Code Surfing

Build Apps in a Day with AI

Onde on the Tech

Introduction: What is Code Surfing?

There's a moment every developer knows. You're staring at a blank file, cursor blinking, and you need to build something. Maybe it's a simple script. Maybe it's a whole application. Either way, you're about to type a lot of code.

Or are you?

The Wave Has Changed

In 2024, something shifted. AI coding assistants went from "interesting toy" to "I can't imagine working without this." By 2025, the best developers weren't the ones who typed the fastest—they were the ones who could *direct* AI to build what they imagined.

We call this **Code Surfing**.

Like a surfer riding a wave, you're not fighting the water. You're working *with* it. The AI is the wave—powerful, fast, sometimes unpredictable. Your job is to stay balanced, point in the right direction, and enjoy the ride.

What This Book Is (And Isn't)

This is not a book about prompting tricks. It's not a collection of ChatGPT hacks. It's not theory.

This is a practical guide based on real projects. Every technique in this book comes from actual work:

A Telegram bot built in an afternoon

A children's book publishing pipeline

- A dashboard that went from idea to deployed in hours
- Multi-agent workflows that coordinate like a small team

I didn't read about these techniques. I *did* them. And now I'm going to show you how.

Who Should Read This

Developers who want to 10x their output

Founders who want to build MVPs without a team

- **Creators** who have ideas but think "I can't code"
- **Anyone** curious about how AI is changing software

If you can describe what you want in plain English, you can code surf.

The Tools

Throughout this book, we'll use:

Claude (via Claude Code CLI) - our primary AI assistant

Grok - for image generation and alternative perspectives

- **Browser automation** - for when APIs don't exist
- **Standard tools** - Node.js, Python, Git, the usual suspects

You don't need all of these to start. You just need one AI and an idea.

How to Read This Book

Each chapter builds on the last, but you can skip around. If you're impatient (good—that's the code surfing spirit), jump to Chapter 3 and build something immediately. Come back to the fundamentals when you get stuck.

The best way to learn is to do. So open your terminal, fire up your AI assistant, and let's catch some waves.

"The best code is the code you didn't have to write."

Next: Chapter 1: Setting Up Your Surfboard →

Chapter 1: Setting Up Your Surfboard

Before you catch your first wave, you need the right equipment. In code surfing, your "surfboard" is your development environment—the AI assistant, the terminal, and the invisible infrastructure that makes everything flow.

Choosing Your AI Assistant

Not all AI assistants are created equal. Here's what matters:

ToolBest ForLimitations

Claude Code Terminal-first development, file editing, GitRequires CLI comfort **Cursor**IDE integration, visual editingSubscription model **GitHub Copilot**Autocomplete, inline suggestionsLess conversational **ChatGPT**Quick questions, brainstormingNo file access **Grok**Image generation, X integrationLimited coding focus

My recommendation: Start with Claude Code if you're comfortable in a terminal. It can read your files, edit them, run commands, and maintain context across your entire project.

The Conversation as Your Dev Log

Here's a mindset shift: your conversation with the AI *is* your development log.

Traditional development:

Think about what to build

Write code

Test

Debug

- Repeat

Code surfing:

Describe what you want

AI writes code

You review and refine

AI adjusts

- Ship

The conversation captures your intent, the iterations, the decisions. It's documentation that writes itself.

File Structure That AI Understands

AI works better when your project is organized predictably:

```
project/
├── README.md          # AI reads this first
├── CLAUDE.md          # Special instructions for Claude
├── src/               # Source code
├── tests/             # Test files
├── docs/              # Documentation
└── package.json       # Dependencies
```

Pro tip: Create a `CLAUDE.md` file with project-specific instructions. The AI will read it automatically and follow your rules.

Example `CLAUDE.md`:

```
# Project Rules

- Use TypeScript, not JavaScript
- All API calls go through src/api/

- Run tests before committing
- Never commit .env files
```

Environment Variables and Secrets

Never put secrets in your code. AI will see them. Git will track them. This is basic, but it's worth repeating.

```
# .env (gitignored)

API_KEY=your-secret-key
DATABASE_URL=postgres://...

# In code

const apiKey = process.env.API_KEY;
```

The Automation Mindset

Here's where code surfing gets powerful: **automate everything repeatable**.

If you do something twice, ask yourself:

Can I script this?

Can I make the AI do this automatically?

- Can I create a workflow that handles this forever?

Examples from real projects:

PDF generation: One command creates print-ready books

Social media: Bot posts with approval workflow

- **Deployment**: Push to Git, production updates automatically

The goal is to reduce friction to zero. The less you have to think about *how* to do something, the more you can focus on *what* to build.

Levels of Automation

As your projects grow, so should your automation:

Level 1: Manual (Where We Start)

Run commands by hand

Copy files manually

- Deploy by uploading

Level 2: Scripts (Where Most Projects Live)

`npm run build` does everything

One command to deploy

- AI-generated utility scripts

Level 3: Pipelines (Production Ready)

Git push triggers builds

Automated testing

- Staging → Production promotion

Level 4: Full CI/CD (Enterprise Scale)

Multiple environments (dev, staging, production)

Automated rollbacks

- Feature flags
- Monitoring and alerts

Where should you be? Probably Level 2 for most projects. Don't over-engineer. But know that Level 3-4 exists for when you need it.

Pre-Production vs Production

Even for small projects, separating environments saves headaches:

```
Development (your machine)
    ↓
Staging (test server)
    ↓
Production (real users)
```

Minimum viable separation:

Use different API keys for dev/prod

Have a test database

- Don't test on production (you'll break things)

For now, a simple approach:

```
const isProd = process.env.NODE_ENV === 'production';
const apiUrl = isProd
  ? 'https://api.yourapp.com'
  : 'http://localhost:3000';
```

As apps grow, you'll want proper staging servers. But start simple.

Your First Setup Checklist

[ ] AI assistant installed and configured

[ ] Terminal comfortable (or IDE ready)

[ ] Git initialized

[ ] `.env` file created (and gitignored)

- [ ] `CLAUDE.md` with project rules
- [ ] Basic folder structure in place

Don't overthink this. You can always reorganize later. The point is to start surfing, not to build the perfect surfboard.

"Premature optimization is the root of all evil. But no optimization is the root of all slowness."

Next: Chapter 2: The Art of Prompting →

Chapter 2: The Art of Prompting for Code

The difference between a frustrating AI session and a productive one usually comes down to one thing: how you ask.

This isn't about magic words or secret prompts. It's about communication. The AI is smart, but it can't read your mind. Yet.

Be Specific, Get Specific

Bad prompt:

> *"Make me a website"*

Better prompt:

> *"Create a Next.js landing page with a hero section, three feature cards, and a contact form. Use Tailwind CSS. Mobile responsive."*

Best prompt:

> *"Create a Next.js landing page for a children's book publisher. Hero section with book cover image, tagline

> *'Stories that spark imagination'. Three feature cards: 'Illustrated Books', 'Educational Content', 'Multi-language'. Contact form at bottom. Tailwind CSS, mobile-first. Color palette: warm yellows and soft blues."*

The more context you provide, the closer the AI gets to what you actually want.

Context is King

AI doesn't know:

Your project structure (unless you show it)

Your coding preferences (unless you tell it)

- Your business goals (unless you explain them)
- What you tried before (unless you share it)

Always include relevant context:

```
I'm building a Telegram bot for a publishing company.
Current file structure:
src/bot.ts (main bot logic)
src/handlers/ (command handlers)


  • src/api/ (external API calls)
```

```
I need to add a command that lets users approve or reject
posts before they go live. The approval should be stored
in a JSON file for now (we'll add a database later).
```

This prompt tells the AI:

What project you're working on

How it's organized

- What you need
- Future considerations

The Iterative Dance

Code surfing is a conversation, not a monologue.

Round 1: Ask for what you want

```
Create a function that generates a PDF from HTML
```

Round 2: Refine based on result

```
Good, but add support for custom fonts.
Use Puppeteer instead of html-pdf.
```

Round 3: Handle edge cases

```
What if the HTML has images?
```

```
Convert them to base64 first.
```

Round 4: Polish

```
Add error handling and a progress log.
```

Four rounds, complete solution. Each round builds on the last.

When to Start Over

Sometimes the AI goes down the wrong path. Signs it's time to reset:

You've been fixing the same bug for 5+ iterations

The code is getting more complex, not simpler

- You realize your initial request was wrong

Don't sunk-cost yourself. Say:

> *"Let's start fresh. Forget the previous approach. Here's what I actually need: [clearer description]"*

The Magic Phrases

Certain phrases unlock better responses:

For exploration:

"What are my options for..."

"Compare approaches for..."

- "What would you recommend for..."

For precision:

"Show me the exact code for..."
"Step by step, how do I..."

- "Give me a minimal example of..."

For debugging:

"This code gives error [X]. Why?"
"What's wrong with this approach?"

- "How would you fix this?"

For improvement:

"How can I make this more efficient?"
"What am I missing?"

- "Review this code for issues"

Prompt Templates

Save prompts that work. Here are some starters:

New Feature

```
I need to add [FEATURE] to [PROJECT].
Current relevant code: [paste or describe]
Requirements:
- [Requirement 1]
- [Requirement 2]

Constraints:
- [Constraint 1]
- [Constraint 2]
```

Bug Fix

```
Bug: [What's happening]
Expected: [What should happen]
Code: [relevant code]
Error message: [if any]
What I've tried: [previous attempts]
```

Code Review

```
Review this code for:
Security issues
Performance problems

- Code style
- Edge cases
```

```
[paste code]
```

Architecture Decision

```
I need to choose between:
A) [Option A]
B) [Option B]

Context: [project description]
Priorities: [speed? maintainability? scalability?]

Which do you recommend and why?
```

The Meta-Skill

The real skill in prompting is knowing what you don't know.

If you're unsure about something, ask:

> *"What questions should I be asking about [topic]?"*

The AI will help you discover the unknown unknowns.

Common Mistakes

Too vague: "Make it better" → Better: "Make it faster by caching the API responses"

Too much at once: "Build me a complete e-commerce platform" → Better: "Start with a product listing page"

No context: "Fix this bug" → Better: "This function returns null when the input is empty. It should return an empty array."

Ignoring errors: "It doesn't work" → Better: "I get 'TypeError: Cannot read property X of undefined' on line 42"

Practice Exercise

Try this prompt with your AI:

```
I want to build a simple CLI tool that:
Reads a markdown file
Extracts all headers (# and ##)


  • Outputs them as a table of contents

Use Node.js. Keep it under 50 lines.
Show me the code and explain each part.
```

Watch how the AI responds. Then iterate:

"Add line numbers to each header"

"Support ### headers too"

- "Make it work with stdin so I can pipe files"

This is code surfing. Each wave builds on the last.

"A well-asked question is half-answered."

Next: Chapter 3: Your First Project →

Chapter 3: The Real Costs of Code Surfing

Let's talk money. There's a romantic idea that AI makes everything free. It doesn't.

If you want to code surf seriously—building real projects, shipping products, automating workflows—you'll need to invest. This chapter breaks down what things actually cost.

The Honest Truth

Free tiers exist, but they're limited. You'll hit walls:

Rate limits

Context length restrictions

- Missing features
- Slow responses

When you're learning, free is fine. When you're building, you pay.

Monthly Cost Breakdown

Here's what a serious code surfing setup costs in 2026:

Essential (Minimum to Get Started)

ServiceCost/MonthWhat You Get

Claude Pro$20Claude 3.5 Sonnet, longer context GitHub Copilot$10IDE autocomplete **Total$30/month**

This gets you in the door. You can build real things.

Professional (What I Actually Use)

ServiceCost/MonthWhat You Get

Claude Pro$20Main AI assistant Claude Code (API)$50-200Heavy usage, Opus model X Premium$8-16Grok, image generation Cursor Pro$20AI-native IDE Vercel Pro$20Deployment, serverless Domain names$10-20Various projects **Total$130-280/month**

Enterprise (Full Stack)

ServiceCost/MonthWhat You Get

All Professional tier$200+Everything above AWS/GCP$50-500Servers, databases Monitoring (Datadog, etc)$30+Production visibility Error tracking (Sentry)$26+Bug

catching CI/CD (CircleCI, etc)$30+Automated pipelines **Total$350-1000+/month**

The API Usage Reality

Here's where costs can explode: **API usage**.

Claude Code tracks your spending. A heavy day of development might cost:

Simple session: $0.50-2

Complex refactoring: $5-15

- Multi-file project setup: $10-30

Per month, expect $50-200 if you're building actively.

How to Control API Costs

Use Haiku for simple tasks - 10x cheaper than Opus

Be concise - Shorter prompts = lower costs

Don't repeat context - Reference previous work

Batch requests - Do multiple things at once

- **Know when to stop** - Diminishing returns are real

Free Alternatives (With Tradeoffs)

If budget is tight, here's what works:

PaidFree AlternativeTradeoff

Claude ProClaude.ai (free tier)Limited messages/day CopilotCodeiumLess accurate CursorVS Code + ExtensionsMore manual setup Vercel ProVercel HobbyLimited builds GrokIdeogram/DALL-E freeLower quality

My advice: Start free, upgrade when you hit limits. You'll know when it's time.

Hidden Costs

Things people forget:

Time

Learning curve: 20-40 hours to get comfortable

Debugging AI mistakes: Still takes time

- Reviewing generated code: You can't blindly trust

Technical Debt

AI loves over-engineering

Quick solutions become permanent

- Refactoring still costs money

Context Switching

Every new AI tool has its quirks

Prompting skills don't fully transfer

- Updates break workflows

ROI: Is It Worth It?

Let's do the math.

Old way: Hire a developer ($50-150/hour)

- 10-hour project = $500-1500

Code surfing way: Your time + AI costs

- 10-hour project = Your time + $20-50 in API
- (Assuming you're the one building)

If you're a founder: Code surfing lets you build MVPs before hiring **If you're a developer**: Code surfing makes you 3-5x faster **If you're a creator**: Code surfing unlocks things you couldn't do before

The math works out. But only if you actually ship.

My Recommendation

Start with:

Claude Pro ($20/month) - Best bang for buck

X Premium ($8/month) - Grok for images

- **Free tier everything else**

Total: **$28/month**

Upgrade when you:

Hit rate limits regularly
Need faster responses

- Want better models
- Are making money from what you build

The Mindset Shift

Don't think of AI costs as expenses. Think of them as:

Time multipliers - $20 saves 20 hours
Capability unlocks - Do things you couldn't before

- **Learning investments** - Skills that compound

The most expensive option is not building at all.

"If you think hiring a professional is expensive, wait until you hire an amateur." —Red Adair

(The same applies to AI tools.)

Next: Chapter 4: Your First Project →

Chapter 4: The Tools Landscape (January 2026 Edition)

Warning: This chapter has an expiration date.

The AI landscape changes weekly. What I write today might be outdated by the time you read it. That's not a bug—it's the nature of the wave we're surfing.

A Disclaimer You Should Take Seriously

Everything in this chapter reflects **January 2026**. By the time you read this:

New models will exist

Pricing will have changed

- Some tools might be dead
- Better alternatives might have emerged

Don't memorize tools. Learn patterns.

The specific tools don't matter as much as understanding *what each type of tool does* and *how to evaluate new ones*.

The Current Landscape

For Code Generation

ToolBest ForMy Take (Jan 2026)

ClaudeComplex reasoning, long context, file editingMy daily driver. Opus for hard stuff, Sonnet for most things, Haiku for simple tasks **GPT-4**General purpose, pluginsGood but I find Claude better for code **Gemini**Google integration, long docsUseful for specific cases **Copilot**Inline autocompleteNice to have, not essential **Cursor**AI-native IDEGreat if you like IDEs

For Image Generation

ToolBest ForMy Take

GrokFast iterations, good quality, free with X Premium**My go-to for illustrations**. The X integration is convenient. **DALL-E**Reliability, OpenAI ecosystemConsistent but less creative **Midjourney**Artistic qualityBeautiful but slower workflow **Stable Diffusion**Local control, customizationFor power users **Ideogram**Text in imagesNiche but useful

Why Grok for images?

Included with X Premium ($8/month)

No separate credits or limits

- Fast iteration (generate, adjust, regenerate)
- Good enough for book illustrations, social media, prototypes

For Browser Automation

ToolBest ForMy Take

Claude in ChromeAI-controlled browsingGame changer. AI can see and interact with web pages **Playwright**Programmatic controlWhen you need scripts **Puppeteer**Headless ChromePDF generation, screenshots

For Development Environment

ToolBest For

VS Code Free, extensible **Cursor**AI-first experience **Zed**Speed, collaboration **Terminal + Claude Code**Maximum control

How to Evaluate New Tools

When a new AI tool drops (which happens constantly), ask:

- **What does it do that my current tools don't?**

- If the answer is "nothing special" → skip it

- **Does it integrate with my workflow?**

- Standalone tools that require context switching → friction

- **What's the pricing model?**

- Per-token? Subscription? Free with limits?

- **Who's building it?**

- Big company = stability, slow updates - Startup = fast innovation, might disappear

- **What's the community saying?**

- Not hype. Actual usage reports.

Things That Changed Recently

Just in the past 6 months:

Claude got much better at code

Grok added image generation

- Cursor took off
- [This will be outdated by the time you read it]

The point: Stay curious, but don't chase every new thing.

My Current Stack (January 2026)

```
Code: Claude Code (CLI) + VS Code
Images: Grok (via X)
```

```
Browser: Claude in Chrome extension
Deployment: Vercel + GitHub
PDF: Puppeteer
Database: SQLite for small stuff, Postgres for real apps
```

This works for me. Your stack will be different. That's fine.

The Meta-Skill: Learning New Tools Fast

Instead of memorizing tools, learn this process:

Install and run the example (5 minutes)

Try your actual use case (15 minutes)

- **Hit a wall and read docs** (10 minutes)
- **Decide: keep or discard** (2 minutes)

Total: 30 minutes to evaluate any tool.

If it doesn't click in 30 minutes, it's probably not for you (right now).

What Won't Change

Some things are stable:

Git - Version control isn't going anywhere

Terminal - Commands are forever

HTTP - APIs speak HTTP

JSON - Data format of the web

- **JavaScript/Python** - Lingua franca of AI tools

  Learn these well. They'll outlast any specific AI tool.

Future-Proofing Yourself

- **Learn concepts, not just tools**

- "How to prompt" > "How to use Claude specifically"

- **Build transferable skills**

- Debugging is debugging - Architecture is architecture - Good taste is good taste

- **Stay loosely coupled**

- Don't build everything around one AI - Abstract the AI layer when possible

- **Read the changelogs**

- Subscribe to updates from tools you use - New features unlock new possibilities

The Only Constant

Everything changes. That's the wave.

The best surfers aren't the ones who memorize one wave. They're the ones who can read any wave and adapt.

Same with code surfing. The tools are temporary. The skill is forever.

"This too shall pass." (Especially true in AI.)

Next: Chapter 5: Your First Project →

Chapter 5: What You Actually Need to Know

Code surfing doesn't mean you know nothing. There's a foundation of skills and tools that makes everything else possible. This chapter is about that foundation.

The Minimum Viable Knowledge

You don't need a CS degree. But you need *something*. Here's the honest list.

Must Have (Can't Surf Without It)

Git and GitHub

`git add`, `git commit`, `git push` - the holy trinity

Branching and merging (at least the basics)

- Reading a diff
- Understanding what a repository is

Why: Every AI tool assumes you're using version control. Claude Code commits for you. Copilot reads your repo. Without Git, you're swimming against the current.

Terminal Basics

Navigate folders (`cd`, `ls`, `pwd`)
Run commands (`npm`, `node`, `python`)

- Read output and errors
- Basic file operations (`cp`, `mv`, `rm`)

Why: AI gives you commands to run. If you can't run them, you're stuck.

File Structure Intuition

What's a `package.json`?
What's a `.env` file?

- What are `node_modules`?
- Where does code go vs configuration vs assets?

Why: AI generates files. You need to know where to put them.

Should Have (Makes Life Easier)

One Programming Language

JavaScript/TypeScript (most versatile for web)
Python (great for scripts and AI)

- You don't need to be an expert. You need to *read* code.

Why: AI writes code. You review it. If you can't read it at all, you can't catch mistakes.

Basic Debugging

Reading error messages

Using `console.log()` or `print()`

- Googling error messages (yes, still)

Why: AI makes mistakes. You'll need to debug them.

HTTP and APIs

What's a GET vs POST request?

What's JSON?

- What are headers?

Why: Modern apps are APIs talking to each other. This is the language.

Nice to Have (But Not Required)

Docker basics

Cloud concepts (AWS, Vercel)

- Database fundamentals
- Networking basics

You can learn these as you go. AI will help.

Git and GitHub Deep Dive

Let's spend more time here because it's crucial.

Why GitHub Specifically?

GitHub is the standard. Not GitLab, not Bitbucket (though they work). GitHub because:

AI tools integrate with it

Copilot is GitHub's product

- Actions for automation
- Everyone's on it

Essential Commands

Start a new repo

```
git init
```

Clone someone's repo

```
git clone https://github.com/user/repo.git
```

Daily workflow

```
git status              # What's changed?
git add .               # Stage everything
git commit -m "message" # Save changes
git push                # Upload to GitHub
```

Branching

```
git checkout -b feature  # New branch
git checkout main        # Back to main
git merge feature        # Merge branch
```

Oops

```
git stash               # Temporarily hide changes
git reset --soft HEAD~1 # Undo last commit (keep changes)
```

GitHub Features You Should Use

Issues - Track bugs and features **Actions** - Automate testing, deployment **Projects** - Kanban boards (we'll talk about this) **Discussions** - Community Q&A **Pages** - Free static hosting

The .gitignore File

Always have one:

```
node_modules/
.env
.DS_Store
*.log
dist/
```

Never commit: secrets, dependencies, build output.

Dictation and Voice Input

Here's a productivity multiplier most people ignore: **voice**.

Why Dictate?

Speaking is faster than typing (3-4x for most people)

Less RSI strain

- You can "code surf" while walking
- Natural language → AI prompt is seamless

Tools for Dictation

Wispr Flow (Mac)

Always listening (when activated)

Transcribes directly into any app

- Works with terminal, IDE, browser
- $10/month

Other options:

macOS built-in (free, decent)

Whisper (free, local, requires setup)

- Otter.ai (for meetings, not coding)

Workflow with Voice

```
[Speak] "Create a function that validates email addresses"
[Wispr types it]
[Claude generates code]
[Speak] "Now add support for plus addressing"
[Iterate]
```

You barely touch the keyboard. It's a different way of working.

Tips for Voice + AI

Speak in complete thoughts - Don't stop mid-sentence

Use punctuation - Say "period" and "comma" if needed

- **Code words are hard** - Spell out: "camelCase, C-A-M-E-L-C-A-S-E"
- **Review before sending** - Voice makes typos too

The Learning Curve

Here's the honest timeline:

Week 1-2: Fumbling

Everything takes longer than expected
You're learning the tools AND the concepts

- This is normal

Week 3-4: Clicking

Commands become muscle memory
You start predicting what AI will do

- First real productivity gains

Month 2-3: Flow

You forget how you worked before
New projects feel easy

- You have opinions about tools

Month 4+: Mastery

You teach others

You build custom workflows

- You see inefficiencies everywhere

Everyone goes through this. Don't get discouraged in week 1.

What You Don't Need

Things people think are required but aren't:

Computer Science degree - Not needed

Years of experience - Months are enough

Math skills - Rarely comes up

Perfect typing - Voice exists

- **Expensive hardware** - M1 MacBook Air is plenty
- **Multiple monitors** - Nice but optional

The barrier to entry is lower than ever. The limiting factor is *wanting to learn*.

Building the Foundation

If you're starting from zero, here's a 4-week plan:

Week 1: Git and Terminal

Complete a Git tutorial (GitHub has good ones)

Practice basic terminal commands daily

- Create your first repo

Week 2: JavaScript Basics

Variables, functions, objects

Just enough to read code

- Use an interactive tutorial (freeCodeCamp)

Week 3: Your First AI Project

Install Claude Code or Cursor

Build something simple (todo app, calculator)

- Get comfortable prompting

Week 4: Integration

Connect Git + AI + deployment

Push a project to GitHub

- Deploy to Vercel

After this, you're ready to code surf for real.

"The best time to plant a tree was 20 years ago. The second best time is now."

Next: Chapter 6: Building a Real Project →

Chapter 6: Building Something Real

Enough theory. Let's build.

This chapter walks through a real project: a Telegram bot that handles content approval for a publishing workflow. This is something we actually built and use daily.

The Problem

You're running a small publishing operation. You create content (posts, images, PDFs) and need someone to approve them before they go live. Email is slow. Slack is noisy. You want something simple.

Solution: A Telegram bot that sends content for approval and posts it when approved.

The Stack

Node.js - Runtime

Telegram Bot API - Messaging

- **X API** - Social posting
- **File system** - Simple storage (no database yet)

Step 1: Define What We Want

Before touching code, I described the bot to Claude:

```
I need a Telegram bot for content approval.

Workflow:

Bot receives a message with content (text, image, or both)
Bot sends it to me (admin) with Approve/Reject buttons

If I tap Approve, it posts to X
If I tap Reject, it asks for changes

  • Track what's been approved/rejected in a JSON file

Use node-telegram-bot-api package.
Store X API credentials in .env
```

This is my complete prompt. Notice:

Clear workflow (numbered steps)

Specific package suggestion

- Environment variable mention
- Simple storage decision

Step 2: Scaffold the Project

Claude generated:

```
// bot.js
const TelegramBot = require('node-telegram-bot-api');
const fs = require('fs');
require('dotenv').config();

const bot = new TelegramBot(process.env.TELEGRAM_BOT_TOKEN,
const ADMIN_CHAT_ID = process.env.ADMIN_CHAT_ID;

// Store pending approvals
let pending = {};

bot.on('message', async (msg) => {
  const chatId = msg.chat.id;

  // Only process from admin
  if (chatId.toString() !== ADMIN_CHAT_ID) {
    return bot.sendMessage(chatId, 'Unauthorized');
  }

  // Store content for approval
  const id = Date.now().toString();
  pending[id] = {
    text: msg.text,
    photo: msg.photo ? msg.photo[msg.photo.length - 1].file_
    timestamp: new Date().toISOString()
  };

  // Send approval request
  const keyboard = {
    inline_keyboard: [[
      { text: '✅ Approve', callback_data: `approve_${id}` },
      { text: '❌ Reject', callback_data: `reject_${id}` }
```

```
    ]]
  };

  await bot.sendMessage(ADMIN_CHAT_ID,
     New content:\n\n${msg.text || '[Image]'},
    { reply_markup: keyboard }
  );
});

// Handle button presses
bot.on('callback_query', async (query) => {
  const [action, id] = query.data.split('_');
  const content = pending[id];

  if (!content) {
    return bot.answerCallbackQuery(query.id, { text: 'Conten
  }

   if (action === 'approve') {
    // Post to X
    await postToX(content);
    delete pending[id];
    await bot.answerCallbackQuery(query.id, { text: 'Posted!
  } else {
    delete pending[id];
    await bot.answerCallbackQuery(query.id, { text: 'Rejecte
  }
});
```

First iteration: ~50 lines. Working prototype.

Step 3: Iterate

Me: "Add persistence so pending items survive restarts"

Claude added:

```
const PENDING_FILE = './pending.json';

function loadPending() {
  try {
    return JSON.parse(fs.readFileSync(PENDING_FILE, 'utf8'))
  } catch {
    return {};
  }
}

function savePending() {
  fs.writeFileSync(PENDING_FILE, JSON.stringify(pending, nul
}
```

Me: "Add a /status command that shows pending count"

Claude added:

```
bot.onText(/\/status/, (msg) => {
  const count = Object.keys(pending).length;
  bot.sendMessage(msg.chat.id, Pending items: ${count});
});
```

Me: "Add error handling for the X posting"

```
try {
  await postToX(content);
  bot.sendMessage(ADMIN_CHAT_ID, '✅ Posted successfully');
```

```
} catch (error) {
  bot.sendMessage(ADMIN_CHAT_ID, ❌ Failed: ${error.message}
}
```

Each iteration: 2-5 minutes. Total build time: ~30 minutes.

Step 4: Deploy

For a simple bot, this is enough:

```
npm install
node bot.js
```

Keep it running with pm2:

```
npm install -g pm2
pm2 start bot.js --name "approval-bot"
pm2 save
```

Now it runs forever (until the server restarts, then pm2 brings it back).

The Finished Product

After iterations, the bot:

Receives content via Telegram

Shows inline approve/reject buttons

Posts approved content to X

Logs everything

- Handles errors gracefully
- Persists state to disk

Total code: ~150 lines Build time: ~1 hour (including testing) Value delivered: Saves 10+ minutes per approval

What We Learned

The prompt matters more than the code. A clear description of what you want beats vague requests.

Iterate in small steps. Don't ask for everything at once. Add features one by one.

Test as you go. Run the bot after each change. Catch errors early.

Ship the MVP. The first version didn't have persistence. It worked. We added it later.

Your Turn

Try building something similar:

A bot that reminds you to drink water

A script that backs up a folder to cloud storage

- A CLI that fetches weather and displays it nicely

Start simple. Add features. Ship.

"Done is better than perfect. But done AND working is best."

Next: Chapter 7: Building PDFs and Documents →

Chapter 7: Building PDFs and Documents

PDF generation is one of those tasks that sounds simple until you try it. This chapter shows how to build a complete book production pipeline.

The Problem

You want to generate professional PDFs:

Books with proper layouts

Reports with headers and footers

- Documents with embedded images
- Multi-page spreads

Traditional approach: Learn LaTeX, fight with Word, use complex desktop software.

Code surfing approach: HTML → PDF via Puppeteer.

Why HTML?

HTML is the universal format:

You already know it (or can learn in an hour)

CSS handles layout, fonts, colors

- AI understands it perfectly
- Renders consistently across platforms

The trick: **Puppeteer converts HTML to PDF via headless Chrome.** Chrome's print engine is actually excellent.

The Stack

```
HTML + CSS (your content)
    ↓
Node.js script
    ↓
Puppeteer (headless Chrome)
    ↓
Professional PDF
```

Step 1: Basic PDF Script

Ask Claude:

```
Create a Node.js script that:
Reads an HTML file
Converts it to PDF using Puppeteer
```

- Supports A4 and US Letter sizes
- Adds page numbers

`Use puppeteer. Keep it simple.`

You'll get something like:

```
const puppeteer = require('puppeteer');
const fs = require('fs');

async function htmlToPdf(inputPath, outputPath, options = {}
  const browser = await puppeteer.launch();
  const page = await browser.newPage();

  const html = fs.readFileSync(inputPath, 'utf8');
  await page.setContent(html, { waitUntil: 'networkidle0' })

  await page.pdf({
    path: outputPath,
    format: options.format || 'A4',
    printBackground: true,
    margin: {
      top: '1in',
      bottom: '1in',
      left: '1in',
      right: '1in'
    },
    displayHeaderFooter: true,
    footerTemplate:
```

/

```
    ,
    headerTemplate: '
'
  });

  await browser.close();
  console.log( PDF created: ${outputPath} );
}

htmlToPdf('book.html', 'book.pdf');
```

Run it: `node create-pdf.js`

Done. You have PDF generation.

Step 2: Book Template

Now ask for a proper book layout:

```
Create an HTML template for a children's book:
8.5x8.5 inch square format
Full bleed images


  • Large readable fonts
  • Print-ready CSS

Include CSS for page breaks, margins, and bleed.
```

The CSS for print is specific:

```
@page {
  size: 8.5in 8.5in;
  margin: 0;
}

.page {
  width: 8.5in;
  height: 8.5in;
  page-break-after: always;
  position: relative;
  overflow: hidden;
}

.bleed {
  position: absolute;
  top: -0.125in;
  left: -0.125in;
  right: -0.125in;
  bottom: -0.125in;
}

.safe-area {
  position: absolute;
  top: 0.5in;
  left: 0.5in;
  right: 0.5in;
  bottom: 0.5in;
}
```

Step 3: Handling Images

Images in PDFs can be tricky. The solution: **base64 encoding**.

```
function imageToBase64(imagePath) {
  const image = fs.readFileSync(imagePath);
  const ext = imagePath.split('.').pop();
  return ${image.toString('base64')}
}

 // In your HTML
const html =

;
```

Now images are embedded directly in the HTML. No broken links, no missing files.

Step 4: Multi-Page Books

For a real book with multiple pages:

```
function createBookHtml(pages) {
  return
```

```
      ${pages.map((page, i) =>
      
      
      
      
      
${page.text}
      
      
      
      
      ).join('')}
      
  ;
}
```

Real Example: Children's Book Pipeline

Here's our actual workflow for Onde books:

- **Content in JSON**

```
{
  "title": "AIKO",
  "pages": [
    {
      "text": "Sofia received a special gift...",
```

```
      "image": "images/chapter1.jpg"
    }
  ]
}
```

Generate images (Grok or DALL-E)

Run script → PDF

- **Send to Telegram** for approval
- **Upload to KDP** when approved

Total time: 30 minutes for a complete book PDF.

Tips for Quality

Fonts:

```
@import url('https://fonts.googleapis.com/css2?family=...');
```

Or embed locally for offline use.

Image quality:

Use 300 DPI for print

JPEG for photos, PNG for illustrations

- Test at actual print size

Page breaks:

```
.page { page-break-after: always; }
.no-break { page-break-inside: avoid; }
```

Testing:

Always open PDF and check EVERY page

Text overlapping images = bad

- Bleed extends to edges = good

Common Mistakes

Images not loading: Use base64 or absolute paths.

Text cut off: Check margins and safe areas.

Blurry images: Source images too small. Use 2x resolution.

Wrong colors: Set `printBackground: true` in Puppeteer.

Slow generation: Reduce image sizes, limit pages per run.

The Automation Path

Once the script works:

Watch folder for new content

Auto-generate PDF on change

- Send preview to Telegram

- One-click approval → upload to KDP

This is the pipeline we built. Books go from idea to published in hours, not weeks.

What AI Can't Do (Yet)

Choose the right image for a page (taste required)

Judge if text and image work together (human eye)

- Decide final layout (design sense)

AI generates. You curate. That's the partnership.

"The best tool is the one you actually use."

Next: Chapter 8: Common Pitfalls →

Chapter 8: Common Pitfalls and How to Avoid Them

Code surfing isn't magic. It's a skill with real failure modes. This chapter is about the traps and how to escape them.

The Hallucination Trap

What happens: AI confidently generates code that looks right but doesn't work. Functions that don't exist. APIs with wrong parameters. Libraries that were never written.

Example:

```
// AI generated this
import { validateEmail } from 'email-validator-pro';

// Problem: that package doesn't exist
// Or it exists but with different API
```

How to avoid:

Test immediately - Run code after every generation

Verify packages - Check npm/PyPI before installing

- **Read the code** - Don't just copy-paste
- **Ask for sources** - "What package is this from? Link to docs?"

When to suspect hallucination:

Package names you've never heard of

APIs that seem too convenient

- Code that's suspiciously perfect
- Methods that sound made up

The Over-Engineering Spiral

What happens: You ask for a simple feature. AI gives you an enterprise architecture with 15 files, 3 design patterns, and a configuration system.

Example:

```
You: "Add a function to save user preferences"

 AI: "I'll create a PreferenceManager class with
a Strategy pattern for different storage backends,
a Factory for preference types, an Observer for
changes, and a Memento for undo support..."
```

How to avoid:

Be explicit about simplicity - "Keep it under 20 lines"

Start minimal - "Just use a JSON file"

- **Push back** - "That's too complex. Simpler please."
- **Set constraints** - "One file only. No classes."

Red flags:

More than 3 files for a simple feature

Abstractions before you need them

- "Future-proofing" for futures that won't happen
- Design patterns for their own sake

The Context Loss Problem

What happens: After many iterations, AI forgets what you discussed earlier. It contradicts previous decisions. It reimplements things differently.

Example:

```
Round 1: "Use camelCase for all variables"
...50 messages later...
AI: "Here's the code with snake_case variables"
```

How to avoid:

Create a CLAUDE.md file - Project rules that persist

Summarize periodically - "So far we've agreed on X, Y, Z"

- **Reference earlier decisions** - "As we decided in Round 1..."
- **Start fresh sessions** - Context resets can be good

Tools that help:

Claude's extended context (200K tokens)

Project files that AI reads at start

- Explicit constraints in each prompt

The Sunk Cost Fallacy

What happens: You've spent 30 minutes going down a path. It's clearly wrong. But you keep pushing because you've "invested" time.

Signs you're stuck:

Same error after 5+ attempts

Code getting more complex, not simpler

- You're not sure what you're even building anymore
- The solution feels forced

The fix:

Just say:

```
"Let's start fresh. Forget everything we've tried.
Here's what I actually need: [clear description]"
```

No shame. No wasted effort. The 30 minutes taught you what doesn't work.

Security Blindspots

What happens: AI generates code that works but has security holes. It doesn't think about attacks unless you ask.

Common issues:

SQL injection in database queries

XSS in user-rendered content

Hardcoded secrets in code

Missing input validation

- Exposed error messages

How to avoid:

Explicitly ask - "Review this for security issues"

Use security prompts - "Add input validation"

- **Know the basics** - OWASP Top 10
- **Don't trust user input** - Ever

Example prompt:

```
Review this code for:
SQL injection
XSS vulnerabilities
```

```
- Input validation
- Secret exposure


  - Error handling
```

The "It Works on My Machine" Problem

What happens: Code works in development. Fails in production. Different environments, different results.

Common causes:

Missing environment variables

Different Node/Python versions

Missing dependencies

Path differences

- Permission issues

How to avoid:

Use .env files - And document required variables

Lock versions - package-lock.json, requirements.txt

- **Test in production-like environment** - Docker helps
- **Log everything** - Especially on first deploy

The Copy-Paste Programmer Trap

What happens: You copy AI code without understanding it. It works. Until it doesn't. And you can't debug it.

The problem:

You don't know why decisions were made

You can't modify it confidently

- Bugs are mysterious
- Technical debt accumulates

How to avoid:

Read every line - Even if you don't write it

Ask "why" - "Why did you use this approach?"

- **Refactor to understand** - Rewriting reveals structure
- **Comment as you go** - Force yourself to explain

The test: Can you explain the code to someone else? If not, you don't understand it yet.

The "AI Should Know" Assumption

What happens: You assume AI knows your project, your preferences, your constraints. It doesn't.

Reality check:

AI doesn't know your file structure (unless you show it)

AI doesn't know your coding style (unless you tell it)

- AI doesn't know your business goals (unless you explain)
- AI doesn't know what you tried before (unless you share)

Fix: Over-communicate context. More is better.

The Premature Optimization Trap

What happens: You ask AI to optimize code that doesn't need optimizing. Hours spent on microseconds.

When to optimize:

You've measured and found a bottleneck

Users are complaining about speed

- Costs are actually high

When NOT to optimize:

"It might be slow someday"

"Best practices say..."

- "It feels inefficient"

Rule: Make it work. Make it right. Make it fast. In that order.

The Tools Obsession

What happens: You spend more time evaluating AI tools than using them. Every week there's a new "game changer."

Reality:

80% of tools do the same thing

The best tool is one you know well

- Switching has real costs
- Mastery beats novelty

Healthy approach:

Pick a tool (Claude, Cursor, whatever)

Use it for 3 months

- Evaluate new tools quarterly, not daily
- Only switch if there's 10x improvement

Summary: The Meta-Skill

The real skill in code surfing is **knowing when AI is wrong**.

This requires:

Basic coding knowledge (to spot errors)

Testing habits (to catch bugs)

- Humility (to admit you're stuck)

- Judgment (to know when to stop)

AI makes you faster. It doesn't make you right. That's still your job.

"Trust, but verify." —Ronald Reagan (also applies to AI)

Next: Chapter 9: Shipping and Iterating →

Chapter 9: Shipping and Iterating

The most important feature is shipping. Everything else is preparation.

Good Enough Is Good Enough

Perfectionism kills projects. The graveyard of unshipped software is vast.

The 80/20 rule:

80% of the value comes from 20% of the features
The last 20% of polish takes 80% of the time

- Users care about 80% less than you think

Practical application:

Build the core feature
Make it work reliably

- Ship it
- Add polish based on feedback

The MVP Mindset

Minimum Viable Product. We've all heard it. Few actually do it.

What MVP really means:

One feature done well

Basic but functional UI

- Works 95% of the time
- Can be deployed today

What MVP doesn't mean:

Half-finished features

Broken functionality

- "We'll fix that later" (and never do)
- Embarrassing quality

The test: Would you use this yourself? If yes, ship it.

Deployment: Keep It Simple

For most projects, you don't need Kubernetes. You need something that works.

Tier 1: Static sites

Vercel, Netlify, GitHub Pages

Free tier is enough

- Deploy in 2 minutes

Tier 2: Simple backends

Vercel Serverless Functions

Railway, Render

- $5-20/month

Tier 3: Complex apps

VPS (DigitalOcean, Linode)

AWS/GCP (if you need it)

- Docker when it helps, not just because

The progression: Start at Tier 1. Move up only when you hit real limits.

The Feedback Loop

Ship → Measure → Learn → Ship again.

What to measure:

Do people use it? (analytics)

Do they come back? (retention)

- What breaks? (error tracking)
- What do they ask for? (feedback)

Tools that help:

Plausible/Simple Analytics (privacy-friendly)

Sentry (error tracking)

- Telegram bot (for personal feedback)

Versioning with AI

AI makes versioning easier, not optional.

The workflow:

```
git commit -m "feat: add user preferences"
git push


Deploy automatically vi


Next day


"Claude, add dark mode to user preferences"
git commit -m "feat: add dark mode toggle"
git push
```

Each commit is a checkpoint. If something breaks, you can always go back.

Continuous Improvement

The shipped product is version 1.0. There will be many more versions.

Weekly improvement cycle:

Monday: Review feedback from last week

Tuesday-Thursday: Implement top 2-3 requests

- Friday: Ship updates
- Weekend: Let it run

What to improve:

Bugs users actually hit

Features users actually request

- Performance users actually notice

What to ignore:

Edge cases with 0 users

"Nice to have" that no one asked for

- Optimizations for scale you don't have

When to Stop

Not every project needs to run forever.

It's okay to stop when:

The problem is solved (for you)

No one else uses it

- Maintenance exceeds value
- You've learned what you needed

How to stop gracefully:

Document how it works

Archive the repo (don't delete)

- Write a post-mortem for yourself
- Move on without guilt

The Real ROI

What did code surfing actually give you?

Speed: Days become hours. Hours become minutes.

Capability: You build things you couldn't before.

Learning: You understand more by building more.

Shipping: The most important output.

Your 24-Hour Challenge

Here's how to test everything in this book:

Hour 0-1: Pick a project

Something you actually want
Simple enough for a day

- Useful enough to keep

Hour 1-4: Build it

Use Claude/Cursor
Iterate in small steps

- Test as you go

Hour 4-6: Polish

Error handling
Basic UI cleanup

- Documentation (README)

Hour 6-8: Deploy

Push to GitHub
Deploy to Vercel/Railway

- Share the link

The rest of the 24 hours: Sleep. You've shipped something real.

What Comes Next

Code surfing is just the beginning.

Near future:

Agents that build while you sleep

Voice-first development

- AI that truly understands codebases

What doesn't change:

Taste matters

Shipping matters

- Users matter
- Quality matters

The tools evolve. The skills compound.

Final Thoughts

This book was written by an AI, about working with AI, for people who want to build with AI.

That's not ironic. That's the point.

The future of building is collaborative. Human creativity plus machine capability. Your vision plus AI execution.

The wave is here. The surfboard is ready.

Time to ride.

"The secret of getting ahead is getting started." —Mark Twain

---

Appendices

Appendix A: Tool Comparison Chart

Last Updated: January 2026

This will be outdated. Check the portal for live updates.

Code Generation Tools

ToolBest ForCostRating

Claude Code Complex reasoning, file editing, long context, CLI power$20/mo + API ($50-200/mo heavy use)⭐⭐⭐⭐⭐ Daily Driver **Claude (Web)**Quick questions, brainstormingFree tier / $20/mo Pro ⭐ ⭐ ⭐ ⭐ **Cursor**AI-native IDE, inline editing$20/mo ⭐ ⭐ ⭐ ⭐ Great IDE **GitHub Copilot**Autocomplete, quick suggestions$10/mo⭐⭐⭐ Nice to Have **GPT-4**General purpose, plugins$20/mo ⭐ ⭐ ⭐ **Gemini**Google integration, long docsFree / $20/mo⭐⭐⭐

Image Generation

ToolBest ForCostRating

GrokFast iterations, X integration$8-16/mo (X Premium)⭐⭐⭐⭐⭐ Best Value **DALL-E 3**Reliable, OpenAI ecosystemPay per use⭐⭐⭐⭐ **Midjourney**Artistic quality$10-60/mo ⭐⭐⭐⭐ Beautiful **Stable Diffusion**Local control, customizationFree (self-hosted) ⭐⭐⭐ Power Users **Ideogram**Text in imagesFree tier⭐⭐⭐ Niche

Browser Automation

ToolBest ForCostRating

Claude in ChromeAI-controlled browsingFree (with Claude)⭐⭐⭐⭐⭐ Game Changer **Playwright**Programmatic browser controlFree⭐⭐⭐⭐ **Puppeteer**Headless Chrome, PDFsFree ⭐⭐⭐⭐ **Selenium**Cross-browser testingFree⭐⭐⭐

Development Environment

ToolBest ForCostRating

VS Code Free, extensible, universalFree⭐⭐⭐⭐⭐ **Cursor**AI-first experience$20/mo⭐⭐⭐⭐ **Zed**Speed, collaborationFree (beta)⭐⭐⭐⭐ **Terminal + Claude Code**Maximum controlFree (plus API)⭐⭐⭐⭐⭐

Deployment

ToolBest ForCostRating

VercelNext.js, static sites, serverlessFree / $20/mo Pro ⭐ ⭐ ⭐ ⭐ ⭐ **Netlify**Static sites, formsFree / $19/mo⭐ ⭐ ⭐ ⭐ **Railway**Full stack apps$5/mo+⭐ ⭐ ⭐ ⭐ **Render**Docker, databasesFree / $7/mo+ ⭐ ⭐ ⭐ ⭐ **DigitalOcean**VPS, more control$5/mo+⭐⭐⭐⭐

Voice Input

ToolBest ForCostRating

Wispr Flow Always-on dictation (Mac)$10/mo ⭐ ⭐ ⭐ ⭐ ⭐ **macOS Dictation**Built-in, quickFree ⭐ ⭐ ⭐ **Whisper**Local, privateFree (self-hosted)⭐⭐⭐

Recommended Stacks

Beginner ($30/mo)

Claude Pro ($20)

GitHub Copilot ($10)

- VS Code (free)
- Vercel (free tier)

Professional ($150/mo)

Claude Pro + API ($70-100)

Cursor ($20)

- X Premium for Grok ($8-16)
- Vercel Pro ($20)

Power User ($300/mo)

Claude API heavy usage ($150)

Cursor ($20)

X Premium ($16)

Vercel Pro ($20)

- Railway ($20)
- Various SaaS ($75)

Check the online vademecum for updated recommendations.

Appendix B: Prompt Templates

Copy-paste ready prompts for common tasks.

Project Setup

New Project Scaffold

```
Create a [LANGUAGE] project for [PURPOSE].

Structure:

src/ for source code
tests/ for tests


  • README.md with setup instructions

Include:

Package manager config (package.json / requirements.txt)
.gitignore


  • Basic linting config
```

```
Keep it minimal. No unnecessary dependencies.
```

Add Feature

```
I need to add [FEATURE] to [PROJECT].

Current relevant code:
[paste relevant files or describe structure]

Requirements:

- [Requirement 1]
- [Requirement 2]

Constraints:

[Max lines/files]
[Style preferences]


- [Dependencies to use/avoid]
```

Code Quality

Code Review

```
Review this code for:
Security issues (injection, XSS, etc.)
Performance problems
```

```
Code style consistency
Edge cases and error handling

  • Unnecessary complexity

[paste code]

Be direct. Tell me what's wrong and how to fix it.
```

Refactor Request

```
Refactor this code to be:
More readable
Less complex

  • Better structured

Keep the functionality identical.
Don't add features.
Don't over-engineer.

[paste code]
```

Bug Fix

```
Bug: [What's happening]
Expected: [What should happen]
Code: [relevant code]
Error message: [if any]
```

```
What I've tried: [previous attempts]

Help me fix this. Explain the root cause.
```

API and Backend

API Endpoint

```
Create an API endpoint that:
Method: [GET/POST/PUT/DELETE]
Path: /api/[path]

Input: [describe expected input]
Output: [describe expected response]

  - Errors: [what errors to handle]

Use [framework: Express/Fastify/etc].
Include input validation and error handling.
```

Database Query

```
I need a query that:
Database: [PostgreSQL/SQLite/MongoDB]
Purpose: [what data to get/update]

  - Tables/Collections: [list relevant ones]
```

```
Show me the query and explain it.
Include any indexes I should add.
```

Frontend

React Component

```
Create a React component for [PURPOSE].

Props:

- [prop1]: [type and description]
- [prop2]: [type and description]

Behavior:

- [behavior 1]
- [behavior 2]

Use [TypeScript/JavaScript].
Use [styling approach: Tailwind/CSS modules/etc].
Keep it simple and reusable.
```

Form with Validation

```
Create a form for [PURPOSE].

Fields:

- [field1]: [type, validation rules]
- [field2]: [type, validation rules]
```

```
On submit: [what should happen]
Show validation errors inline.
Use [form library if any: React Hook Form, etc].
```

DevOps

Docker Setup

```
Create a Dockerfile for [PROJECT TYPE].

Requirements:

Base image: [node:20/python:3.11/etc]
Build steps: [what needs to compile]


- Runtime: [what runs in production]
- Port: [exposed port]

Include docker-compose.yml if needed.
Keep images small.
```

CI/CD Pipeline

```
Create a GitHub Actions workflow that:
Runs on [push to main / PR / etc]
[Tests to run]


- [Build steps]
```

- [Deploy to: Vercel/Railway/etc]

Include caching for dependencies.
Fail fast on errors.

Documentation

README Template

Write a README for [PROJECT] that includes:
- One-line description
- Quick start (3 steps max)

- Configuration options
- Common commands

- Contributing guidelines (brief)

Keep it scannable. No walls of text.

API Documentation

Document this API:
[paste endpoints or code]

Format:

- Endpoint path and method
- Required headers

```
- Request body (with examples)
- Response format (with examples)

  - Error codes

Use markdown. Keep it practical.
```

Debugging

Error Investigation

```
I'm getting this error:
[paste full error message and stack trace]

Relevant code:
[paste code around the error]

Environment:

- [OS, Node version, etc]
- [Recent changes]

What's causing this and how do I fix it?
```

Performance Issue

```
This code is slow:
[paste code]

Context:
```

```
[Input size / frequency]
[Current performance: X seconds]

- [Target performance: Y seconds]

How can I make it faster?
Explain the bottleneck.
```

Architecture

Design Decision

```
I need to choose between:
A) [Option A with brief description]
B) [Option B with brief description]

Context:
[Project type and scale]
[Team size and experience]

- [Priorities: speed/maintainability/cost]

Which do you recommend and why?
What are the tradeoffs?
```

System Design

```
Design a system for [PURPOSE].
```

```
Requirements:
- [Scale: users, requests/second]
- [Data: types, volume]
- [Features: list core features]

Constraints:
- [Budget]
- [Team size]
- [Timeline]

Give me a high-level architecture.
Identify the key decisions.
```

Customize these for your projects. The best prompts are specific.

Appendix C: Project Ideas to Practice

10 projects you can build in a day with code surfing.

Beginner (2-4 hours each)

1. Personal Dashboard

What: A single page showing your important info at a glance.

Features:

Current weather

Today's calendar events

- Quick links
- Motivational quote

Stack: HTML + CSS + JavaScript **Prompt**: "Create a personal dashboard that shows weather and links"

---

2. Markdown Note Taker

What: A simple app to write and save markdown notes.

Features:

Write in markdown

Preview rendered HTML

- Save to local storage
- Export to file

Stack: React or Vanilla JS **Prompt**: "Build a markdown editor with live preview and local storage"

---

3. Habit Tracker

What: Track daily habits with streaks.

Features:

Add habits

Check off daily

- Show streak count
- Simple stats

Stack: React + Local Storage **Prompt**: "Create a habit tracker with streak counting"

Intermediate (4-6 hours each)

4. Telegram Bot for Reminders

What: A bot that sends you reminders.

Features:

`/remind [time] [message]` - Set reminder

`/list` - Show pending reminders

- `/cancel [id]` - Cancel reminder
- Stores in JSON file

Stack: Node.js + node-telegram-bot-api **Prompt**: "Build a Telegram reminder bot with natural language time parsing"

---

5. URL Shortener

What: Create short links, track clicks.

Features:

Shorten URLs

Custom slugs

- Click counting
- Simple analytics

Stack: Next.js + SQLite **Prompt**: "Create a URL shortener with click tracking using Next.js"

---

6. Invoice Generator

What: Create and export professional invoices.

Features:

Company/client info
Line items with calculations

- Tax handling
- PDF export

Stack: React + jsPDF or Puppeteer **Prompt**: "Build an invoice generator that exports to PDF"

---

7. Content Approval Bot

What: Review content before posting to social media.

Features:

Receive content via Telegram
Show approve/reject buttons

- Post to X when approved

- Log history

Stack: Node.js + Telegram Bot API + X API **Prompt**: "Create a Telegram bot for approving social media posts"

Advanced (6-10 hours each)

8. Personal API

What: A unified API for your personal data.

Features:

Endpoints for: /now, /projects, /bookmarks

Simple auth

- CORS for embedding
- JSON responses

Stack: Express or Fastify + SQLite **Prompt**: "Build a personal API with endpoints for /now and /projects"

---

9. Expense Tracker with Categories

What: Track spending with categories and reports.

Features:

Add expenses with category

Monthly/weekly views

- Category breakdown
- Export to CSV

Stack: Next.js + Postgres (or SQLite) **Prompt**: "Create an expense tracker with category-based reporting"

---

10. AI Writing Assistant

What: A tool to help with writing tasks.

Features:

Paste text, get suggestions

Improve clarity

- Fix grammar
- Multiple tones

Stack: Next.js + OpenAI or Claude API **Prompt**: "Build a writing assistant that improves text clarity using AI"

Project Selection Guide

Pick based on your interest:

Into productivity? → Habit Tracker, Dashboard

Like automation? → Telegram Bots

- Building a business? → Invoice Generator, URL Shortener
- Learning APIs? → Personal API, AI Assistant

Pick based on your skill level:

First project? → Markdown Note Taker

Some experience? → Telegram Bot

- Want a challenge? → AI Writing Assistant

The meta-rule: Build something you'll actually use. Motivation comes from utility.

How to Approach Each Project

Start with the prompt - Describe what you want

Get the scaffold - Basic structure and files

Add core feature - The one thing it must do

Test it - Does it actually work?

Add second feature - Iterate

Deploy - Make it real

- **Use it** - The real test

Don't build all 10. Build 1, ship it, use it. Then build the next.

"The best project is the one you finish."