How I use Claude Code as a solo dev

Not an AI guru's guide. A small, repeatable system for building real apps with nobody checking your work.

I'm not going to tell you AI will replace developers, or that you can ship a startup in a weekend with one magic prompt. This is the workflow I actually use as a **solo dev building mobile apps**, where there is no team, no code reviewer, and nobody to catch my mistakes but me.

It's boring on purpose. Boring is what keeps an app shippable for months instead of falling apart after week two.

THE WHOLE SYSTEM RESTS ON ONE IDEA

Claude should never have to re-learn my project from scratch. It should read a small map, find the right document, and only then touch code.

PART	WHAT IT COVERS
01 · The problem	Why dumping context into a session backfires
02 · Documentation	A three-layer map: CLAUDE.md, area indexes, topic docs
03 · The rules	Docs first, keep them fresh, test everything
04–05 · Plan	Kickoff, clarifying questions, plan review
06–07 · Prove it	Tests, then checking it yourself
08–09 · Debug, close	Root-cause debugging and keeping docs alive
10–13 · Reference	Worked example, mistakes, cheat sheet, templates

01 · THE PROBLEM

Context dumping

Four things go wrong when you paste the codebase in so Claude "has context".

Burns your context window

The feature touches three files, but you've loaded thirty.

Quality drops

The more irrelevant material Claude wades through, the likelier it latches onto the wrong pattern or ignores the right one.

Slow and expensive

Every session starts with minutes of setup before any real work happens.

Inconsistent

Each session you paste something slightly different, so Claude behaves slightly differently.

WHAT MOST PEOPLE DO

Open a new session. Claude knows nothing about the app, so paste in half the codebase "so it has context". Then ask for one small feature.

THE FIX

The fix isn't a bigger window. It's a better map.

02 · THE DOCUMENTATION SYSTEM

Three layers, read in order

Claude reads the layers top to bottom and only goes deeper when it needs to.

LAYER 1
CLAUDE.md
The map. Always loaded, always short.

→

LAYER 2
Area index
Table of contents for one area.

→

LAYER 3
Topic doc
The detail on one topic.

→

LAST
Code
Only when the docs aren't enough.

HOW CLAUDE READS IT

```
Layer 1: CLAUDE.md                the map (always loaded, always short)
Layer 2: docs/<area>/index.md     the table of contents for one area
Layer 3: docs/<area>/<topic>.md   the detail on one topic
         (code)                   only when the docs aren't enough
```

Why this works

1. **A tiny, consistent map.** Claude starts every session with the same small file.
2. **Only relevant detail.** It pulls in the docs that match the task, nothing else.
3. **Useful to you too.** Come back to an area after two months and read the same doc Claude does.
4. **Compressed architecture.** The docs are a human-readable version of your design, much cheaper to read than the code they describe.

02 · LAYER 1

A lean CLAUDE.md

CLAUDE.md is the one file Claude Code loads at the start of every session, which makes it the most valuable and most dangerous file in your repo. Every line costs context, every session, forever. It answers four questions and nothing else: how is the code structured, where do I find things, what conventions do we follow, and what would bite you if you didn't know.

THE PRUNING TEST

"Would Claude get something wrong in most sessions without this line?" If not, it belongs in a topic doc.

EXAMPLE · CLAUDE.MD FOR AN IOS APP (SWAP IN YOUR STACK)

```
# StreakApp

iOS habit tracker. SwiftUI + SwiftData locally, Supabase for auth and sync.
Solo project. Keep changes small and reviewable.

## Project map
- `App/`            App entry, dependency wiring
- `Features/`       One folder per feature (Streaks, Onboarding, Paywall)
- `Core/`           Shared services (Networking, Notifications, Storage)
- `DesignSystem/`   Colors, typography, reusable components
- `Tests/`          Unit + integration tests, mirrors `Features/`
- `docs/`           All project documentation (see below)

## Conventions
- MVVM. Views are dumb; logic lives in `@Observable` view models.
- Every network call goes through `Core/Networking/APIClient`.
- No hardcoded colors or fonts. Use `DesignSystem` tokens.
- Errors surface through `AppError`. Never swallow errors silently.

## Gotchas
- Ask for notification permission in context, never on launch.
- Changing a SwiftData model requires a migration in
  `Core/Storage/Migrations.swift`.

## Documentation rules (IMPORTANT)
1. Do NOT read source code to get context. Start with `docs/`: read
   `docs/index.md`, then the relevant area index, then the topic doc.
2. Only read code when the docs don't answer the question, or when you
   are about to edit that specific file.
3. If documentation is missing, outdated or wrong, update it. No doc for
   the topic? Create one and add it to the area's index.
4. Before finishing, check: did I change behavior the docs describe?
   If yes, update the docs in the same change.

## Testing rules
- Every major feature ships with unit tests AND one integration test.
- Run the full suite before saying a task is done, and report the result.

## Workflow
- Ask clarifying questions before writing a plan if anything is ambiguous.
- Present a plan and wait for my approval before writing code.
```

02 · LAYER 2

An index per area

Each major part of the app gets a folder. Inside: one markdown file per topic, plus an `index.md` whose only job is to tell Claude **which document to read for which question**.

REFERENCE · DOCS FOLDER STRUCTURE

```
docs/
├── index.md                  ← top-level index (lists each area)
├── architecture/
│   ├── index.md
│   ├── overview.md
│   ├── navigation.md
│   └── dependency-injection.md
├── backend/
│   ├── index.md
│   ├── database-schema.md
│   ├── auth-flow.md
│   ├── sync-strategy.md
│   └── row-level-security.md
├── features/
│   ├── index.md
│   ├── streaks.md
│   ├── reminders.md
│   └── paywall.md
├── design-system/   (index, colors-and-type, components)
└── testing/         (index, conventions, test-data)
```

COPY-PASTE · DOCS/BACKEND/INDEX.MD

```
# Backend docs

Read the doc that matches your task. Do not read them all.

| If you are working on...                        | Read                    |
|-------------------------------------------------|-------------------------|
| Adding/changing a table or column               | `database-schema.md`    |
| Sign-in, sign-up, sessions, tokens              | `auth-flow.md`          |
| Offline behavior, conflict resolution, syncing | `sync-strategy.md`      |
| Who can read/write which rows                   | `row-level-security.md` |
```

02 · LAYER 3

A topic doc, small

What a good topic doc looks like: where it lives, how it works, what will bite you.

COPY-PASTE · DOCS/FEATURES/REMINDERS.MD

```
# Reminders

## What it does
Local notifications that nudge the user to complete a habit before their
streak breaks. Scheduled on-device. No server push involved.

## Key files
- `Features/Reminders/ReminderScheduler.swift`: schedules and cancels
- `Features/Reminders/ReminderSettingsView.swift`: UI
- `Core/Notifications/NotificationService.swift`: permission + delivery

## How it works
1. User picks a time per habit in Settings.
2. `ReminderScheduler` creates one repeating UNCalendarNotificationTrigger.
3. On completion for the day, today's pending notification is cancelled
   and tomorrow's is left in place.

## Rules and gotchas
- iOS caps pending local notifications at 64. The scheduler prioritises
  habits with the longest active streaks.
- Permission is requested the first time the user enables a reminder.

## Tests
- `RemindersTests/ReminderSchedulerTests.swift` (unit)
- `RemindersTests/ReminderFlowIntegrationTests.swift` (integration)

## Last updated
After adding the streak-at-risk reminder.
```

a **Small and specific.** One topic, one file.

b **It says where the code lives,** so Claude goes straight to the right file instead of searching.

c **Gotchas are written down,** including the 64-notification cap that later shapes a plan.

03 · THE RULES

Rules that make it work

The structure only helps if Claude actually uses it. Three rules in CLAUDE.md make sure it does.

Rule 1 · Docs first

"Do not read my code to get context. Check the relevant documentation first. Only read code when it is absolutely necessary."

Rule 2 · Keep docs updated

"If documentation is outdated, update it. If there's none for the topic, create it and add it to the index."

Claude follows Rule 1 well but not perfectly. Treat it as a strong default, not a guarantee. If you see it wandering through folders, remind it: "*Check the docs first. Which doc covers this?*"

"Necessary" means the docs don't answer the question, or Claude is about to edit a file and needs its current contents. The goal is to stop **exploratory** reading, not **targeted** reading.

Rule 2 is what makes the system self-maintaining. Without it, docs rot within a month and then actively mislead Claude, which is worse than no docs at all. **Rule 3** is tests for every major feature (page 10).

What to keep out of CLAUDE.md

INFORMATION	WHERE IT GOES
Applies to nearly every task	`CLAUDE.md`
Applies to one area of the app	`docs/<area>/<topic>.md`
Explains *why* a decision was made	The topic doc for that area
A one-off instruction for this task	Your prompt, not any file
Already obvious from reading code	Nowhere

04 · STARTING A FEATURE

Context, brief, questions

Two minutes at the start of every feature saves hours.

STEP 1
Give the map
CLAUDE.md plus only the relevant docs. No pasted code.

→

STEP 2
Explain it
Goal, audience, behavior, and what's out of scope.

→

STEP 3
Ask first
Questions before any plan.

Step 3 is the most underrated. Most bad output comes from Claude silently filling gaps with guesses. Describe the feature like you would to a developer you trust but who is new to the project.

COPY-PASTE · OPENING PROMPT

```
Read CLAUDE.md, then docs/features/index.md, docs/features/reminders.md,
and docs/backend/sync-strategy.md.

I want to add a "streak at risk" reminder. If a user has a streak of 3+
days and hasn't completed the habit by 7pm, send a local notification
telling them their streak is about to break.

Out of scope: server push notifications, and any changes to the paywall.

Before you write a plan, ask me any clarifying questions you have. Don't
assume answers to anything that would change the design.
```

WHAT A GOOD RESPONSE LOOKS LIKE

```
A few questions before I plan this:

1. Should 7pm be fixed, or should the user be able to change it?
2. If the user completes the habit at 6:55pm, should the notification
   be cancelled? (I'm assuming yes.)
3. Does this apply to every habit, or can users opt out per habit?
4. With multiple at-risk habits: one notification or one per habit?
5. The reminders doc mentions the 64 pending-notification cap. How
   should this new type be prioritised against daily reminders?
```

Question 5 came straight from the doc, something you'd never think to specify. That's the system paying off. Short answers are fine; *then* Claude plans.

05 · REVIEWING THE PLAN

Three ways to review

I use all three depending on the feature. Self-review catches a surprising number of issues: "What could break? What did you miss? Does anything conflict with the docs?"

OPTION 1
Edit it
Change a step, cut scope, add a constraint. Send it back.

→

OPTION 2
Self-review
Ask Claude what could break or conflicts with the docs.

→

OPTION 3
Approve
If it's right, say so and Claude implements.

What I look for in a plan

1. **Only the files it should.** A plan that modifies twenty files for a small feature is a red flag.
2. **Your conventions.** Does it follow CLAUDE.md, or invent its own pattern?
3. **Tests.** Are they in the plan?
4. **Doc updates.** Does it mention them?
5. **Anything you don't understand.** Ask before approving. You're the only reviewer this code will ever get.

COPY-PASTE · TIGHTENING A PLAN

```
The plan looks good except:
- Drop step 4 (refactoring NotificationService). That's a separate task.
- In step 2, don't create a new settings screen. Add the toggle to the
  existing ReminderSettingsView.
- Add a test for the 64-notification cap case.

Update the plan and I'll approve it.
```

TIP · PLAN MODE

Toggle plan mode with `Shift+Tab`. It stops Claude editing files while you're still agreeing on the approach: a nice safety net for this step.

06 · TESTS

Tests are not optional

As a solo dev I have no QA team. The tests are my QA team, and Claude is very good at writing them, so there's no excuse. It goes in CLAUDE.md as a **rule**, not a suggestion.

COPY-PASTE · CLAUDE.MD TESTING RULES

```
## Testing rules
- Every major feature ships with unit tests AND integration tests.
- Run the full test suite before saying a task is done, and report results.
- If a test fails, do not delete or weaken it to make it pass. Fix the
  code, or tell me why the test is wrong.
```

That last line matters. Without it, a failing test can get "fixed" by quietly changing the assertion.

Unit tests

One piece in isolation: given a 5-day streak and no completion, does the scheduler return a 7pm notification? Catch logic bugs.

Integration tests

Pieces together: complete the habit at 6:55pm; is the pending notification actually cancelled? Catch the bugs between components.

EXAMPLE · TESTS CLAUDE MIGHT WRITE (ADAPT TO XCTEST, JEST, PYTEST)

```
@Test func streakAtRiskNotificationIsScheduledForActiveStreaks() async throws {
    let scheduler = ReminderScheduler(center: MockNotificationCenter())
    let habit = Habit.fixture(currentStreak: 5, completedToday: false)

    try await scheduler.scheduleStreakAtRisk(for: habit, at: .hour(19))

    #expect(scheduler.center.pending.count == 1)
}

@Test func completingHabitCancelsStreakAtRiskNotification() async throws {
    let env = TestEnvironment()   // integration: real scheduler + real store
    let habit = try await env.createHabit(streak: 5)
    try await env.scheduler.scheduleStreakAtRisk(for: habit, at: .hour(19))

    try await env.complete(habit)

    #expect(env.center.pending.isEmpty)
}
```

07 · CHECK IT YOURSELF

Green tests are not done

Green tests mean the feature does what the tests say. That isn't the same as working. After the tests pass, **run it and use it** like a user would: simulator for an iOS app, a real device for anything hardware-related such as notifications, camera or haptics.

CHECK	LOOK FOR
Happy path	Does it do what I asked?
Edge cases	Empty states, first launch, no network, very long text, rapid taps
Feel	Does it look and animate right? Tests can't tell you this
Regression	Did anything next to it break?
Light / dark, small / large	A quick pass is enough

This is where I find things I didn't know I wanted: "*Oh, this should also show up on the home screen*" only happens when I'm looking at the real thing.

Simulator

Fast loop for layout, flows, edge cases and light/dark passes. Where most of the checking happens.

Real device

Anything hardware-related: notifications, camera, haptics. The simulator can't vouch for these.

WHY IT MATTERS

Green tests prove what they test. Only using the app tells you whether it is right.

08 · DEBUGGING

Paste, explain, investigate

I don't guess the fix and tell Claude what to change. I give it evidence: **the error or symptom** pasted exactly, **what I did** to get there, and the instruction to **investigate, find the root cause, then fix**.

COPY-PASTE · BUG REPORT EXAMPLE

```
The app crashes when I complete a habit right after turning on airplane mode.

Here's the error from the Xcode console:

Fatal error: Unexpectedly found nil while unwrapping an Optional value
  at SyncQueue.swift:87

What I did:
1. Opened the app while online, signed in
2. Turned on airplane mode
3. Tapped "Complete" on a habit that had a 5-day streak
4. Crash

Don't just patch line 87. Investigate why this is nil, find the root
cause, explain it to me, and then fix it. Add a test that reproduces
this bug first.
```

a **"Root cause" is the magic phrase.** It stops Claude slapping a `?` or nil-check on the crash line and calling it done.

b **A reproducing test first** means the bug can never silently come back.

c **Explaining it to me** is how I avoid the same class of mistake again.

If the first fix doesn't work, paste the **new** error and say what changed. Don't just say "still broken": the new error is new information.

09 · CLOSING THE LOOP

Docs stay alive

At the end of every feature, before closing the session, check the docs reflect what was built.

COPY-PASTE · CLOSING PROMPT

```
Before we wrap up:
1. Did anything we changed affect what the docs describe? Update them.
2. Is there anything new that deserves its own doc? Create it and add it
   to the relevant index.md.
3. Show me a summary of which docs you touched.
```

Because of the rule in CLAUDE.md, Claude usually does this unprompted, but I verify. Then I skim the docs diff like I do the code. It takes one minute. It's the difference between a system that works for a month and one that works for years: every feature leaves the map a bit more accurate.

10 · FULL WORKED EXAMPLE

One feature, start to finish

The "streak at risk" notification, end to end.

#	WHAT I DO	TIME
1	Open a fresh session. Point Claude at CLAUDE.md + 2 relevant docs.	1 min
2	Describe the feature and what's out of scope. Ask for clarifying questions first.	2 min
3	Answer Claude's five questions.	2 min
4	Read the plan. Cut one step, add one constraint. Ask Claude to self-review it.	5 min
5	Approve. Claude implements, writes unit + integration tests, runs the suite.	Claude works
6	Build to simulator and device. Test the happy path and three edge cases.	10 min
7	Find a bug: notification fires even after completing at 6:55pm. Paste the repro steps.	2 min
8	Claude finds the root cause (cancel runs before the store save finishes), fixes it, adds a regression test.	Claude works
9	Re-test locally. Works.	3 min
10	Closing prompt: update docs. Skim the diff. Commit.	3 min

Notice what never happened: I never pasted a wall of code, and Claude never went on a file-by-file exploration trip.

11 · MISTAKES

Mistakes so you don't have to

Eight things I got wrong first.

1. **Letting CLAUDE.md grow.** It crept from 40 lines to 300 and Claude started ignoring parts of it. Prune regularly; area-specific sections move to that area's docs.
2. **Writing docs for humans who don't exist.** Short, specific, factual. Where is it, how does it work, what will bite you.
3. **Skipping clarifying questions to save time.** Every time, I redid the work. The two minutes are always worth it.
4. **Approving a plan I didn't fully read.** If you don't understand the plan you won't understand the code, and you'll be debugging it at midnight.
5. **Trusting green tests.** They only prove what they test. Always check it yourself.
6. **Fixing bugs by telling Claude the fix.** My guess at the cause was wrong about half the time. Give symptoms and evidence, not a diagnosis.
7. **Letting docs go stale.** A wrong doc is worse than no doc. The update rule is non-negotiable.
8. **One giant session for everything.** Long sessions get noisy. When a feature is done, start fresh. The docs carry the memory, not the chat.

12 · CHEAT SHEET

Two checklists

Setup once, then run the per-feature loop every time.

Setup (once per project)

- ☐ Lean `CLAUDE.md`: structure, locations, conventions, gotchas
- ☐ `docs/` folder with an `index.md` per area
- ☐ Rule: docs first, code only when necessary
- ☐ Rule: update or create docs as you go
- ☐ Rule: tests for every major feature

Per feature

- ☐ Fresh session
- ☐ Give CLAUDE.md + relevant docs only
- ☐ Describe the feature and what's out of scope
- ☐ Ask Claude to ask clarifying questions first
- ☐ Review the plan: edit, have Claude review it, or approve
- ☐ Claude implements + writes and runs tests
- ☐ Manually test locally
- ☐ Paste errors with context; ask for root cause
- ☐ Update docs, skim the diff, commit

13 · TEMPLATES

Four prompts to copy

COPY-PASTE · FEATURE KICKOFF

```
Read CLAUDE.md, then the docs relevant to this task:
- [doc 1]
- [doc 2]

I want to build: [what it does, in plain language]
Who it's for / why: [context]
Out of scope: [what not to touch]

Before you write a plan, ask me any clarifying questions. Don't assume
answers to anything that would change the design.
```

COPY-PASTE · PLAN REVIEW

```
Review your own plan before I approve it. What could break? What did you
assume? Does it conflict with anything in the docs? Is there a simpler way?
```

COPY-PASTE · BUG REPORT

```
Problem: [what's wrong vs. what I expected]

Error / output:
[paste exactly]

What I did:
1. ...
2. ...

Investigate and find the root cause before changing anything. Explain it to
me, add a test that reproduces it, then fix it.
```

COPY-PASTE · WRAP-UP

```
Before we wrap up: did anything we changed affect what the docs describe?
Update them. Anything new deserving its own doc? Create it and add it to the
index. Summarise which docs you touched.
```

FINAL THOUGHT

None of this is clever

Good luck, and happy shipping.

COPY-PASTE · STARTER DOCS/INDEX.MD

```
# Documentation index

Read the index for your area first. Do not read every doc.

| Area          | Covers                                | Index                     |
|---------------|---------------------------------------|---------------------------|
| Architecture  | App structure, navigation, DI         | `architecture/index.md`   |
| Backend       | Database, auth, sync, security rules  | `backend/index.md`        |
| Features      | How each feature works                | `features/index.md`       |
| Design system | Colors, type, components              | `design-system/index.md`  |
| Testing       | Conventions, test data, how to run    | `testing/index.md`        |
```

The whole system

1. **Give Claude a small, accurate map.**
2. **Make it think before it builds.**
3. **Make it prove it works.**
4. **Check it yourself.**
5. **Leave the map better than you found it.**

START HERE

None of this is clever. There's no secret prompt. Start with just the lean CLAUDE.md and one docs folder. Add more only when you feel the pain.