VERSION 1

The Builder's AI Prompt Workflow

From single prompt to full structured build

WHAT'S INSIDE

- Part 1 — 6-step prompt workflow for better AI outputs
- Part 1 — 8 anti-patterns that cost you time and rework
- Part 1 — Copy-paste prompt template
- Part 2 — Project-level build strategy across sessions
- Part 2 — Master spec document structure
- Part 2 — Staged document approach — one doc per phase
- Part 2 — Verify-before-advancing rule
- Part 2 — Session constraints and summary patterns

FREE RESOURCE · devclarity.dev

VERSION 1

The Builder's AI Prompt Workflow

From single prompt to full structured build

WHAT'S INSIDE

- Part 1 — 6-step prompt workflow for better AI outputs
- Part 1 — 8 anti-patterns that cost you time and rework
- Part 1 — Copy-paste prompt template
- Part 2 — Project-level build strategy across sessions
- Part 2 — Master spec document structure
- Part 2 — Staged document approach — one doc per phase
- Part 2 — Verify-before-advancing rule
- Part 2 — Session constraints and summary patterns

FREE RESOURCE · devclarity.dev

INTRODUCTION

Why most AI-assisted builds go wrong

Most developers approach AI like a search engine — type something vague, hope for something useful, spend an hour in revision cycles. The problem isn't the model. It's the missing structure in both the prompt and the build process.

This guide is split into two parts. Part 1 covers prompt-level habits — how to write a single prompt that gets a precise, useful response. Part 2 covers project-level strategy — how to structure an entire build across multiple sessions so nothing gets lost, context stays clean, and each stage is verified before the next begins.

The core principle

AI works best when it knows what you're building, what the rules are, and what the output should look like — before it writes a single line of code. At the project level, that means writing the spec before opening a coding session, not during it.

These patterns come from real production builds across multiple SaaS products — not theory. Apply both parts and you'll spend significantly less time in revision cycles and debugging sessions caused by AI misunderstanding the build context.

PART 1 · PROMPT-LEVEL HABITS

The 6-Step Workflow

Follow these steps in order for every AI coding session. Each step removes a common source of wasted output.

1 State context first — every session

Before any task, tell the AI what project you're working on, the current phase, the tech stack, and what was done in the last session. AI has no memory between sessions — without this, it will make silent assumptions.

```
This is a Next.js SaaS app called My App. We're in the user dashboard phase,

PostgreSQL database, REST API backend. Last session we completed the auth flow.

Today's task is X.
```

2 Use numbered requirements — not prose

Prose requirements get interpreted loosely. A numbered list forces the AI to address each item explicitly and makes it easier to spot what was missed.

```
Build the user profile page with these requirements:

1. Display name, email, avatar

2. Edit name inline — no separate edit page

3. Avatar upload to S3, max 2MB

4. Show error state if upload fails

5. All form validation client-side with Zod
```

3 Ask for a feasibility check before building

Before generating code, ask the AI to flag any risks, missing information, or technical concerns. This surfaces problems in seconds rather than after 100 lines of code.

```
Before building, flag any issues with this approach. Are there any missing

requirements, technical risks, or decisions I need to make first?

List them and wait for my input before proceeding.
```

The 6-Step Workflow (steps 4–6)

4 Request analysis before code

Ask the AI to explain its plan before writing any code. This lets you catch misunderstandings in the approach before they're baked into the implementation.

```
Before writing any code, explain your approach in plain English.

What files will be created or modified? What is the overall structure?

Wait for my confirmation before proceeding.
```

5 Specify the output format explicitly

Tell the AI exactly what you want back — complete files, diffs only, a single function, or a code block with explanation. Left unspecified, you may get partial output that creates more questions than it answers.

```
Output the complete file — no truncation, no "rest of file unchanged" shortcuts.

If the file is over 200 lines, output it in numbered sections and wait for

me to confirm each before continuing.
```

6 Ask for an honest assessment at the end

After the task is complete, ask the AI to evaluate its own output. This surfaces known limitations, edge cases, and things it would do differently — before you push to production.

```
Now that this is built: what are the weaknesses in this implementation?

What edge cases are not handled? What would you do differently if we were

building for production at scale?
```

PART 1 · ANTI-PATTERNS

What Not To Do

These are the eight habits that consistently produce vague, incorrect, or incomplete AI output. Recognise them before they cost you time.

- ✕ **Starting without context**
  Opening a session with a task and no project background forces the AI to guess the stack, the phase, and the constraints. The output will be generic at best, wrong at worst. Always state context first.
- ✕ **Asking for everything at once**
  A prompt that asks for a full feature — UI, API, database, validation, error states, tests — in one go produces shallow coverage of everything and deep coverage of nothing. Break it into steps.
- ✕ **Accepting the first output without review**
  AI output is a first draft. It is frequently missing error handling, incomplete on edge cases, or not quite matching your architecture. Always review before integrating.
- ✕ **Vague requirements**
  "Make it look nice" and "handle errors properly" are not requirements. The AI will interpret these charitably and produce something plausible but not what you meant. Be specific.
- ✕ **Skipping the feasibility check**
  Asking the AI to build something without a prior feasibility check means discovering architectural problems after the code is written. Fifteen seconds of analysis saves fifteen minutes of rewriting.
- ✕ **Not specifying the output format**
  Without explicit format instructions you may get a partial file, a diff, a prose explanation, or a code block — whichever the AI decides is appropriate. Specify what you want.
- ✕ **Letting scope expand mid-session**
  Adding requirements partway through a task causes the AI to retrofit new requirements onto existing output, often breaking earlier decisions. Finish the current task cleanly before adding scope.
- ✕ **Assuming the AI remembers previous sessions**
  It does not. Every session starts with no knowledge of prior work. Without a context statement at the start, you are building on an empty foundation every time.

PART 1 · REFERENCE

Copy-Paste Prompt Template

Use this template at the start of every AI coding session. Fill in the bracketed sections and remove the ones that don't apply.

```
## CONTEXT

Project: [Name and one-sentence description]

Stack: [Framework, database, hosting, key libraries]

Current phase: [e.g. auth, dashboard, payments]

Last session: [What was completed]

Today's task: [Specific feature or fix]

## REQUIREMENTS

1. [Requirement one]

2. [Requirement two]

3. [Requirement three — continue as needed]

## CONSTRAINTS

- [e.g. No raw SQL — use Prisma only]

- [e.g. All routes must be authenticated]

- [e.g. TypeScript strict mode — no implicit any]

## BEFORE YOU BUILD

1. Flag any missing requirements or risks

2. Explain your approach in plain English

3. List the files you will create or modify

4. Wait for my confirmation before writing any code

## OUTPUT FORMAT

[e.g. Complete files only — no truncation]

[e.g. One file at a time — wait for confirmation between files]

## AFTER COMPLETION

List: known weaknesses, unhandled edge cases, and what you

would do differently for production.
```

Tip: Save this as a snippet in your editor or a note. The goal is to make using this structure zero-friction — paste, fill, send.

PART 2 · PROJECT STRATEGY

Project-Level Build Strategy

Part 1 covers how to write a single good prompt. Part 2 covers something different and equally important — how to structure an entire build across multiple sessions, documents, and stages.

The problem with a single large spec document is that AI coding agents read the whole thing and start making assumptions about later stages while building earlier ones. Context gets cluttered. Mistakes from Stage 1 get carried into Stage 3 before they're caught. The build becomes harder to verify and harder to course-correct.

The core insight

Split your build into stages. Give the AI only the document it needs for this stage. Verify the output. Sign off. Then move to the next stage with a fresh focused document. Each stage is clean, reviewable, and correctable before the next begins.

BEFORE — One big spec document
AI reads entire spec →
Makes assumptions about later stages →
Bleeds Stage 3 logic into Stage 1 →
Context gets cluttered over sessions →
Mistakes discovered late →
Hard to verify cleanly stage by stage

AFTER — One document per stage
Stage 1 doc → build → verify → sign off
Stage 2 doc → build → verify → sign off
Stage 3 doc → build → verify → sign off
... → AI only knows what it needs now →
Mistakes caught early, cheaply

PART 2 · STEP 1

Write the Master Spec Before Any Code

Before opening a coding session, write a master spec document. This is not a technical design doc — it is a structured brief that gives the AI everything it needs to understand the build and nothing it doesn't need yet.

Section	Contents	Purpose
1. Project Overview	One paragraph: what it does, who uses it, core value prop	Gives the AI a mental model of the whole before any detail
2. Tech Stack	Framework, database, ORM, auth, hosting, key libraries — with rationale	Prevents the AI from suggesting incompatible or off-brand solutions
3. Core Features	Feature list for the MVP — grouped logically, not by database table	Defines scope so the AI doesn't over-build
4. Data Model Outline	Key entities and their relationships — not a full schema yet	Gives the AI enough to generate a correct Prisma schema in Stage 2
5. Out of Scope	Explicit list of what is NOT in the MVP	This is the most skipped section and the most important one
6. Stage Order	Ordered list of build stages — scaffold, schema, auth, features	Defines the sequence so each document is built in the right order
7. Key Constraints	Rules the AI must follow throughout the build	Repeated at the start of every session to prevent regression

Why the Out of Scope section is critical

Without an explicit out-of-scope list, AI agents will make reasonable guesses about what should be included and build more than you asked for. Payments, i18n, mobile support, admin dashboards — these each add days of scope. Name them explicitly as excluded.

PART 2 · STEP 2

One Document Per Stage

Once the master spec is written, break the build into focused stage documents. Each document is scoped to a single stage. The AI receives only that document for that session — nothing more.

01 **Foundation — Project Scaffold**

Sets up the project structure, dependencies, and configuration. No features. No database. No auth. Just the working skeleton.

```
Using the tech stack in this document, scaffold a Next.js 15 project with:

- App Router folder structure

- Tailwind CSS configured

- Prisma with PostgreSQL

- NextAuth.js v5

Do not build any features yet. Foundation only.

When complete: summarise what was created and confirm the next stage.
```

02 **Database Schema**

Generates the full Prisma schema from the data model outline in the master spec. Verify the schema compiles and migrations run before proceeding.

```
Using the data model in the master spec, create the full Prisma schema.

Include all relations, enums for status and role fields, and sensible indexes.

Run npx prisma generate and confirm it compiles clean.

When complete: summarise the schema and confirm the next stage.
```

03 **Authentication**

Implements auth only. No feature routes, no dashboard. Verify login and session work end to end before any feature work begins.

```
Implement NextAuth.js v5 with:

- Google OAuth provider

- Email/password credentials provider

- Session stored in PostgreSQL via Prisma adapter

Protect a test route to confirm session works.

When complete: summarise what was built and confirm the next stage.
```

04 Features — One at a Time

Each feature gets its own focused session. Define the feature requirements in a short document, build, verify, then move to the next. Never build two features in a single session.

```
Build [Feature Name] only, as described in this document.

Requirements:

1. [Requirement]

2. [Requirement]

Do not modify any other files.

When complete: summarise what was built, list any edge cases not

handled, and confirm the next stage.
```

Verify Before Advancing + Session Constraints

Step 3 — The Verify-Before-Advancing Rule

Do not start a new stage until the current stage passes a basic verification check. This sounds obvious but is consistently skipped under time pressure — and is the single most common source of compounding build errors.

Stage	Minimum Verification Before Advancing
Foundation	App runs locally. No console errors. Folder structure matches spec.
Schema	prisma generate runs clean. Migration applies without errors. Inspect tables in DB.
Auth	Can log in with test account. Session persists on page refresh. Protected route blocks unauthenticated access.
Each Feature	Core happy path works end to end. At least one error state tested. No TypeScript errors.

Step 4 — Session Constraints Pattern

AI has no memory between sessions. Without a constraints block at the start of every session, architectural decisions made in Session 1 may be silently reversed in Session 5. Paste your constraints at the top of every prompt.

```
## PROJECT CONSTRAINTS — include at the start of every session

- Use Prisma for all database access — no raw SQL

- TypeScript strict mode — no implicit any, explicit return types on all functions

- All API routes must be authenticated — no unprotected endpoints

- tsc --noEmit must pass clean before presenting any code

- Use Prisma singleton pattern (lib/prisma.ts) — never instantiate PrismaClient directly

- Enforce data scoping — users may only access their own records

- Follow the folder structure in the spec exactly — do not invent new directories

- Real-time features use [library] — not a custom WebSocket server

- [Payments / file storage / email] are out of scope for this stage
```

Tip: Keep your constraints block in a text snippet. Paste it at the top of the context statement at the start of every session. It takes ten seconds and prevents hours of regression.

REFERENCE

Quick Reference — Both Parts

Part 1 — Prompt Habits

Do	Don't
State project context at the start of every session	Open with a task and no background
Use numbered requirements	Write requirements as prose
Ask for feasibility check before building	Skip the analysis step
Request plan explanation before code	Let the AI decide its own approach
Specify output format explicitly	Accept whatever format appears
Ask for honest assessment after completion	Assume the first output is production-ready
Finish the current task before adding scope	Add requirements mid-session
Repeat constraints at every new session	Assume the AI remembers previous decisions

Part 2 — Project Strategy Checklist

#	Action	When
1	Write master spec document	Before any code
2	Define explicit out-of-scope list	In the master spec
3	Define stage order	In the master spec
4	Create a constraints block	Once, reuse every session
5	Build Stage 1 — scaffold only	First session
6	Verify Stage 1 before advancing	After each stage
7	Create focused document for next stage	Before each session
8	Paste constraints at start of every session	Every session
9	Ask for end-of-stage summary	End of every stage
10	Verify again before advancing to next stage	Before each new stage

CLOSING

The Two-Minute Version

If you take one thing from Part 1

Before every AI coding session: state context, number your requirements, ask for feasibility check, ask for a plan, specify output format, ask for honest assessment. That sequence alone will cut your revision cycles in half.

If you take one thing from Part 2

Write the spec before you open a coding session. Break the build into stages. Give the AI only the document for this stage. Verify before advancing. Repeat the constraints at the start of every session. Mistakes caught at Stage 1 cost minutes. Mistakes caught at Stage 5 cost days.

This is a free resource from devclarity.dev — built by a developer, for developers. If this guide helped you ship something better, that's the point.