

PROMPT PACK

SOLO DEVELOPER: 20 AI PROMPTS TO BUILD, TEST & SHIP

Scope, build, test, refactor, and ship production software faster without cutting corners.

26 Oct 2026 · 7 min read

F4F · @USF4F ON X

AI is useful when you give it a clear job. These 20 prompts are built for solo developers and employed builders who need to ship reliable software without cutting corners.

They are starting points, not answers. Give the model your real context, ask it to show its assumptions, and check anything important with a human.

HOW TO USE THIS PACK

- Replace every [bracket] with specific details about your project or codebase.
- Paste in real notes, code, or error logs instead of describing them from memory.
- Ask for options first, then ask the model to challenge its own favourite.
- Keep the output short enough to act on today.

PLAN

1. SCOPE THE FEATURE TO THE BARE MINIMUM

Act as a pragmatic staff engineer. I need to build [feature] for [user role] in [current stack]. Break this down into the smallest milestone that delivers user value. Return a table with three columns: Must-have for V1, Fast follows for V2, and Explicitly excluded. Keep V1 under 4 bullets. Do not invent requirements beyond what I provided.

2. PICK THE SIMPLEST IMPLEMENTATION

Act as a systems architect who values simplicity over cleverness. I need to implement [feature] in [tech stack]. Propose the three simplest viable approaches, ordered from least code to most code. For each, state estimated lines of new code, new dependencies, and main failure mode. Recommend one approach, then challenge your own top pick.

3. SURFACE HIDDEN TECHNICAL DEBT

Act as a site reliability engineer reviewing a new design. I plan to build [feature] by [implementation strategy] using [database or service]. List the top 5 operational edge cases, data consistency risks, or bottlenecks this creates, ranked by severity. Show your assumptions and flag anything you are unsure about.

4. BUILD AN INCREMENTAL EXECUTION PLAN

Act as a technical project lead. I am implementing [feature] across [frontend, backend, or database]. Create a step-by-step checklist where each step takes under 60 minutes, compiles cleanly, and leaves the app testable. Limit this to 6 steps. For each step, provide one manual test to verify it works before moving on.

BUILD

5. GENERATE HIGH-VALUE TESTS

Act as a QA automation specialist. Here is the implementation for [module or function]: [code]. Write a test suite in [test framework] covering: 2 happy paths, 3 boundary conditions, and 3 malformed inputs. Focus strictly on tests that catch regressions in core logic. Do not write mock-heavy tests for trivial getters.

6. SCAFFOLD A MINIMAL IDIOMATIC COMPONENT

Act as a senior [language or framework] developer. Scaffold a minimal implementation of [component or module] that meets these constraints: [inputs, outputs, performance needs]. Follow idiomatic patterns for [framework/version]. Keep the solution under 60 lines of code. Do not add third-party libraries without asking.

7. REFACTOR SAFELY FOR CLARITY

Act as a refactoring specialist. Here is a working but messy function: [code]. Refactor it to improve readability, reduce nesting, and isolate side effects. Present your answer in three parts: a bulleted list of code smells found, the refactored code, and 3 manual checks to confirm no runtime regression occurred.

8. DEFINE STRICT SCHEMA BOUNDARIES

Act as a backend engineer. Define a data contract between [client] and [service] for [use case]. Provide the validation schema in [Zod, Pydantic, or TypeScript types]. Include explicit types, custom error messages, and nullable rules. Reject unknown properties by default. Show your assumptions about optional fields.

DEBUG

9. DECODE AN UNFAMILIAR ERROR

Act as a systems debugging expert. Here is an error stack trace and the relevant snippet: [error and code]. Diagnose the root cause in under 120 words. Do not suggest restarting servers. Give the two most probable causes, ranked by likelihood, and provide the exact diagnostic log line to insert to confirm which it is.

10. TRACE ELUSIVE RACE CONDITIONS

Act as a concurrency and state specialist. Here is the state handling logic for [component or worker]: [code]. We see this intermittent behavior: [unexpected behavior]. Map the exact sequence of asynchronous events or mutations that causes this bug. Then provide the smallest surgical patch to guarantee deterministic execution.

11. OPTIMIZE A SLOW DATABASE QUERY

Act as a database performance engineer. Here is a slow query and the table schema: [query and schema/indexes]. Identify why this query degrades at scale. Propose two distinct fixes: rewrite the query, and specify the exact index to add. Explain the mechanism for both. Do not invent benchmark numbers.

12. CREATE A MINIMAL REPRODUCIBLE EXAMPLE

Act as an open-source maintainer. I hit a bug in [library] when [action]. Here is my application code: [messy excerpt]. Strip this down to an absolute minimal reproducible example in under 30 lines of code with zero external dependencies beyond [library]. The repro must run standalone and reliably fail.

REVIEW

13. AUDIT YOUR OWN DIFF FOR BUGS

Act as an uncompromising principal engineer reviewing code. Here is my git diff for [feature or bugfix]: [diff]. Review it strictly for: silent regressions, unhandled error paths, memory leaks, and missing edge cases. Group issues into 'Must fix before merge' and 'Non-blocking polish'. For each, cite the diff line. No compliments.

14. CHECK SECURITY AND AUTHORIZATION

Act as an application security engineer. Here is an endpoint handling sensitive data: [code]. Audit it against OWASP Top 10 risks, specifically broken authorization, injection, tenant isolation, and leaking secrets. Flag your assumptions about upstream middleware. If no security flaw exists, state that explicitly.

15. VERIFY DEFENSIVE ERROR HANDLING

Act as a backend reliability engineer. Here is a method calling external services: [code]. Audit how errors are caught and logged. Identify swallowed exceptions, unhandled rejections, or error responses that leak internal infrastructure to the client. Rewrite the error blocks to guarantee safe cleanup and structured failure logging.

16. SPOT LEAKY ABSTRACTIONS

Act as a software architect. Here is the module structure for [module name]: [code]. Identify where internal implementation details leak into callers, or where modules are tightly coupled. Suggest 3 concrete refactoring steps, ranked by architectural impact. Recommend your top choice, then challenge your own top pick.

SHIP AND EXPLAIN

17. DRAFT A CLEAN PULL REQUEST DESCRIPTION

Act as an engineering team lead. Based on these commits: [commit log or summary], write a clear pull request description. Use this structure: 'Why this is needed' (2 sentences), 'What changed' (bullet points grouped by subsystem), and 'How to verify' (a 4-step manual testing plan). Keep total length under 200 words.

18. EXPLAIN A TRADEOFF TO A NON-TECHNICAL STAKEHOLDER

Act as a VP of Engineering speaking to non-technical stakeholders. We chose [Option chosen] over [Alternative option] because [reason]. Write a 150-word explanation for [Product Manager or Client]. Use concrete business analogies, explain the impact on timelines and maintenance costs, and avoid technical jargon.

19. WRITE A RELEASE AND ROLLBACK RUNBOOK

Act as a DevOps release engineer. We are deploying [major feature/migration] involving [schema or API changes]. Produce a zero-downtime deployment and verification runbook with three tables: Pre-deployment checks, Step-by-step release execution, and Rollback criteria and procedure. Each step must have an owner, estimated duration, and exact pass/fail condition.

20. PRODUCE AN HONEST CHANGELOG

Act as a technical writer for a developer product. Based on these closed issues: [raw PR summaries], write changelog notes for version [version]. Group items into: Breaking Changes, New Features, and Bug Fixes. For each breaking change, provide a 1-sentence code migration snippet showing old versus new syntax. Do not invent features.

A NOTE ON JUDGMENT

AI can help you ask better questions, but it cannot decide for you. Treat every suggestion, architecture proposal, and code block as an unverified draft that requires your direct review and validation against your production environment. Never share private customer data or proprietary credentials without authorization.

If one of these prompts helps you move forward, tell us which one. We will keep adding resources that are useful, specific, and honest.