Monday Made Easier

VIBE CODING COURSE

Vibe Coding Handbook

Monday Made Easier

A practical course for building real apps with AI, from simple prompts and documents to full-stack products with auth, backend, AI features, deployment, and verification.

CORE METHOD

Scope. Build. Inspect. Verify.

Vibe coding works when you use AI to move faster without giving up responsibility for stack choices, security, data, and production behavior.

FOR PARTICIPANTS

From prompt to product.

This handbook moves from low-risk artifacts to real deployed software: pages, app shells, databases, auth, RLS, AI calls, payments, tests, and reusable workflows.

SMART TOOLS. REAL APPS. CONFIDENT SHIPPING.

Contents

Main Conclusion

Vibe coding is a new way to build software with AI, but it is not magic.

You do not need to start as a developer. You do need a clear mental model of what is being built:

- the idea
- the user
- the screens
- the data
- the backend
- the login and permissions
- the AI features
- the deployment
- the checks that prove it works

The useful promise is:

> **You can build useful software faster with AI when you know how to scope the idea, choose the right tools, guide the build, diagnose problems, and verify the result.**

This handbook keeps the technical ideas, but explains them in plain language.

The Whole Workflow

Most successful AI-assisted builds follow this path:

```
Idea -> conversation -> structure -> stack choice -> first screen -> data model
-> backend -> auth -> AI feature -> test -> deploy -> improve
```

In practice, that means:

Step	What you do	What Codex or another tool helps with
Idea	Explain the problem and audience	Turn vague notes into a product brief

Step	What you do	What Codex or another tool helps with
Conversation	Ask questions before building	Surface assumptions and missing details
Structure	Decide screens, data, and user paths	Draft app maps, schemas, and build plans
Stack choice	Pick frontend, backend, auth, database, deploy path	Compare options and trade-offs
First screen	Build one visible page	Create files, components, and layout
Data model	Decide what the app remembers	Draft tables, fields, and relationships
Backend	Save and fetch real data	Write server code and validation
Auth	Add login and permissions	Wire sessions, protected routes, policies
AI feature	Add one useful AI action	Call the model server-side with grounded data
Test	Check the real user path	Run tests, browser checks, and diagnostics
Deploy	Put it online safely	Configure env vars, build, and smoke test
Improve	Repeat in small steps	Refactor, add features, and save workflows

Move through this path calmly. Do not jump from idea to "build the whole app."

What Vibe Coding Is

Vibe coding means building software by explaining what you want in natural language while an AI coding tool creates and edits the underlying code.

You are still responsible for the product decisions.

AI can help with:

- product briefs
- screen plans
- data models

- code generation
- debugging
- refactoring
- tests
- documentation
- deployment checks

AI should not be trusted blindly with:

- security decisions
- payment logic
- private data
- production deployment
- customer-facing messages
- changes you cannot explain

The simple rule:

> **Let AI move fast, but keep ownership of the decisions and the checks.**

The Minimum Mental Model

Before you choose tools, you need to know the basic parts of an app.

App part	Plain meaning	Beginner question
Frontend	What the user sees and clicks	What screens does the user need?
Backend	The hidden logic behind the app	What should happen when the user does something?
Database	Where the app remembers information	What needs to be saved?
Auth	How the app knows who the user is	Who is signed in?
Authorization	What that user can access	What are they allowed to see or change?
API	A controlled way for systems to talk	What outside service is needed?

App part	Plain meaning	Beginner question
Deployment	Putting the app online	Can someone else open and use it?
Environment variables	Private settings and keys	Are secrets kept out of the browser?
Tests	Checks that prove the app works	What evidence shows the path works?

Part 1: Start With The App Shape

The first decision is not the tool. The first decision is the shape of the app.

Ask:

```
What kind of app is this?
Who will use it?
What information does it need to remember?
Does it need login?
Does it need payments?
Does it need realtime updates?
Does it use private or sensitive data?
Who will maintain it after the first version?
```

Common app shapes:

App shape	Examples	Usually needs
Landing page	offer page, waitlist, service page	frontend, form, analytics
Internal tool	dashboard, tracker, workflow helper	frontend, database, auth
CRUD app	client tracker, task app, inventory	forms, tables, backend, database
SaaS	paid product, team workspace	auth, billing, roles, deployment
Marketplace	buyers, sellers, listings	auth, payments, permissions
AI wrapper	summarize, draft, classify, search	server-side model calls, data rules
Realtime app	chat, collaboration, live dashboard	realtime backend, permissions
Mobile-first app	consumer app, field app	mobile stack, auth, offline considerations

The better the app shape, the better the tool choice.

Part 2: Tool Map For Vibe Coding

There is no single best tool. There are categories.

Tool category	Examples	Best for	Watch-out
AI workbench	Codex, Claude Code	Projects with files, tools, browser checks, Git, plugins, and repeatable workflows	Needs clear context and verification
AI code editor	Cursor, IDE extensions	Editing a real codebase with AI help in the editor	Easy to accept code without understanding it
AI app builder	Lovable, Bolt, Replit-style tools	Fast prototypes and UI-to-app experiments	Security, export, and ownership must be checked
UI generator	v0-style tools, screenshot-to-code workflows	Exploring screens and components quickly	Can create pretty UI before the data model exists
Frontend framework	Next.js, React frameworks	Real web apps with pages, components, and routing	You need to understand client/server boundaries
Backend platform	Supabase, Convex, Firebase, Neon/Postgres	Auth, data, storage, realtime, server logic	Each has a different security model
Deployment platform	Vercel, Netlify, builder-native hosting	Sharing and testing online	Local success does not prove production success

Part 3: Default Stack

Recommended default:

Layer	Default choice	Why this helps
AI workbench	Codex or Claude Code	Can work with files, run checks, inspect browser previews, and manage the project
Framework	Next.js App Router with TypeScript	Common full-stack React path with server and frontend in one project
UI	Tailwind CSS and shadcn/ui	Fast, inspectable, component-based UI
Backend	Next.js server code plus Supabase	Shows server boundaries without a separate custom API
Database	Supabase Postgres	Real relational data with visible tables and SQL
Auth	Supabase Auth first; Clerk as a serious alternative	Supabase keeps the V1 stack simpler; Clerk is strong for teams and organizations
Authorization	RLS and workspace membership checks	Shows why login alone is not security
AI feature	Server-side OpenAI API call	Shows API keys, privacy, and reviewable outputs
Deployment	Vercel	Natural Next.js deployment path with preview deployments
Testing	Smoke tests plus browser checks	Makes verification part of the workflow

This is not the only good stack. It is a good default because it keeps the number of moving pieces manageable while still showing real app architecture.

Part 4: Example App

The example app is **LaunchPad CRM**, a lightweight client and project tracker for a small service business.

This example shows the important app layers without needing an unusual idea.

V1 features:

- sign up, sign in, and sign out
- create a workspace
- add clients
- add projects for clients
- add tasks and notes
- view a simple dashboard
- generate an AI project summary
- protect workspace data
- deploy a working version

Why this app works as the example:

- the business idea is easy to understand
- the data model is real but not huge
- auth and permissions matter
- AI can add a useful summary feature
- it can be tested from end to end

Avoid adding payments, team invites, file uploads, and complex automation until the basic user path works.

Part 5: The Product Brief

Before building, talk to the model and shape the idea.

Prompt:

```
Help me turn this app idea into a compact product brief.

App idea:
[plain-English idea]

Ask me up to 10 useful questions first.

Then produce:
- target user
- problem
- first version
- not included yet
- core screens
- data the app must remember
- likely stack
- biggest risks
- first build step
```

Keep questioning until the structure is clear.

Good follow-up questions:

- What must the user do first?
- What data is private?
- What happens if two users belong to different teams?
- What can be fake in the prototype?
- What must use real data from the start?
- What would make this unsafe to ship?

Part 6: Frontend Choices

The frontend is the app surface: screens, buttons, forms, navigation, tables, charts, and empty states.

Recommended V1 frontend:

Choice	Why
Next.js App Router	Good default for modern web apps and deployment
TypeScript	Helps catch mistakes earlier
Tailwind CSS	Fast layout and styling
shadcn/ui	Clean, practical components that AI tools can work with

Use this checklist to inspect frontend output:

- Does the screen match the brief?
- Is the main action obvious?
- Does it work on mobile?
- Are there fake numbers or placeholder data?
- Are forms clear and usable?
- Does the design feel like the app's purpose?

Common failure:

> **The UI looks finished, but nothing real is connected.**

Correction:

Ask Codex to label which parts are real, mocked, or placeholder.

Part 7: Backend And Database Choices

The backend is where the app does hidden work: saving data, checking permissions, calling APIs, and handling private logic.

The database is where the app remembers things.

Backend options:

Option	Best for	Watch-out
Next.js Server Actions / Route Handlers	Forms, CRUD, AI calls, webhooks	Client/server boundaries can blur
Supabase	Relational apps, dashboards, auth plus RLS, storage	RLS must be designed and tested
Convex	Realtime collaboration, reactive dashboards, live state	Functions, auth integration, and permissions must be planned
Firebase	Mobile-first apps, Google ecosystem, document/realtime data	Security rules and data shape need care
Neon/Postgres with ORM	Conventional SQL apps and portable backend patterns	More setup decisions for auth and migrations
Builder-native backend	Fast prototypes	Export, ownership, and production readiness must be checked

Database choices:

Data choice	Best fit	Watch-out
Supabase Postgres	CRM, dashboards, SaaS, relational data	Requires understanding relationships and policies
Convex	Live dashboards and collaborative state	Not a SQL-first mental model
Firebase Firestore	Mobile-first and document-shaped apps	Data is often duplicated and security rules matter
Neon/Postgres	SQL apps with separate auth/backend choices	More pieces to wire

Data choice	Best fit	Watch-out
Builder-native data	Early experiments	Check data export and long-term ownership

Plain rule:

If the app has customers, projects, tasks, invoices, bookings, or records that relate to each other, start with relational data.

Part 8: Data Model Without Losing People

A data model is the map of what the app remembers.

For LaunchPad CRM:

Thing	What it means	Relationship
User	person signed in	belongs to one or more workspaces
Workspace	business or team area	has members, clients, projects
Client	customer account	belongs to a workspace
Project	work for a client	belongs to a client and workspace
Task	thing to do	belongs to a project
Note	useful context	belongs to a project
AI summary	generated project summary	belongs to a project

Compact schema direction:

```
users
workspaces
workspace_members
clients
projects
tasks
notes
ai_summaries
```

Ask before building:

```
Review this app idea and propose a simple data model.

For each table or collection, show:
- what it stores
- important fields
- what it belongs to
```

```
- who can read it
- who can create or update it
- what can go wrong if permissions are wrong
```

You do not need to become a database expert. You do need to understand what the app saves and who can access it.

Part 9: Auth And Authorization

Authentication answers:

```
Who is signed in?
```

Authorization answers:

```
What is this signed-in person allowed to see or change?
```

Critical lesson:

> **A login page is not security. Security exists when routes, server code, and database rules all agree.**

Minimum contract:

```
If a user is not signed in:
- they cannot access private pages
- they cannot read private records
- they cannot write private records

If a user is signed in:
- they can only access records that belong to their workspace
- writes include the correct workspace/user ownership
- unauthorized attempts fail clearly
```

V1 roles:

Role	Can do
Owner	manage workspace and records
Member	read and update workspace records

Do not add complex roles until the product needs them.

Two-user test:

```
Create User A and Workspace A.
Create User B and Workspace B.
Add clients to both.
Confirm User A cannot see or edit User B's clients.
Confirm User B cannot see or edit User A's clients.
```

Part 10: AI Features Inside Apps

The first AI feature should be boring, useful, and easy to review.

Recommended V1 feature:

> **Generate a project summary from that project's notes and tasks.**

Why it works:

- it uses data already inside the app
- the output can be reviewed
- mistakes are visible
- the user remains in control

Minimum AI contract:

Contract part	What it means
Input	project name, client name, notes, open tasks
Output	short summary, risks, next actions
Rules	use only provided data, mark uncertainty, keep API keys server-side, let the user review

Avoid for V1:

- autonomous sending
- hidden AI decisions
- AI over private data without a privacy decision
- chat over the whole database before permissions are clear

Part 11: AI Reliability Checklist

AI features need a little more discipline than normal app screens because the output can vary.

Before adding AI, ask:

```
Does this feature need AI, or would rules, filters, templates, or search be enough?
```

Use AI when the task involves language, judgment, summarizing, classifying, extracting, drafting, or reasoning over messy context.

Do not use AI just to look impressive.

The Four Checks

Check	Plain meaning	Example
Grounding	What information is the model allowed to use?	Only this project's notes and tasks
Evaluation	How do we know the answer is good?	Compare 10 sample summaries against a checklist
Cost and speed	Is the feature fast and affordable enough?	Limit input, choose the right model, avoid sending the whole database
Safety	What can go wrong if the model follows the wrong instruction?	Prompt injection, data leaks, invented facts, accidental sending

Evaluation For Beginners

Do not evaluate an AI feature by trying it once.

Create a small test set:

- 5 normal examples
- 2 messy examples
- 2 missing-context examples
- 1 example where the model should refuse or ask for more information

Then score:

- Did it use only the provided context?
- Did it invent anything?
- Did it follow the requested format?
- Was it useful?
- Did it mark uncertainty?
- Would a human feel safe using it?

The point is not academic grading. The point is to catch common failures before users do.

Context, RAG, And Agents

Understand three levels:

Pattern	What it means	Use when
Direct prompt	Put the needed context directly in the request	The context is small and already available
RAG	Retrieve the most relevant documents or records, then ask the model to answer from them	The app has lots of notes, docs, pages, or knowledge
Agent	Let the model take steps with tools, such as search, inspect, update, or call APIs	The workflow needs multiple actions and can be checked safely

Simple rule:

> **Start with direct context. Add RAG when there is too much context to paste. Add agents only after the normal workflow is stable and reviewable.**

AI Feature Prompt

Ask Codex to decide whether AI is needed, define the context and output format, create 10 test cases, list failure cases, identify cost, latency, privacy, and prompt-injection risks, and explain how the user reviews or corrects the output.

Feedback Loop

After launch, save examples of good outputs, bad outputs, user corrections, missing context, slow or expensive requests, and confusing responses.

Use those examples to improve prompts, evaluation checks, retrieval rules, and product design.

Part 12: Payments, Email, Files, And Analytics

These are useful, but they should come after the core path works.

Feature	Add when	Watch-out
Stripe	The app has a real paid plan	Do not trust the browser to mark payment active
Email	Invites, receipts, digests, or notifications are needed	Review before sending customer-facing AI text
File uploads	Users need documents, images, or attachments	Storage permissions matter
Analytics	You need usage and funnel insight	Do not log private data
Error tracking	Other people are using the app	Errors need enough context to diagnose
Automations	Workflow is stable and repeatable	Do not automate unclear or high-risk actions

Simple rule:

Build the path first. Add integrations after the path is real.

Part 13: Diagnosis Workflow

Vibe coding gets much stronger when you stop saying "try again" and start diagnosing.

Use this prompt:

```
We have this issue:
[describe what happened]

Expected:
[what should have happened]

Actual:
[what happened instead]

Do not guess.
Trace the user path.
Identify the boundary where it fails.
List the evidence.
Make the smallest fix for the confirmed root cause.
Then tell me exactly how to retest it.
```

Common boundaries:

Boundary	What can fail
Browser to UI	button, form, layout, state
UI to server	request shape, validation, missing input
Server to database	wrong query, missing user, bad write
Database to policy	RLS or security rules block the request
Server to AI API	key missing, data too large, bad output shape
Local to production	env vars, redirects, build settings
Payment provider to app	webhook, signature, status mapping

Habit:

Find the failing boundary before changing code.

Part 14: Verification Standard

Every real feature needs evidence.

Minimum verification:

- happy path works
- one likely failure path is checked
- real data is used where it matters
- secrets are not exposed
- auth and permissions still work
- AI outputs are grounded, useful, and reviewable
- local and deployed behavior are understood
- you can explain what changed

Verification prompt:

```
Verify this feature.

Check:
- happy path
- likely failure paths
- auth and permission behavior
- data persistence
- AI output quality if the feature uses AI
- hallucination or unsupported-claim risk
- cost, latency, and privacy risk
- secrets exposure
- production-specific configuration

Report:
- what passed
- what failed
- what was not checked
- what evidence supports the result
```

Do not mark a feature done because the screen looks good.

Part 15: Safe Launch Path

A safe first version is small, tested, and honest about limits.

Before sharing:

- deploy the app
- set environment variables correctly
- run the main user path
- test sign up, sign in, sign out
- test data persistence
- test two-user privacy
- test the AI feature with real sample data
- check AI output against the evaluation examples
- check logs for obvious errors
- write known limitations

First-share prompt:

```
Prepare this app for a safe first share.

Check:
- build status
- env vars
- login flow
- main user path
- database writes
- permission rules
- AI feature behavior
- AI evaluation examples
- cost, latency, and privacy risks
- placeholder content
- known limitations

Do not call it production-ready unless the deployed user path has been tested.
```

Part 16: Recommended Build Sequence

Use this sequence for the workbook or your own build.

Step	Build	Done when
1	Product brief	app shape, user, V1 scope, risks are clear
2	Screen map	dashboard, clients, projects, settings are planned
3	App shell	navigation and basic layout exist
4	Static screens	main pages look clear with placeholder data
5	Data model	tables and relationships are drafted
6	Real client records	create/read clients with database data
7	Projects and tasks	records connect correctly
8	Auth	users can sign in and out
9	Authorization	users cannot access the wrong workspace
10	AI summary	server-side AI feature works with review
11	AI evaluation	sample cases prove the feature is useful and grounded
12	Deployment	live app opens and main path works
13	Improvement loop	record next fixes and reusable prompts

This keeps complexity visible and controlled.

Part 17: Decision Cheat Sheet

If choosing quickly:

Need	Start with
Simple marketing page	AI builder or Next.js static page
Internal dashboard	Next.js + Supabase
Client/project tracker	Next.js + Supabase Auth + Postgres + RLS
Team SaaS	Next.js + Clerk + Postgres or Convex
Realtime collaboration	Convex or Firebase-style realtime backend
Mobile-first app	Firebase or mobile-native stack
Conventional SQL product	Next.js + Neon/Postgres + ORM + separate auth
Fast visual prototype	Lovable, Bolt, Replit-style builder, or v0-style UI generation
Production app with private data	Owned repo, explicit auth, tested permissions, deployment checks

If unsure, ask Codex:

```
Compare 3 stack options for this app.

App:
[brief]

For each option, show:
- frontend
- backend
- database
- auth
- deployment
- best fit
- biggest risk
- what I must understand before using it
```

```
Recommend one default path for V1.
```

Part 18: Common Mistakes

Mistake 1: Asking For The Whole App At Once

Correction:

Build one small slice, check it, then continue.

Mistake 2: Choosing Tools Before Understanding The App

Correction:

Define the app shape, data, auth, risk, and maintenance needs first.

Mistake 3: Treating Pretty UI As A Working App

Correction:

Check whether data is real, saved, protected, and visible after refresh.

Mistake 4: Treating Login As Security

Correction:

Test authorization with two users and two workspaces.

Mistake 5: Letting AI Pick Every Architecture Decision

Correction:

Ask for options, trade-offs, and a recommendation. Then decide.

Mistake 6: Adding AI Before The Normal Workflow Works

Correction:

Make the app useful without AI first, then add one reviewable AI feature.

Mistake 7: Trusting AI Output After One Try

Correction:

Use sample cases, expected outputs, and review rules before relying on the AI feature.

Mistake 8: Debugging By Repeating The Same Prompt

Correction:

Ask for diagnosis, evidence, failing boundary, smallest fix, and retest steps.

Mistake 9: Shipping Local Success

Correction:

Deploy and test the real user path in the deployed environment.

Part 19: Glossary

Agent:

An AI system that can use tools and take steps toward a goal.

Vibe coding:

Building software with AI by describing goals, reviewing changes, and steering the result.

Frontend:

The screens and interactions the user sees.

Backend:

The hidden logic that saves data, checks permissions, and talks to services.

Database:

Where the app stores information it needs to remember.

Authentication:

Proving who the user is.

Authorization:

Deciding what the user is allowed to see or change.

RLS:

Row-level security. Database rules that control which records a user can read or change.

API key:

A private key that lets the app use another service. It should stay server-side.

Webhook:

A server endpoint that receives events from another service, such as Stripe.

Environment variable:

A private setting used by the app, often for keys, URLs, or production configuration.

Smoke test:

A short end-to-end check that proves the most important path works.

RAG:

Retrieval-augmented generation. The app finds relevant context first, then gives that context to the model.

Evaluation:

A repeatable way to check whether an AI feature is useful, grounded, safe, and consistent enough.

Prompt injection:

When malicious or accidental text tries to override the app's real instructions.

Latency:

How long the user waits for the feature to respond.

Closing Message

The goal is not to turn you into a traditional software engineer.

The goal is to give you enough mental model to build responsibly with AI:

- know what kind of app they are making
- choose tools for the app shape
- understand frontend, backend, data, auth, and deployment
- guide Codex with context and constraints
- diagnose before changing things
- verify before sharing
- improve in small steps

Good vibe coding is not just prompting. It is product thinking, technical judgment, and verification made easier to practice with AI.