positive – indicates high utility.
Developer satisfaction	Team poll: 4/5 average satisfaction with AI assistance	-	Comments: sped up boring tasks, some learning curve.

(The above is an illustrative example. Actual metrics would depend on what tasks were done in that period.)

Using the scorecard: At the end of trial, we can discuss: - Did speed improvements outweigh overhead? e.g., perhaps we see faster completion times, albeit with slightly more review effort. If net positive (and likely to improve as team/AI get better), that's promising. - Quality looks maintained or even improved (no new bugs, tests were thorough, etc.). - If review burden was significantly higher, maybe we identify why (trust, or AI style differences) and address that (maybe update CLAUDE.md to align style, or simply expect it to reduce as familiarity grows). - Reliability issues: maybe one or two, but if nothing major, it's stable enough for more use. - Safety: no incidents -> good. If there was an issue (like it once tried to commit a secret by mistake but was denied), we'd note to strengthen that guardrail or remind users.

From these findings, we can decide: - Proceed with broader use of Claude Code (maybe start using it on all tasks, not just trial selection). - Or if some metrics were concerning (e.g., if code quality had dipped or review burden is too high), address those: e.g., more skills to ensure consistency, or more training for devs on how to best use prompts.

For ongoing evaluation beyond trial: - Could integrate some metrics into retrospectives (e.g., count how many tasks used AI, how it impacted velocity). - Possibly track defect rate from AI code vs non-AI code

over a quarter. - Developer sentiment can be gauged periodically (maybe after initial novelty, see if they still find it beneficial or if any fatigue/issues).

But for a quick trial, the above categories suffice.

Trial Scorecard Template:

(We could provide a simple table as above to fill in, or bullet points under each category.)

For example, an output might be:

- **Speed**: AI-assisted tasks were completed ~30% faster on average. A feature that normally took ~3 days was done in 2 [3], and a bug fix took 4 hours vs ~8 hours historically.
- **Quality**: No post-merge defects found in AI-generated code during the trial. All new and existing tests passed (Claude even wrote an additional 5 tests for features) [48]. Code met style guidelines with only minor adjustments.
- **Review Effort**: Reviewers reported spending slightly more time initially (approx. 50% more on first AI PR), mainly to double-check logic. However, no major rework was needed; comments were minor. As familiarity grew by the second week, review time normalized.
- **Reliability**: Claude Code was generally reliable. One minor hiccup where it repeated an edit loop (resolved by clarifying the prompt). No crashes or major misinterpretations.
- **Safety**: No violations of security or privacy guidelines observed. Claude respected all tool restrictions (e.g., it did not attempt any disallowed commands) and did not introduce insecure code patterns.

This paints a positive picture with data points, giving stakeholders confidence in continuing to use and maybe expand usage of Claude Code, with perhaps a note to keep monitoring review load and to refine processes as needed.

Failure Modes & Recovery Table

Even with best practices, things can go wrong. This table lists common failure modes we've identified throughout this playbook, signs to detect them early, and strategies to recover if they occur:

Failure Mode	Description	Early Warning Signs	Recovery Strategy
Silent Assumptions by AI (Claude makes an assumption not in evidence)	Claude proceeds with an inferred requirement or missing piece (e.g., assumes a variable is always non-null, or invents functionality that user didn't ask for). [158]	AI output includes code or decisions not discussed (e.g., adding a feature "for future extensibility" that wasn't requested). Or it doesn't ask a clarifying question where a human normally would.	Pause and scrutinize AI proposals. If something seems assumed, ask Claude "Why did you do X?" or instruct it to confirm assumptions. Provide the correct information and have it revise. Reinforce guardrail "if unsure, ask" [20] in CLAUDE.md to prevent recurrence. In code review, watch for extraneous code and remove it if unnecessary.

Failure Mode	Description	Early Warning Signs	Recovery Strategy
Over-Broad Edits (AI changes too much or outside intended scope)	Claude modifies parts of code unrelated to the task (perhaps trying to "improve" other areas or refactor broadly). [19]	The diff shows changes in files or functions you didn't expect. Or test failures in areas untouched by the described task.	Use scope guardrails: explicitly tell Claude which files to restrict changes to (or lock others via permissions). If it already happened, revert unrelated changes (manually or via `git checkout` of those segments) and re-run Claude focusing only on intended scope. Reiterate instructions more narrowly ("Only fix the bug in function X, do not alter other modules").
Shallow Verification (AI insufficiently tests/ validates its solution)	Claude says "All good!" without thorough checking (maybe passes simple tests but missed edge cases).	It doesn't run full test suite or ignores a failing test (maybe focusing only on given examples). Or it provides minimal test coverage in new tests (covering only happy path).	Prompt Claude to run all tests (via allowed tools) and to consider edge cases ("Think of edge cases and add tests for them" [50]). Use a sub-agent or second Claude to review the code for untested branches [77] . Ingrain in CLAUDE.md: "after implementing, run full tests and double-check edge cases." If an oversight is discovered later (bug in AI code), write a test for it and have Claude fix the code - treat as learning.
Context Dilution (conversation clutter reduces AI performance)	After long sessions, Claude seems to forget earlier context or brings irrelevant history into answers. [14]	Responses reference past topics that are no longer relevant, or Claude needs repetition of info you gave (sign it lost track). Possibly performance slows or outputs get incoherent.	Clear context regularly (`/clear`) between distinct tasks [14] . Summarize necessary info in CLAUDE.md or a short recap prompt before clearing, then proceed. Use sub-agents for tangents or heavy logs to keep main context clean [17] . If dilution happened, do a context reset and re-provide the essential details for the current task.

Failure Mode	Description	Early Warning Signs	Recovery Strategy
Overconfidence / Over-automation ("vibe slop")	Claude outputs a solution confidently even if it's partially incorrect or it takes actions without user confirmation (due to overly broad skills/commands). [159]	It provides an answer or code that looks polished but might have subtle errors, and it doesn't express any uncertainty. Or an auto-skill executes a change without clearly asking (maybe you notice code changed unexpectedly).	Increase explicit checks: for critical code, ask Claude to explain why its solution is correct [42] . Encourage a habit: "Explain how you verified this." If auto-skills are too aggressive, require confirmation (there's a setting for auto-skill confirm, ensure it's on) or disable the skill. In reviews, maintain healthy skepticism (treat AI code as draft requiring verification) [43] . Possibly dial back "ultra" modes (like not always trust its 'ultrathink' unless needed).
Scope Creep (Task unintentionally expands)	Claude attempts to solve related or perceived issues beyond the original ask (e.g., refactors a whole class when asked to change one method) [28] .	The plan or code includes improvements or changes outside the requested scope ("I also cleaned up the class formatting" or "I noticed another method could be optimized and did it"). While sometimes helpful, it can cause unintended side effects or delays.	Re-focus Claude via prompt: "Do not make any other changes beyond X" [19] . If it already did, either isolate those changes to separate commits for separate review, or roll them back unless they are clearly beneficial and low-risk. Remind in CLAUDE.md that scope creep is not allowed unless asked (maybe a note like "Don't optimize or refactor unrelated code unless explicitly requested").

Failure Mode	Description	Early Warning Signs	Recovery Strategy
Architectural Drift (AI solution diverges from intended architecture)	Claude's output violates architectural boundaries or design patterns (e.g., directly calls database from UI layer, or uses a different design approach that isn't used elsewhere).	Code review shows layering violations (maybe it instantiated a repository in a controller bypassing service layer), or introduction of an unapproved library/ pattern. Possibly tests or static analysis flag an architecture rule break.	If caught early (Claude proposal stage), instruct redesign: "Adhere to our architecture: controllers call services, not DB directly" [19]. If discovered in review, ask for changes or modify the code to fit pattern. Add guardrails: e.g., a skill or rule that lists architectural invariants to always follow (so Claude has that context) [29]. If drift happened unknowingly and merged, treat as tech debt: create a task to re-align it quickly (and maybe adjust CLAUDE.md to prevent recurrence).

This table provides a quick reference: if a developer encounters a given issue, they can identify it and apply the recommended recovery steps. For example, if code reviews start catching that Claude adds extra features (scope creep), the team can tighten prompts and guardrails accordingly.

Finally, a **Curated Reading List** would be presented as sources referenced for further reading or best practice guides that informed our strategies:

Curated Reading List:

- Anthropic Engineering: *"Claude Code: Best practices for agentic coding"* – Official guide with workflows, context management tips, multi-claude usage [1] [4].
- Anthropic Blog: *"Equipping agents for the real world with Agent Skills"* – Introduction to Skills (how to structure SKILL.md, progressive disclosure) [160] [148].
- Szymon Pacholski, *"How I Tame Legacy Code with Claude Code"* – Practitioner's take on using subagents for context and writing CLAUDE.md for style/behavior [17] [20].
- Addy Osmani, *"Treat AI-Generated Code as a Draft"* – Insights on pitfalls of AI code (over-reliance, review challenges) and recommendations for labeling AI contributions and thorough review [32] [42].
- Cognition Team, *"Windsurf Codemaps: Understand Code, Before You Vibe It"* – Discusses "vibe coding" vs disciplined understanding, reinforcing why we need structured approach and how tools like Codemaps help avoid context issues [161] [159].
- David Adam, *"What AI Hype Misses About Real Software Engineering Work"* – Highlights difference between AI success in greenfield vs legacy code and the importance of comprehension (justifies our heavy context and testing focus) [162] [163].
- Official Claude Code Docs (docs.claude.com): *Settings*, *Commands*, *Skills*, *Subagents* sections – Reference for configuration file formats and capabilities (useful when implementing .claude/ config as recommended) [128] [164].

- OpenAI, *"Iterative Prompt Development"* (though about GPT) – relevant for understanding how to refine prompts which we applied implicitly throughout vibe engineering to correct AI when off track.
- Internal Team Retrospective on AI Pilot – (if any internal document was created after a pilot; hypothetical placeholder if a company had their own internal case study to refer to).

Each of these references either provided evidence or frameworks that we integrated into this playbook. They can be read for deeper understanding or alternative techniques (like how others use multi-agent or memory management strategies, etc.).

---

1 2 3 4 5 6 7 12 13 14 15 16 23 24 25 30 34 35 38 40 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 77 82 85 86 88 89 90 91 94 95 96 97 98 102 108 109 110 111 112 113 114 118 119 124 125 126 127 137 138 157 Claude Code Best Practices \ Anthropic
https://www.anthropic.com/engineering/claude-code-best-practices

8 9 64 83 99 117 How does Claude Code manage large codebases beyond its context limit? : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1leclvr/how_does_claude_code_manage_large_codebases/

10 11 84 148 155 160 Equipping agents for the real world with Agent Skills \ Anthropic
https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills

17 18 20 76 How I Tame Legacy Code with Claude Code | by Szymon Pacholski | Nov, 2025 | Medium
https://medium.com/@spacholski99/how-i-tame-legacy-code-with-claude-code-9802c6fc9604

19 26 27 28 29 31 71 72 73 74 78 87 103 104 115 116 158 Claude for Refactoring Legacy Code in Large Projects - Claude AI
https://claude-ai.chat/guides/refactoring-legacy-code/

21 22 80 81 100 101 128 129 130 131 132 133 134 135 136 149 Claude Code settings - Claude Code Docs
https://code.claude.com/docs/en/settings

32 33 42 43 79 120 121 122 123 Treat AI-Generated code as a draft - by Addy Osmani
https://addyo.substack.com/p/treat-ai-generated-code-as-a-draft

36 37 39 41 139 140 141 142 143 144 145 146 147 154 156 164 Agent Skills - Claude Code Docs
https://code.claude.com/docs/en/skills

65 66 75 162 163 What AI Hype Misses About Real Software Engineering Work
https://davidadamojr.com/what-ai-hype-misses-about-real-software-engineering-work/

67 68 69 70 92 93 150 151 152 153 Legacy Code Modernization with Claude Code: Breaking Through Context Window Barriers | Tribe AI
https://www.tribe.ai/applied-ai/legacy-code-modernization-with-claude-code-breaking-through-context-window-barriers

105 How are you treating AI-generated code : r/devsecops - Reddit
https://www.reddit.com/r/devsecops/comments/1nmbc60/how_are_you_treating_aigenerated_code/

106 107 Anyone try Claude Code on a big codebase? : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1j4bof5/anyone_try_claude_code_on_a_big_codebase/

159 161 Cognition | Windsurf Codemaps: Understand Code, Before You Vibe It
https://cognition.ai/blog/codemaps