DEVELOPER TOOLS

The Vibe Coding Starter Kit

Build real apps with AI as your co-pilot: prompts, workflows, and project templates

by Rook

First Edition · February 2026

Table of Contents

Vibe Coding Starter Kit

CHAPTER 01

System Prompts & Workflows for AI-Assisted Development

Stop writing code. Start directing it.

CHAPTER 02

What is Vibe Coding?

Vibe coding is a development philosophy where you describe what you want in natural language and let an AI assistant write the implementation. You're not pair programming — you're directing. Think of yourself as a technical product manager who happens to sit inside an IDE.

The term caught fire in early 2025, but developers have been doing this since GPT-4 landed in coding tools. What's changed is that it actually *works* now. Not for everything, but for a surprisingly large chunk of day-to-day development.

When it works brilliantly:

- CRUD APIs and standard backend routes
- React components with well-known patterns
- Unit tests (AI is genuinely excellent at this)
- Boilerplate: config files, Docker setups, CI pipelines
- Converting designs to code when you describe them clearly
- Refactoring existing code to a new pattern

When it fails spectacularly:

- Complex state management across many files
- Performance-critical code where you need to reason about memory
- Anything requiring deep domain knowledge (financial calculations, medical logic)
- Novel algorithms — AI can only remix what it's seen
- Security-sensitive code (auth flows, encryption) — always review manually

The skill shift is real. Traditional development rewards knowing syntax, memorising APIs, and typing fast. Vibe coding rewards clear thinking, precise descriptions, and knowing *what* to build. You spend less time in documentation and more time in conversation. The developers who thrive aren't the fastest typists — they're the clearest thinkers.

The catch: you still need to understand the code AI produces. Vibe coding without code literacy is just generating technical debt at superhuman speed. This guide assumes you can read code and spot problems. What it teaches you is how to get AI to produce better code in the first place.

CHAPTER 03

Tool Setup Guide

▸ Cursor

Cursor is a fork of VS Code built around AI-first development. It's currently the best all-round vibe coding environment.

Setup: 1. Download from cursor.com 2. Import your VS Code settings (it'll offer on first launch) 3. Sign up for Pro (£16/month) — the free tier is too limited for serious work

Essential keybindings: - `Cmd+K` / `Ctrl+K` — Inline edit (highlight code, describe the change) - `Cmd+L` / `Ctrl+L` — Open chat panel - `Cmd+I` / `Ctrl+I` — Composer (multi-file generation) - `Tab` — Accept autocomplete suggestion - `Esc` — Dismiss suggestion

Best practices: - Use `.cursorrules` files (covered in detail in Section 3) to set project-wide context - Keep the chat panel open — it remembers conversation context within a session - Use `@file` to reference specific files in chat: "Refactor @utils/auth.ts to use JWT instead of sessions" - Composer mode for anything touching 3+ files; inline edit for single-file changes - Set your preferred model in Settings → Models (Claude Sonnet 4 for speed, Claude Opus 4 for complex work)

▸ GitHub Copilot

Still the most widely-used AI coding tool, and it's improved dramatically.

Setup: 1. Install the GitHub Copilot extension in VS Code 2. Enable Copilot Chat (comes bundled now) 3. Settings to tweak: - `github.copilot.enable` — toggle per language - `github.copilot.advanced` — set temperature lower (0.1) for more predictable output

Best use cases: - Line-by-line autocomplete while you're already writing code - Copilot Chat for quick explanations ("what does this regex do?") - `/fix` command in chat for error resolution - `/tests` command to generate test files

Copilot is best when you're *writing alongside it*, not when you're trying to generate whole files from scratch.

▸ Aider

Aider is the power tool. It's a CLI that connects to any LLM and edits your actual files on disc.

Setup:

```
pip install aider-chat
cd your-project
aider --model claude-sonnet-4
```

Model selection: - `claude-sonnet-4` — best balance of speed and quality (recommended default) - `claude-opus-4` — use for complex refactors, architecture work - `gpt-4o` — solid alternative, better at certain Python patterns - `deepseek-chat` — budget option, surprisingly capable

Key commands inside Aider: - `/add filename` — add files to context - `/drop filename` — remove files from context - `/ask` — ask without making changes - `/diff` — show pending changes before applying - `/undo` — revert last change

Why Aider over Cursor? Context control. You explicitly choose which files the AI can see and edit. For large codebases, this precision matters enormously.

▸ Claude Code (CLI)

Anthropic's official CLI tool. Think of it as an AI pair programmer that lives in your terminal.

Setup:

```
npm install -g @anthropic-ai/claude-code
cd your-project
claude
```

Strengths: - Reads your entire project structure automatically - Can run commands, create files, and execute tests - Excellent at multi-step tasks ("set up a new API endpoint with tests and update the router") - CLAUDE.md file in your project root acts like `.cursorrules` for Claude Code

When to use which tool:

TASK	BEST TOOL
Quick autocomplete while typing	Copilot
Single-file edits with context	Cursor inline
Multi-file feature generation	Cursor Composer
Large refactors in big codebases	Aider
Full feature with tests + docs	Claude Code
Exploring/prototyping ideas	Any chat interface

CHAPTER 04

System Prompts Collection

The `.cursorrules` file (or `CLAUDE.md` for Claude Code) sits in your project root and gives the AI persistent context about your project. It’s the single highest-leverage thing you can do to improve AI output quality.

Every project type needs different rules. Here are ten battle-tested configs.

▶ 3.1 React / Next.js App

```
You are an expert React and Next.js developer.

Tech stack:
- Next.js 15 with App Router
- TypeScript (strict mode)
- Tailwind CSS for styling
- shadcn/ui component library
- Zustand for client state
- TanStack Query for server state

Code style:
- Use functional components exclusively. No class components.
- Prefer named exports over default exports.
- Use 'use client' directive only when the component genuinely needs client-side i
- Server Components are the default. Do not add 'use client' unless the component 
- Collocate types with their components. Only extract to a shared types file if us
- Use Tailwind utility classes directly. Do not create CSS modules or styled-compo
- Prefer composition over prop drilling. Use compound component patterns for compl

File naming:
- Components: PascalCase (UserProfile.tsx)
- Utilities: camelCase (formatDate.ts)
- Routes: kebab-case directories (app/user-profile/page.tsx)

Patterns:
- Data fetching happens in Server Components or route handlers. Never fetch in cli
- Form handling: use react-hook-form with zod validation schemas.
- Error boundaries: every route segment should have an error.tsx.
- Loading states: use Suspense boundaries with skeleton components, not spinner ic

Do not:
- Use the pages/ directory. This project uses App Router exclusively.
- Install new dependencies without asking first.
- Use any state management other than Zustand and TanStack Query.
- Write inline styles or CSS-in-JS.
```

Why these rules matter: The biggest issue with AI-generated React code is inconsistency. Without rules, you'll get a mix of client and server components, random state management choices, and three different styling approaches in one project. This config eliminates that drift.

▶ 3.2 Python FastAPI Backend

```
You are an expert Python backend developer using FastAPI.

Tech stack:
- Python 3.12+
- FastAPI with async handlers
- SQLAlchemy 2.0 with async sessions
- Pydantic v2 for validation
- Alembic for migrations
- pytest with pytest-asyncio for testing

Code style:
- All route handlers must be async.
- Use type hints everywhere. No untyped function signatures.
- Pydantic models for all request/response schemas. Never return raw dicts.
- Use dependency injection for database sessions, auth, and config.
- Keep route handlers thin: business logic belongs in service modules.

Project structure:
- app/api/routes/ — route definitions grouped by domain
- app/models/ — SQLAlchemy models
- app/schemas/ — Pydantic request/response models
- app/services/ — business logic
- app/core/ — config, security, dependencies
- tests/ — mirrors app/ structure

Patterns:
- Every endpoint returns a typed Pydantic response model.
- Database queries go in repository classes, not in route handlers.
- Use BackgroundTasks for non-blocking operations (emails, webhooks).
- All datetimes are UTC. Use datetime.datetime with timezone info.
- Pagination: use cursor-based pagination, not offset/limit.

Error handling:
- Raise HTTPException with specific status codes and detail messages.
- Never catch broad Exception classes. Catch specific exceptions.
- Log errors with structlog, not print statements.

Do not:
- Use synchronous database calls.
- Return raw SQLAlchemy model instances from endpoints.
- Use global mutable state.
- Write SQL strings directly. Use SQLAlchemy's query builder.
```

3.3 Node.js / Express API

```
You are an expert Node.js backend developer.

Tech stack:
- Node.js 22 LTS
- Express 5 with TypeScript
- Prisma ORM with PostgreSQL
- Zod for runtime validation
- Jest for testing
- Winston for logging

Code style:
- TypeScript strict mode. No 'any' types.
- Async/await everywhere. No callback-style code.
- Use ES module syntax (import/export), not CommonJS require.
- Prefer const over let. Never use var.
- Functions should do one thing. Max 30 lines per function.

Architecture:
- routes/ — Express route definitions (thin, validation + delegation only)
- controllers/ — request handling and response formatting
- services/ — business logic (framework-agnostic, testable)
- repositories/ — database access via Prisma
- middleware/ — auth, error handling, validation, rate limiting
- types/ — shared TypeScript interfaces and types

Patterns:
- Validate all input at the route level using Zod schemas.
- Centralised error handling middleware. Routes throw typed errors, middleware cat
- Use dependency injection: services receive their dependencies as constructor par
- Environment config via a validated config module, not raw process.env access.
- All responses follow the shape: { data, meta, errors }.

Do not:
- Use Mongoose or MongoDB. This project uses PostgreSQL via Prisma.
- Put business logic in route handlers or controllers.
- Use try/catch in every route. Let errors propagate to the error middleware.
- Install Express middleware without discussing it first.
```

3.4 Mobile App (React Native)

```
You are an expert React Native developer building a cross-platform mobile app.

Tech stack:
- React Native 0.76+ (New Architecture enabled)
- Expo SDK 52 (managed workflow)
- TypeScript strict mode
- React Navigation v7 (native stack)
- Zustand for state management
- TanStack Query for API calls
- expo-secure-store for sensitive data

Code style:
- Functional components with hooks only.
- StyleSheet.create for all styles. No inline style objects.
- Prefer platform-agnostic components. Use Platform.select only when truly necessa
- Dimensions: use useWindowDimensions hook, not Dimensions.get (it doesn't update

Patterns:
- Navigation: typed route params using RootStackParamList.
- Forms: react-hook-form with zod schemas (same as web).
- Images: use expo-image, not React Native's Image component.
- Lists: FlatList with keyExtractor for all scrollable lists. Never use ScrollView
- Animations: use react-native-reanimated for performant animations. No Animated A

Performance:
- Memoize expensive computations with useMemo.
- Use React.memo on list item components.
- Avoid anonymous functions in render (extract to named handlers).
- Use FlashList instead of FlatList for lists with 100+ items.

Do not:
- Use class components.
- Access AsyncStorage for sensitive data (use expo-secure-store).
- Use absolute pixel values for layout. Use flex and percentage-based sizing.
- Install native modules that aren't Expo-compatible without discussing first.
```

3.5 CLI Tool

```
You are an expert developer building a command-line tool.

Tech stack:
- Node.js 22 with TypeScript
- Commander.js for argument parsing
- Chalk for coloured output
- Ora for spinners/progress
- Inquirer for interactive prompts
- Vitest for testing

Code style:
- Single entry point: src/index.ts (thin, just wires up commands)
- Each command in its own file under src/commands/
- Shared utilities in src/utils/
- Config handling in src/config.ts

Patterns:
- Every command has: name, description, arguments, options, and a handler function
- Validate all user input before processing. Fail fast with clear error messages.
- Exit codes: 0 for success, 1 for user errors, 2 for system errors.
- Long-running operations must show a spinner with status updates.
- Support both interactive and non-interactive (piped) usage.
- Config files: use XDG base directory spec (~/.config/toolname/).
- Output: structured data goes to stdout, human messages go to stderr.

Error messages:
- Always suggest what the user should do next.
- Bad: "Error: file not found"
- Good: "Could not find config.yml. Run 'mytool init' to create one, or specify a

Do not:
- Use console.log for user-facing output. Use the chalk-wrapped output helpers.
- Assume a particular shell or OS. Test cross-platform.
- Require global installation. Support npx usage.
```

3.6 Data Pipeline

```
You are an expert data engineer building ETL/ELT pipelines.

Tech stack:
- Python 3.12+
- Polars for data manipulation (not pandas)
- DuckDB for analytical queries
- Prefect for orchestration
- Great Expectations for data validation
- PyArrow for file I/O (Parquet format)

Code style:
- Type hints on every function. Use Polars type system for DataFrame operations.
- Immutable transformations: never mutate DataFrames in place. Chain operations.
- Each pipeline stage is a pure function: input DataFrame → output DataFrame.

Patterns:
- Extract, Transform, Load as separate modules.
- Schema validation at every boundary (after extract, before load).
- Idempotent operations: running the same pipeline twice produces the same result.
- Incremental processing by default. Full refreshes only when explicitly requested
- Logging: log row counts at each stage, flag anomalies (>20% change in row count)
- Date handling: all timestamps in UTC, partitioned by date.

Do not:
- Use pandas. This project uses Polars for performance.
- Write CSV files. Use Parquet for all intermediate and output data.
- Use string concatenation for SQL queries. Use parameterised queries.
- Ignore null handling. Every transformation must explicitly handle nulls.
```

▸ 3.7 Chrome Extension

```
You are an expert Chrome extension developer.

Tech stack:
- Manifest V3 (required for Chrome Web Store)
- TypeScript
- Vite for bundling
- Tailwind CSS for popup/options UI
- Chrome Storage API for persistence

Architecture:
- src/background/ — service worker (event-driven, no persistent state)
- src/content/ — content scripts injected into pages
- src/popup/ — popup UI (small React app)
- src/options/ — options page
- src/shared/ — types, utilities, message definitions

Patterns:
- Message passing between content script ↔ background ↔ popup via typed messages.
- Define all message types in src/shared/messages.ts with discriminated unions.
- Storage: use chrome.storage.local for data, chrome.storage.sync for user prefere
- Permissions: request only what's needed. Use optional permissions where possible
- Content scripts: minimal footprint. Inject only what's necessary.

Security:
- Never use eval() or innerHTML with untrusted content.
- Content Security Policy must be strict in manifest.json.
- Sanitise all data from web pages before processing.
- Use externally_connectable cautiously.

Do not:
- Use Manifest V2 patterns (persistent background pages, browserAction).
- Bundle large libraries in content scripts (they run on every matched page).
- Store sensitive data in chrome.storage without encryption.
- Use broad host permissions. Prefer activeTab where possible.
```

▸ 3.8 Landing Page

```
You are an expert frontend developer building a high-converting landing page.

Tech stack:
- Astro 5 (static output)
- Tailwind CSS
- TypeScript for any interactive islands
- Motion One for animations
- Sharp for image optimisation

Code style:
- Semantic HTML first. Every section uses appropriate HTML5 elements.
- Mobile-first responsive design. Start with mobile layout, add md: and lg: breakp
- Performance budget: <100KB total JavaScript, <1s LCP on 3G.

Structure:
- src/sections/ — each page section as an Astro component
- src/components/ — reusable UI elements
- src/layouts/ — page layouts with SEO meta
- public/ — static assets (optimised images, fonts)

Patterns:
- Above the fold: headline, subheading, CTA button, hero image. No navigation link
- Social proof section: testimonials, logos, metrics.
- Feature sections: benefit-led headlines (what the user gets), not feature-led (w
- CTA repetition: primary CTA appears at minimum 3 times on the page.
- Images: use Astro's Image component for automatic optimisation. WebP format with

SEO:
- Unique title and meta description.
- Open Graph and Twitter Card meta tags.
- Structured data (JSON-LD) for the product/service.
- Canonical URL set explicitly.

Do not:
- Use a SPA framework for a landing page. Static HTML is faster.
- Add cookie banners or tracking scripts without discussing first.
- Use hero carousels or sliders. One strong message beats five rotating ones.
- Lazy-load above-the-fold images.
```

▸ 3.9 Full-Stack SaaS

```
You are an expert full-stack developer building a SaaS application.

Tech stack:
- Next.js 15 (App Router) — frontend and API routes
- TypeScript strict mode throughout
- PostgreSQL with Drizzle ORM
- NextAuth.js v5 for authentication
- Stripe for billing
- Resend for transactional email
- Tailwind CSS + shadcn/ui
- Vercel for deployment

Architecture:
- app/(marketing)/ — public pages (landing, pricing, docs)
- app/(app)/ — authenticated app pages (behind auth middleware)
- app/api/ — API routes
- lib/ — shared utilities, database, auth config
- components/ — reusable UI (organised by domain, not by type)

Patterns:
- Multi-tenancy: workspace/organisation model. Users belong to workspaces.
- Auth: server-side session checks in layouts, not individual pages.
- Billing: Stripe webhooks handle all subscription state changes. Never trust clie
- Database: use Drizzle migrations. Schema changes go through migration files, neve
- Feature flags: simple key-value in database, cached in memory. No external servi
- Background jobs: use Vercel Cron or Inngest for async work (emails, reports, cle

Security:
- CSRF protection on all mutations.
- Rate limiting on auth endpoints and API routes.
- Input validation with Zod on every API route and server action.
- Row-level security: every database query filters by workspace_id.

Do not:
- Build a custom auth system. Use NextAuth.js.
- Store files in the database. Use S3-compatible storage (R2, S3).
- Add an admin panel framework. Build admin routes within the app.
- Over-engineer early. No microservices, no event sourcing, no GraphQL unless the
```

▸ **3.10 Open Source Library**

```
You are an expert developer building a well-documented open source library.

Tech stack:
- TypeScript (strict, with declaration files)
- Vitest for testing
- tsup for bundling (ESM + CJS dual output)
- Changesets for version management
- TypeDoc for API documentation
- GitHub Actions for CI/CD

Code style:
- Zero runtime dependencies unless absolutely necessary. Justify every dependency.
- Tree-shakeable: use named exports, avoid barrel files with side effects.
- Pure functions where possible. Minimise internal mutable state.
- Every public function has JSDoc with @param, @returns, @example, and @throws.
- Generic type parameters over union types for flexibility.

Project structure:
- src/ — source code
- src/index.ts — public API surface (re-exports only)
- tests/ — test files mirroring src/ structure
- docs/ — additional documentation and guides
- examples/ — runnable example projects

Patterns:
- Semantic versioning strictly followed. Breaking changes = major bump.
- Every PR must include: code changes, tests, documentation updates, changeset ent
- README.md: installation, quick start (under 30 seconds to first result), API ove
- Error messages include the function name and expected vs received values.
- Support both ESM and CommonJS consumers. Test both in CI.

Do not:
- Export internal implementation details. Only the public API surface is exported.
- Use Node.js-specific APIs unless the library is Node-only. Prefer Web APIs for c
- Add peer dependencies without major version flexibility (e.g., "react": ">=17" n
- Commit generated files (dist/, docs/api/). These are built in CI.
```

CHAPTER 05

Workflow Patterns

▸ The Describe → Generate → Review → Refine Loop

This is the fundamental workflow. Every vibe coding session follows this pattern, whether you're aware of it or not. Making it explicit makes you faster.

Describe: Be specific about what you want. Bad: "make a login form." Good: "Create a login form component with email and password fields, client-side validation using zod (email format, password min 8 chars), error messages below each field, a submit button that disables during loading, and calls the /api/auth/login endpoint via a TanStack Query mutation."

The more specific your description, the fewer refine cycles you need. Spend 60 seconds writing a thorough description. It'll save you 10 minutes of back-and-forth.

Generate: Let the AI write the first draft. Don't interrupt it. Don't hover over suggestions and accept them one line at a time. Give it the full prompt and let it produce the complete output.

Review: Read the generated code critically. Check for: - Does it actually do what you asked? - Are there obvious bugs or logic errors? - Does it follow your project's patterns? - Are there security concerns? - Is it importing things that don't exist in your project?

Refine: Give targeted feedback. "The validation is correct but move the zod schema to a separate file at lib/validations/auth.ts" is better than "that's not quite right, try again."

Typically you'll do 2-3 refine cycles. If you're past 5, the AI probably doesn't have enough context. Add more files to the conversation or try a different approach.

▸ Architecture-First: Plan Before You Code

The most powerful vibe coding technique isn't about writing code at all. It's about getting the AI to *plan* first.

Before writing any implementation, prompt:

```
I need to build [feature description]. Before writing any code, give me:
1. The file structure — what files will be created or modified
2. The data model — what types/interfaces/schemas are needed
3. The API surface — what endpoints or functions will exist
4. The dependencies — what existing code this will interact with
5. Edge cases — what could go wrong

Do not write any implementation code yet. Just the plan.
```

Review the plan. Adjust it. *Then* ask for implementation. This avoids the painful situation where the AI generates 500 lines of code that's architecturally wrong.

▸ Test-Driven Vibe Coding

This is counterintuitive but devastatingly effective: describe the tests first, then ask the AI to write the implementation that makes them pass.

```
Write tests for a ShoppingCart class with these behaviours:
- Adding an item increases the item count
- Adding the same item twice increases its quantity, doesn't duplicate
- Removing an item decreases the count
- Clearing the cart removes all items
- Total price is calculated from item prices × quantities
- Applying a percentage discount reduces the total
- Cannot apply a negative discount (should throw)

Use Vitest. Write the tests first. Do NOT write the implementation yet.
```

Once you have the tests, review them and then say: "Now write the ShoppingCart implementation that makes all these tests pass."

Why this works: tests are a precise specification. When the AI writes the implementation to match tests, it has an unambiguous definition of "correct." The resulting code is testable by design and the tests actually cover meaningful behaviour.

▸ Debugging with AI: The Right Way to Share Context

When something's broken, resist the urge to paste the error and say "fix this." That produces guesswork. Instead, give structured context:

```
I'm getting an error in my Next.js app.

**What I'm trying to do:** Fetch user data on the profile page using a server comp

**The error:**
TypeError: Cannot read properties of undefined (reading 'id')
  at UserProfile (app/profile/page.tsx:12:34)

**Relevant code:**
[paste the component]

**What I've already tried:**
- Checked that the user session exists (it does, I logged it)
- Verified the API endpoint returns data in Postman

**My suspicion:** I think the async data fetching might not be awaited properly, b
```

This structured format gives the AI enough context to provide a targeted fix rather than a speculative rewrite of your entire component.

▸ Code Review Prompts for AI-Generated Code

After the AI generates code, use these review prompts to catch issues:

```
Review this code for:
1. Security vulnerabilities (injection, XSS, auth bypass)
2. Performance issues (N+1 queries, unnecessary re-renders, memory leaks)
3. Error handling gaps (what happens when the network fails? when data is malforme
4. Edge cases (empty arrays, null values, concurrent access)
5. Consistency with the rest of the codebase
```

You can also ask for specific analyses:

- “What happens if two users submit this form simultaneously?”
- “Could this query be slow with 100,000 rows in the table?”
- “Is there a race condition in this async code?”

Don’t skip review because the code “looks right.” AI-generated code is most dangerous when it’s 95% correct — the bugs hide in the last 5%.

CHAPTER 06

Advanced Techniques

▸ Multi-File Refactoring Strategies

Large refactors are where most people hit the AI's limits. The context window fills up, the AI loses track of the overall goal, and you end up with half-refactored code that doesn't compile.

The solution is **staged refactoring:**

1. **Map phase:** "List every file that imports from lib/old-auth.ts and describe what each one uses from it."
2. **Plan phase:** "Create a migration plan to move from the old auth pattern to the new one. Order the changes so each step leaves the codebase in a working state."
3. **Execute phase:** Do one file at a time. "Now refactor app/api/users/route.ts to use the new auth pattern. Here's the new auth utility for reference: [paste new-auth.ts]."
4. **Verify phase:** "Check that all imports from lib/old-auth.ts have been migrated. List any remaining references."

Never try to refactor everything in one prompt. The AI will lose coherence past 3-4 files.

▸ Getting AI to Follow Your Existing Patterns

The `.cursorrules` file handles general patterns, but for specific situations, show don't tell:

```
Here's how we handle API error responses in this project:

[paste an existing, well-written example]

Now create the same pattern for the /api/orders endpoint.
```

Providing a concrete example from your own codebase is worth more than paragraphs of description. The AI will match the style, naming, and structure almost exactly.

Another technique: **@-reference your best files.** In Cursor, say "Follow the same pattern as @api/users/route.ts" when creating a new endpoint. The AI will read that file and mirror it.

▸ Managing Context Window Limitations

Every AI model has a finite context window. When you hit it, the AI starts “forgetting” the beginning of your conversation. Signs you’ve hit the limit:

- AI contradicts something it said earlier
- Generated code imports from files you discussed but gets the paths wrong
- Suggestions become more generic and less project-specific

Mitigation strategies:

- Start new conversations for new tasks (don’t reuse a 50-message thread)
- Front-load the important context (project structure, key files, constraints)
- Use `.cursorrules` / `CLAUDE.md` — this context is always present, it doesn’t eat into your conversation window
- In Aider, actively `/drop` files you’re done with to free up context space
- Summarise long conversations: “To recap, we’ve built X and Y. Now I need Z.”

▸ When to Stop Vibing and Write It Yourself

Vibe coding is a tool, not a religion. Stop and write the code yourself when:

- **You’re on refine cycle #5** for the same piece of code. You’re spending more time prompt-engineering than coding.
- **The logic is genuinely novel.** If you’re inventing an algorithm, write it. AI can only recombine existing patterns.
- **Performance is critical.** AI-generated code is usually correct but rarely optimal. Hand-tune hot paths.
- **You can’t explain what you want.** If you can’t describe it clearly enough for the AI, you don’t understand the problem well enough yet. Sketch it out manually, *then* bring in the AI.
- **The code is security-critical.** Authentication, authorisation, encryption, payment processing. Review AI suggestions here but write the core logic yourself.

The best vibe coders aren’t the ones who use AI for everything. They’re the ones who know exactly when to use it and when to step in.

CHAPTER 07

Appendix

▸ Quick Reference: Best Prompt Patterns

PATTERN	EXAMPLE
Be specific	“Create a paginated API endpoint” → “Create a GET /api/products endpoint with cursor-based pagination, 20 items per page, sorted by created_at desc, returning a next_cursor field”
Show, don’t tell	“Follow clean code practices” → “Follow the pattern in @api/users/route.ts”
Constrain the output	“Build a form” → “Build a form using react-hook-form and zod. No other form libraries.”
Define the boundary	“Refactor the auth” → “Refactor only lib/auth.ts and its direct imports. Don’t touch the API routes yet.”
Request format	“Help me with tests” → “Write Vitest tests. One describe block per public method. Include edge cases for null/undefined inputs.”
Plan first	“Build the feature” → “Plan the file structure and data model first. Don’t write implementation until I approve the plan.”
Provide context	“Fix this bug” → “This error occurs when [context]. Here’s the relevant code [code]. I’ve tried [attempts]. I suspect [hypothesis].”
Iterate narrowly	“That’s wrong, try again” → “The validation logic is correct but the error messages should use the field labels, not the field names.”

▸ Model Comparison for Coding (Early 2025)

MODEL	SPEED	CODE QUALITY	CONTEXT WINDOW	BEST FOR
Claude Opus 4	Slow	Excellent	200K	Complex architecture, multi-file refactors, nuanced code review
Claude Sonnet 4	Fast	Very good	200K	Daily driver. Best speed/quality ratio for most tasks
GPT-4o	Fast	Very good	128K	Python, data science, well-documented APIs

MODEL	SPEED	CODE QUALITY	CONTEXT WINDOW	BEST FOR
Gemini 2.5 Pro	Medium	Good	1M	Massive context tasks, entire codebase analysis
DeepSeek V3	Fast	Good	128K	Budget option, strong at algorithmic code
Llama 3 (local)	Varies	Decent	128K	Offline work, privacy-sensitive projects
GPT-4o mini	Very fast	Acceptable	128K	Autocomplete, simple generation, cost-sensitive bulk work

My daily setup: Claude Sonnet 4 for 80% of work. Claude Opus 4 when I need deep reasoning. GPT-4o when Sonnet struggles with a specific API or library. Gemini 2.5 Pro when I need to analyse an entire codebase at once.

▸ Resources & Communities

Tools: - cursor.com — AI-first code editor - aider.chat — CLI coding assistant - claude.ai/code — Claude Code CLI - github.com/features/copilot — GitHub Copilot

Communities: - r/ChatGPTCoding — active subreddit for AI-assisted development - Cursor Discord — tool-specific tips and `.cursorrules` sharing - Aider Discord — power user techniques and model comparisons - r/LocalLLaMA — if you want to run models locally for coding

Learning: - docs.cursor.com — official Cursor documentation - Simon Willison's blog (simonwillison.net) — excellent writing on AI-assisted development - Thorsten Ball's blog — practical vibe coding experiences

Prompt libraries: - cursor.directory — community-contributed `.cursorrules` files - GitHub: search "cursorrules" for project-specific examples

---

This guide is maintained and updated as tools and models evolve. You'll receive updates when significant changes warrant them.

Written by someone who vibe-codes daily and has the commit history to prove it.

The Vibe Coding Starter Kit
