

Best Practices for Vibecoding

A practical guide to building reliably with AI tools like Lovable, Replit and Claude

The core idea: Vibecoding works best when you stay the product owner and architect. Describe outcomes clearly, work in small steps, verify everything, and let the AI handle the typing.

1. Plan Before You Prompt

Most failed builds start with a vague first prompt. Spend five minutes defining the project before generating anything.

- **Write a one-page spec:** audience, core problem, 3-5 must-have features, and what is explicitly out of scope.
- **Define the data model early:** list the entities (users, posts, orders) and how they relate. Changing the schema later is the most expensive edit.
- **Pick a boring stack:** use the platform defaults (React, Tailwind, Supabase). Popular stacks are where AI is most accurate.
- **Set project knowledge:** store brand colors, tone, and rules in Lovable's project knowledge or Replit's replit.md so every prompt starts with context.

2. Prompt Like a Product Manager

Specific prompts produce specific results. Describe behavior and outcome, not just features.

Weak prompt	Strong prompt
Add a contact form.	Add a contact form with name, email, message. Validate email, show inline errors, save to the <i>leads</i> table, and show a success toast. Match the existing button style.
Make it look better.	Increase section spacing to 80px, use a 12-column grid, limit body text to 65 characters per line, and make cards consistent with rounded-xl and soft shadows.
Fix the bug.	On mobile, the menu does not close after tapping a link. Expected: it closes and scrolls to the section. Only change the nav component.

- **One change per prompt.** Bundled requests cause regressions and make problems hard to trace.
- **State constraints:** "Do not modify other files" or "keep the existing layout" prevents unwanted rewrites.
- **Give examples and references:** screenshots, sample copy, or a competitor URL beat adjectives like "modern" or "clean".

3. Build in Small, Verified Steps

- **Start with the skeleton:** layout, navigation, and routing first. Add data, auth, and integrations after.
- **Test after every change.** Click through the feature on desktop and mobile before moving on.
- **Use version history.** Treat each working state as a checkpoint you can restore. Never stack ten edits on a broken build.
- **Use plan or chat mode** to discuss approach before the AI edits code, especially for larger changes.
- **Stop the loop early.** If the AI fails to fix the same issue three times, revert, rewrite the prompt with more detail, or try a different approach.

4. Debug Effectively

- Paste the **exact error message** and console output, not a paraphrase.
- Describe what you **expected** versus what **happened**, and the steps to reproduce.
- Ask the AI to **explain the cause first**, then fix. This surfaces wrong assumptions before code changes.
- Add logging when behavior is unclear, then share the logs back.
- For stubborn problems, start a fresh chat with a short summary. Long threads accumulate stale context.

5. Security and Data Safety

AI-generated apps often work but are not secure by default. Review these before launch:

- **Never put secrets in front-end code.** API keys belong in environment variables or server-side functions.
- **Enable row-level security** on every database table and test with a non-admin account.
- **Validate input on the server**, not only in the browser.
- **Check authentication paths:** can a logged-out user reach protected pages or data?
- **Keep production and testing separate**, and never test destructive actions on live data.
- **Review dependencies and permissions** the AI adds, and remove what you do not need.

Rule of thumb: if the app handles payments, personal data, or health information, have an experienced developer review it before launch.

6. Keep Quality High

- **Design system first:** define colors, fonts, spacing, and components once, then reuse them so pages stay consistent.
- **Check performance:** compress images, lazy-load media, and run Lighthouse before launch.
- **Build for SEO and accessibility:** proper headings, alt text, meta tags, readable contrast, and keyboard navigation.
- **Handle states:** ask for loading, empty, and error states on every data-driven screen.
- **Refactor periodically:** ask the AI to remove duplicate code and unused components once features stabilize.

7. Use Git and Back Up Your Work

- Connect the project to **GitHub** from day one so you own the code and have a history outside the platform.
- Use branches for risky experiments; merge only after testing.
- Export database schemas and keep documentation of integrations and environment variables.

8. Pick the Right Tool for the Job

Tool	Best for
Lovable	Polished front-end sites and apps with Supabase backends, fast UI iteration
Replit	Full-stack apps, scripts, automations, and anything needing a running server
Claude	Spec writing, prompt drafting, debugging, code review, and complex logic
WordPress / Shopify	Content-heavy or commerce sites where the ecosystem beats custom code

Tip: draft specs and prompts in Claude, build in Lovable or Replit, and bring errors back to Claude for diagnosis.

9. Pre-Launch Checklist

- Core user flows tested on desktop and mobile
- Forms, emails, and integrations send and receive correctly
- Auth, permissions, and row-level security verified
- No secrets or test data in the code or database
- Custom domain, SSL, analytics, and metadata configured
- Backups and GitHub sync confirmed
- Error monitoring or a feedback channel in place