Module 11: Vibe Coding for Product Managers

Use AI coding tools to turn ideas into testable prototypes — safely — and know exactly where a prototype ends and production engineering begins.

AUTHOR

Varun Kumar

Product Management Master Handbook
v0.1 · 23 October 2026
Built from the 55-lecture course study guide and extended to the 12-module Product Core curriculum.
Hypothetical examples are labelled; no real company results are invented.

Contents

11.1 What vibe coding means 3
11.2 PM use cases 4
11.3 AI coding tools — categories 5
11.4 Prompt-to-prototype workflow 7
11.5 Git and GitHub fundamentals 8
11.6 HTML, CSS and JavaScript fundamentals 9
11.7 Building basic web applications, API integrations and databases 10
11.8 Deployment fundamentals 11
11.9 Debugging and testing AI-generated code 12
11.10 Security risks of generated code 13
11.11 Prototyping versus production development — and when engineering review is necessary 14
11.12 Building and validating MVPs with vibe coding 14
11.13 Hands-on project: from idea to deployed prototype 15
11.14 Exercises with model answers 17
11.15 Interview perspective 19
11.16 Key takeaways 19
11.17 Check your understanding 20
11.18 Recommended resources 21

MODULE 11

Vibe Coding for Product Managers

Use AI coding tools to turn ideas into testable prototypes — safely — and know exactly where a prototype ends and production engineering begins.

≈ 10 h study

Tool details change fast

Features, pricing, credits and plan limits of AI coding tools change frequently. This module describes **categories and workflows** and links to official docs. Verify current terms in your own account before relying on any capability.

11.1 What vibe coding means

Vibe coding is building software by describing what you want in everyday language and letting an AI coding tool write and change the code, iterating by testing the result and giving feedback. The term was popularised by Andrej Karpathy in early 2025.

From your course notes (Lecture 42)

Vibe coding uses everyday-language instructions with an AI coding tool to create or change software. **The output can be useful but may contain mistakes or security problems.** Ask for a small app, try adding and removing data, then reload to see whether data was *actually saved*. Review generated code, permissions and dependencies; test real flows; **do not publish private data or assume a generated screen is production ready.** *The lecture's rigid distinction between "agentic AI" and "AI agents," and its claim that app builders do not use models, are misleading — categories overlap.* Signing in with GitHub does not itself guarantee repository synchronisation.

Level 1 · Beginner

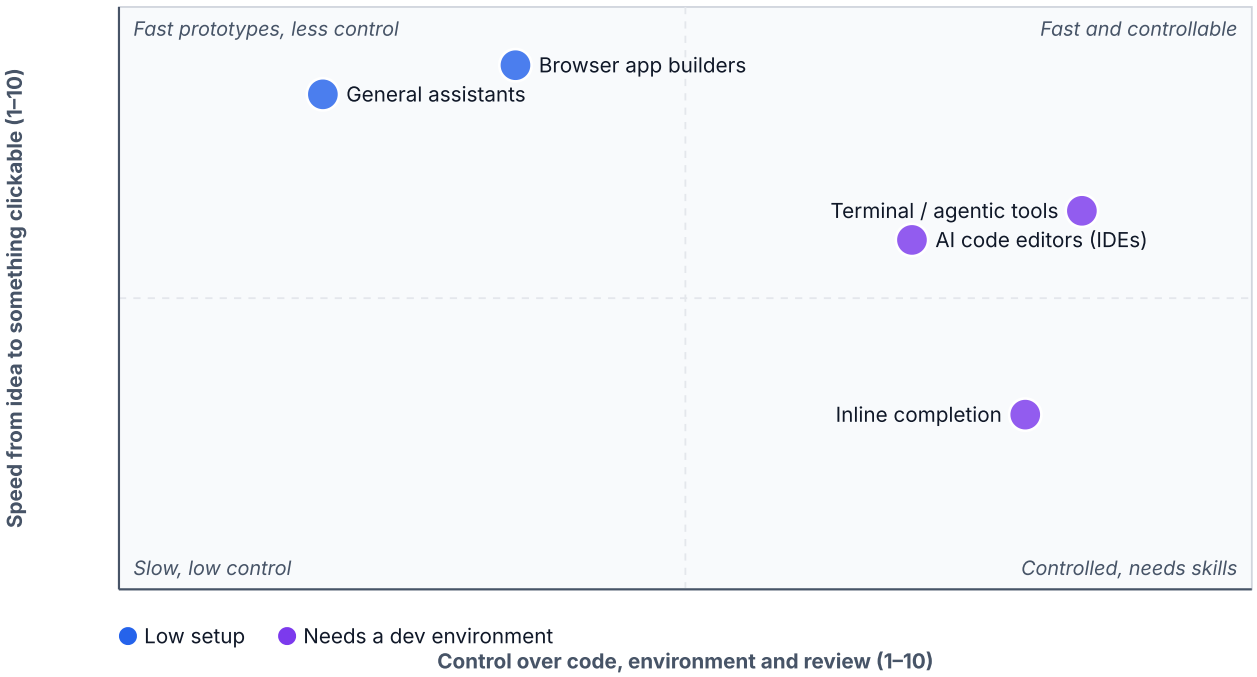
Think of the AI as a very fast junior developer who never gets tired, sometimes misunderstands, and occasionally invents things. You are the product owner and tester: you describe, check, and decide.

11.2 PM use cases

Use case	Why it helps	Watch out for
Clickable/working prototypes for user tests	Test value and usability in days, not weeks	Users may think it's real — label it
Internal tools (dashboards, calculators)	Remove manual work	Data access and security
Data exploration scripts	Answer questions without waiting	Wrong joins, unverified numbers
Demos for stakeholders	Make an idea tangible	Demo ≠ feasible at scale
Learning technical concepts	Build to understand APIs, DBs	Don't ship learning code
Automating workflows	Save time on repetitive tasks	Unapproved access to company systems

11.3 AI coding tools — categories

Category	Examples	Best for	PM-level caution
Browser app builders	Lovable, Bolt (bolt.new), Replit Agent	Prompt-to-app prototypes with hosting	Generated backends/auth need review; lock-in; costs
AI code editors (IDEs)	Cursor, VS Code + GitHub Copilot, Windsurf	Working in a real codebase with AI help	Need basic Git and code-reading skills
Terminal / agentic coding tools	Claude Code, OpenAI Codex CLI, GitHub Copilot agent mode	Multi-file changes, running tests, explaining code	Review every command and diff; permissions
Inline completion	GitHub Copilot suggestions	Faster typing for developers	Suggestions can be subtly wrong
General assistants	Claude, ChatGPT, Gemini (with artifacts/canvas features)	Small single-file prototypes, explanations	Move real work to a proper repo/environment

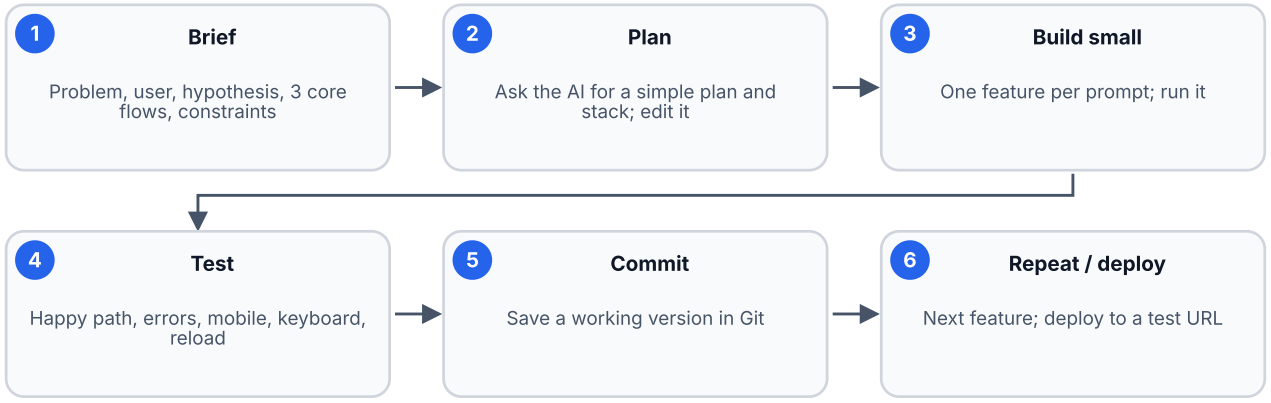


Where AI coding tool categories sit (conceptual). Conceptual placement by category, not a benchmark of specific products. PMs often start top-left and move right as a prototype becomes something engineers will own.

From your course notes (Companion ch. 20–23, 26)

Claude: request a small responsive prototype with dummy data; specify pages, flows, accessibility, error states and *no real payments*; preview it; then move to an approved environment for data, authentication, security review and deployment. **ChatGPT / Gemini:** use as tutor and drafter; verify facts; check account and organisational policies before uploading files. **Lovable:** a generated application is *not automatically secure, correct, accessible or production ready*; iterate one change at a time; map data ownership, authentication and access rules before connecting a backend; use test accounts and mock payments; review server-side authorisation and secret handling with an engineer. **Replit:** separate development from production; store secrets in approved secret management; review dependencies; set up monitoring and backups; confirm ownership, hosting costs and rollback before publishing. *Never paste real customer data into an unreviewed prototype.*

11.4 Prompt-to-prototype workflow



Prompt-to-prototype loop. Small steps with a working, committed version after each one is the single most important habit — it makes mistakes cheap to undo.

From your course notes (Lectures 43, 48, 49)

Choose a familiar product or idea, identify a few core features, **build them in small increments and test each step.** *A screen containing a form is not necessarily connected to persistent storage* — verify the backend, permissions and data flow; a code repository is not automatically the application's live database. Tell the assistant your experience level and preferred technologies, ask for a simple plan, and refine from working pieces instead of accepting unnecessary complexity. When something fails, **report the actual symptoms**, inspect server and database setup, and ask the assistant to investigate.

A good build prompt (template)

```
Context: I'm a PM testing a hypothesis: [one sentence]. Users: [who].
Build: a single-page web app using plain HTML, CSS and JavaScript (no frameworks, no
Core flows: 1) … 2) … 3) …
Data: fictional sample data only; store state in localStorage.
Must handle: empty state, loading state, an error state where [service] fails (show
Accessibility: labels for inputs, keyboard operable, visible focus, buttons ≥ 44px t
Constraints: no external API keys, no real payments, no personal data. Mark the page
Output: index.html, style.css, app.js. Then explain how to run it and list what is s
```

11.5 Git and GitHub fundamentals

> **From your course notes (Companion ch. 24; Lectures 49, 50)**
>
> A **repository** is a project folder with version history; a **branch** isolates work; a **commit** records a snapshot; a **pull request** proposes changes for discussion and review; a **merge** incorporates approved changes. *A merged PR is not necessarily deployed or validated in production.* Check repository visibility and **remove secrets before pushing**; if a secret is exposed, rotate it — deleting it from the latest file doesn't remove it from history. Create an appropriately private repository, review contents, push only when ready; **don't move employer code or data into personal accounts.**

```
git init
```

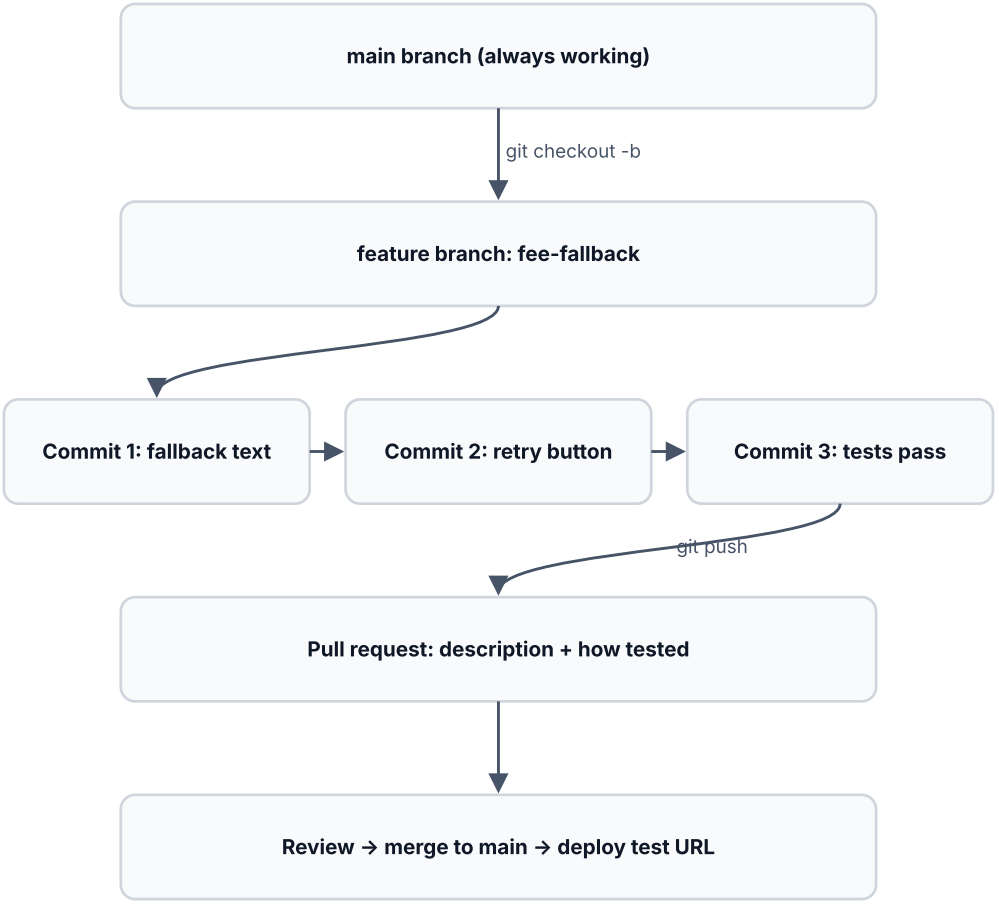
```
git add index.html style.css app.js README.md
```

```
git commit -m "Fee-Clear Cart: first working prototype"
```

```
git checkout -b fee-fallback
```

```
git push -u origin fee-fallback
```

Concept	What to remember
`.gitignore`	List files never to commit (`.env`, `node_modules/`, data exports)
Commit messages	Say *why*, not just *what*
Branch + PR	Even solo, a PR gives a reviewable diff and history
Revert	`git revert <commit>` undoes a change safely by adding a new commit
Secrets	Use environment variables / secret managers; never in code or screenshots



A safe Git workflow for a prototype. Each commit is a save point you can return to. Main stays deployable; experiments live on branches.

11.6 HTML, CSS and JavaScript fundamentals

- **HTML** gives structure and meaning: headings, lists, buttons, forms. Semantic HTML (`<button>`, `<label>`, `<main>`) is also what makes pages accessible.
- **CSS** controls presentation: layout (flexbox, grid), colour, spacing, responsive rules (`@media`).
- **JavaScript** adds behaviour: respond to clicks, update the page, call APIs, store data.

```
<label>Delivery area
  <select id="area"><option value="near">Within 3 km</option></select>
</label>
<button id="pay" type="button" disabled>Pay</button>
```

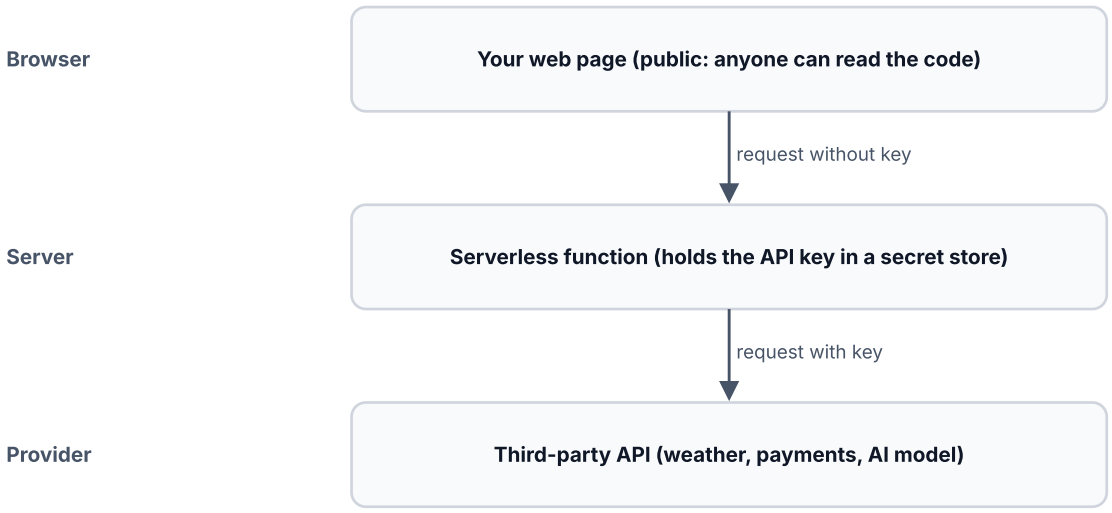
```
.pay { width: 100%; min-height: 44px; background: #2563EB; color: #fff; }
@media (max-width: 480px) { .wrap { padding: 12px; } }
```

```
document.getElementById("pay").addEventListener("click", () => {
  document.getElementById("result").textContent = "Simulated order placed (no paymen
});
```

11.7 Building basic web applications, API integrations and databases

Calling an API from JavaScript (illustrative; a real API needs its own URL, authentication and terms):

```
async function getForecast(city) {
  const res = await fetch(`https://api.example.com/forecast?city=${encodeURIComponen
  if (!res.ok) throw new Error(`HTTP ${res.status}`);   // handle 4xx/5xx explicitly
  return res.json();
}
```



Where secrets may and may not live. The browser never sees the key. If an AI tool puts a key in app.js, move the call server-side and rotate the key.

Key rules: **never put secret API keys in frontend code** (anyone can read them) — call such APIs from a server or serverless function; handle failures, timeouts and rate limits; respect API terms.

Data storage ladder for prototypes:

Option	Good for	Limits
Hard-coded sample data	Pure UX tests	Nothing persists
`localStorage`	Single-user demo persistence in one browser	Not shared, not secure, can be cleared
Spreadsheet/Airtable via an automation	Small shared datasets	Rate limits, access control
Managed backend (e.g., Supabase/Firebase)	Real multi-user prototypes with auth	**Configure access rules (e.g., row-level security) before real data**; costs
Your company's systems	Production	Requires engineering, security, compliance

> **From your course notes (Lectures 40, 43, 49)**
>
> Supabase is a broader platform with PostgreSQL and optional vector capabilities. **Database selection depends on actual workloads;** a vector database is not a replacement for a transactional database. A rental-management prototype used Python/FastAPI, HTML/CSS and SQLite; external payment integration was a possible extension, *not demonstrated as complete.* Use payment-provider **test environments** for prototypes; never handle real card details without safeguards; never store passwords in plaintext.

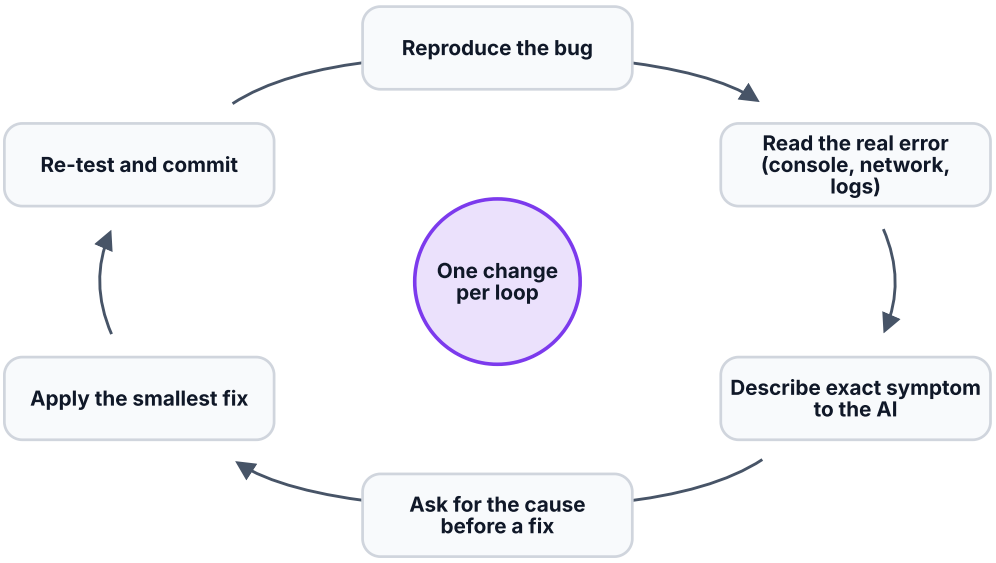
11.8 Deployment fundamentals

- **Static hosting** (no server code): GitHub Pages, Netlify, Vercel, Cloudflare Pages — upload HTML/CSS/JS, get a URL.
- **App hosting** (server code + database): Replit deployments, Render, Railway, Vercel/Netlify functions, cloud providers.
- **Environments:** keep a *test* URL separate from anything real users rely on.
- **Before sharing a link:** remove secrets, check visibility, add a “prototype” banner, confirm data handling, and know how to take it down.

From your course notes (Lecture 42; Lecture 35)

The demo connected a project to a Git repository and published a test URL — **verify the actual integration and visibility settings, then test the real user journey.** *A public prototype link alone does not establish exclusive ownership or access control.*

11.9 Debugging and testing AI-generated code



Debugging loop with an AI coding assistant. The most common failure is skipping step 2 and asking the AI to "fix it" without evidence — it then guesses and changes unrelated code.

Debugging loop: reproduce → read the actual error (browser **Console**, **Network** tab, server logs) → tell the AI the exact symptom and error text → ask it to explain the cause before changing code → apply the smallest fix → re-test → commit.

Testing generated code (as a PM you can always do the first three):

1. **Manual test plan:** happy path, empty state, error states, invalid input, reload/persistence, mobile width, keyboard-only, slow network.
2. **Acceptance criteria** from your PRD, checked one by one.
3. **Ask the AI to write tests** — then read them: do they test behaviour you care about, or just that the code runs?
4. **Automated tests** (unit/integration/end-to-end) for anything that will live longer than a demo.
5. **Peer review** of the diff before merging.

11.10 Security risks of generated code

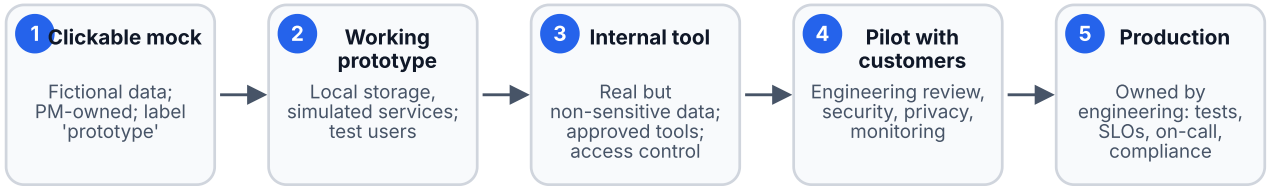
Risk	Example	Mitigation
Exposed secrets	API key hard-coded in `app.js`	Server-side calls; secret manager; scan repos
Broken access control	Any logged-in user can read all records	Server-side authorisation; DB access rules (e.g., RLS); test with two accounts
Injection / XSS	Rendering user text with `innerHTML`	Escape output; parameterised queries
Insecure dependencies	Outdated or non-existent (hallucinated) package names	Verify packages exist and are reputable; pin versions; scan
Over-permissive tools	Agent runs destructive commands	Review commands; least privilege; sandbox
Data leakage	Pasting customer data into prompts	Use dummy data; approved tools only
Missing rate limits	OTP/login endpoints can be abused	Rate limit; monitoring

A note on the starter project

The Fee-Clear Cart prototype uses `innerHTML` only with its own fixed menu data. If you add user-entered text (e.g., notes), render it with `textContent` instead — that is exactly the kind of change an AI tool may get wrong.

11.11 Prototyping versus production development — and when engineering review is necessary

Prototype	Production
Answers a learning question	Serves real users reliably
Fictional or test data	Real, regulated data
One happy path + key errors	All edge cases, scale, monitoring
Can be thrown away	Must be maintained
PM can own	Engineering owns; PM partners



The readiness ladder — what each rung requires. Every step up adds obligations. Vibe coding is fastest on the first two rungs; from rung 3 upward engineering and security must be involved.

Engineering review is necessary before: any real user data or personal information; authentication or payments; integration with company systems; anything customers rely on; anything public under the company's name; anything that runs automated actions (emails, money movement); any code that will be reused in production.

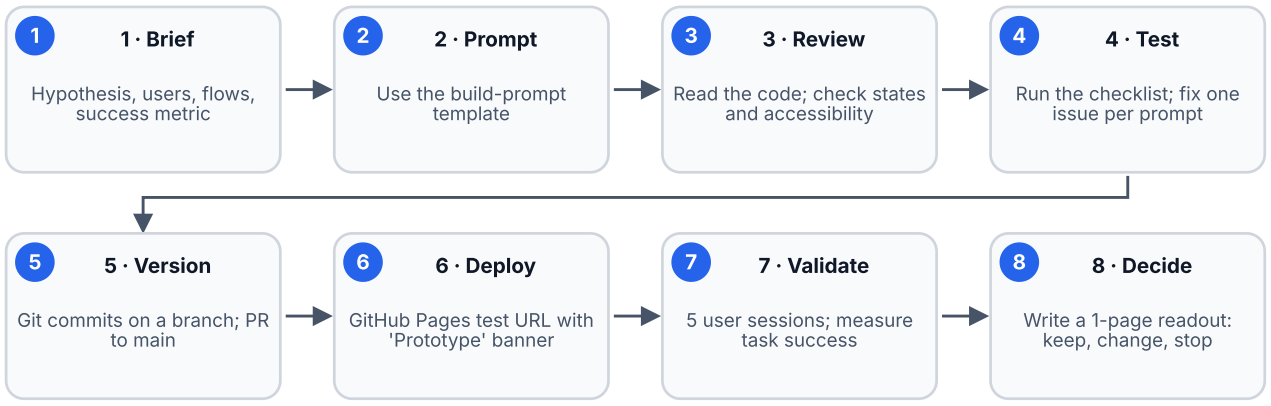
11.12 Building and validating MVPs with vibe coding

Use the speed to **test assumptions**, not to skip discovery: (1) pick the riskiest assumption; (2) build the smallest prototype that tests it; (3) put it in front of 5–8 target users with a task; (4) measure behaviour (task success, time, errors) and listen; (5) decide: iterate, pivot, or stop. *A polished demo is not evidence of demand.*

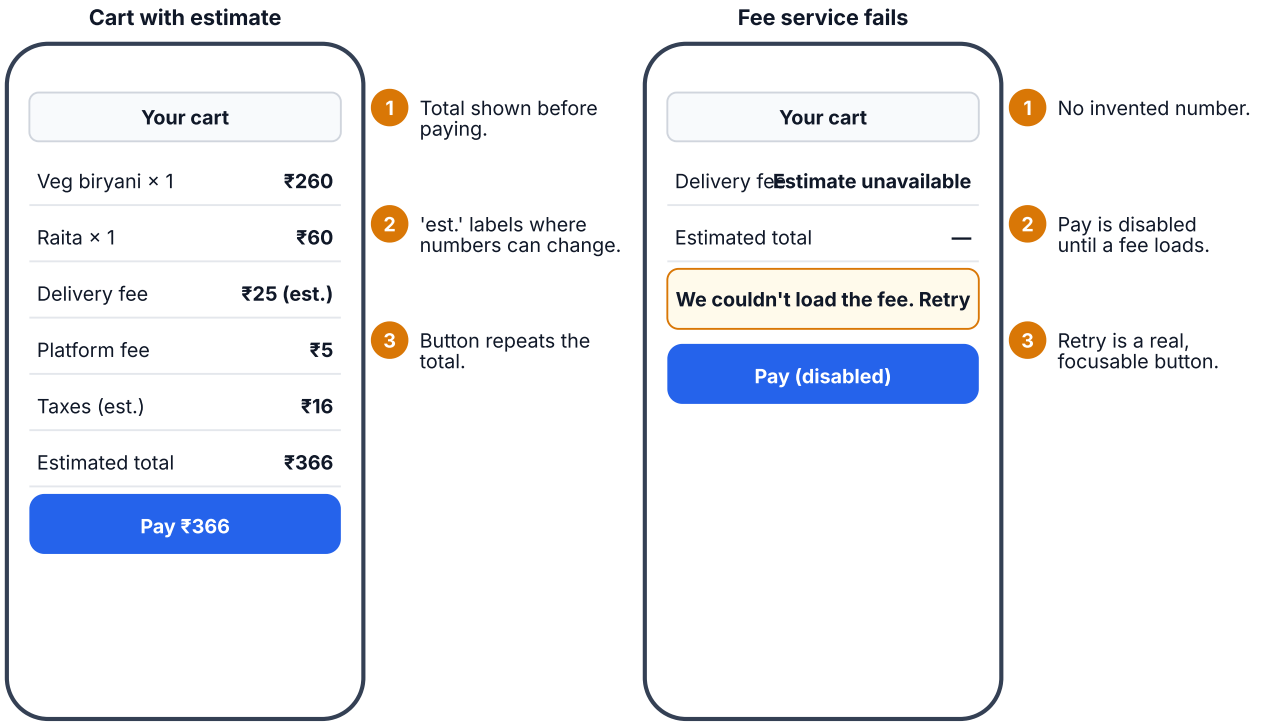
> **From your course notes (Lectures 32, 43, 55)**
>
> Label the prototype's **real and simulated functions**; a polished screen does not prove that a service works. Show the problem, target user, evidence, solution, prototype, business model, success measures and the next validation step. **Distinguish a clickable demo from a working integration, and planned outcomes from measured results. Never publish invented testimonials as if they were real.**

11.13 Hands-on project: from idea to deployed prototype

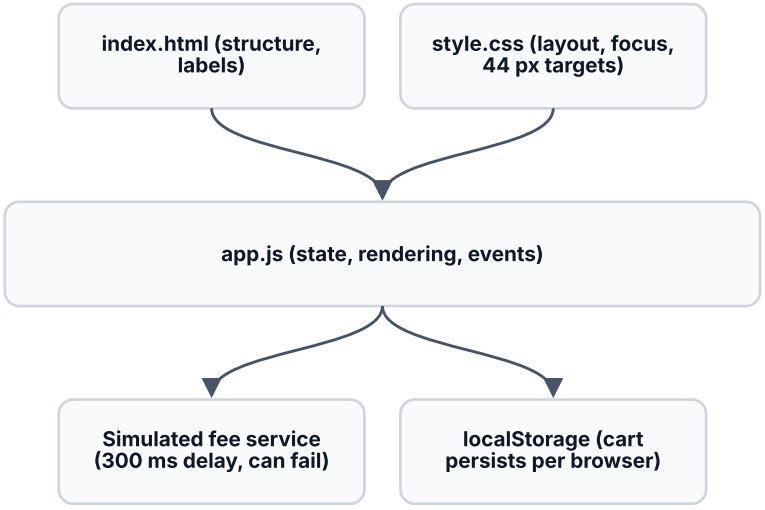
Project: *Fee-Clear Cart* — test whether showing the full estimated total in the cart reduces checkout surprise. The finished starter code is in `templates/vibe-coding-starter/` (open `index.html`; it runs offline). It was tested in a browser at 375 px width: adding items, changing the area, simulated fee failure (“Estimate unavailable”, Pay disabled, Retry), and the simulated order message all behave as specified, with no console errors.



Project milestones. Each milestone has a concrete artefact. Total time for a first-timer: roughly one weekend.



Fee-Clear Cart — the starter app's two key states (illustrative wireframe, not a real product screenshot). Values match the tested build (subtotal ₹320, near area, 5% illustrative tax). This is the teaching prototype in templates/vibe-coding-starter, not a real product.



How the starter app is put together. Three files, no server, no keys. Replacing the simulated service with a real API is the step that requires a server-side function and engineering review.

Step 1 — Brief (example). Hypothesis: first-time buyers who see the full estimated total in the cart will complete checkout more confidently. Users: busy office workers ordering dinner on mobile. Flows:

add items; see estimated total; handle fee-service failure. Metric for the user test: % of participants who correctly state the total *before* tapping Pay (target ≥ 5 of 6).

Step 2 — Prompt. Use the template above. Ask for plain HTML/CSS/JS and a simulated fee service with a failure mode.

Step 3 — Review. Read `app.js`: find where the fee is fetched; confirm that on failure it shows "Estimate unavailable" and disables Pay (it does — `feeState.status = "error"`). Check labels, focus styles and button height in `style.css`.

Step 4 — Test. Run the checklist in the starter's README (add/remove, area change, failure + retry, keyboard-only, reload persistence, 375 px width).

Step 5 — Version. Commit after each working step; push to a branch; open a PR describing what changed and how you tested it.

Step 6 — Deploy. Public GitHub repo (dummy data only) → Settings → Pages → Deploy from branch `main` / root → share the URL **with the Prototype banner visible.** Do not add analytics that collect personal data; if you need behaviour data, observe sessions or log anonymous counts with consent.

Step 7 — Validate. Five or six moderated sessions: "You want this biryani and raita delivered 5 km away. How much will you pay in total? Now place the order." Record task success, hesitations and quotes.

Step 8 — Decide. Readout: what you tested, results (e.g., "5 of 6 stated the correct total before paying; 1 missed the 'est.' label"), what you'll change, what remains unproven (real fee accuracy, real conversion), and the next test (A/B in the real product — Module 7).

11.13.1 Portfolio presentation

Include: the problem and hypothesis; a 60-second screen recording; the live prototype link; what is simulated; test method and **real** results (or "not measured"); what you learned; how a production version would differ (security, data, scale); and your role (be specific about what the AI generated vs what you decided and tested).

11.14 Exercises with model answers

Exercise 1 · Spot the problems

An AI-generated app stores an API key in `app.js`, renders user comments with `innerHTML`, and saves data only to `localStorage`. You plan to share it with 50 customers. What must change?

Model answer

(1) Move the API call server-side (serverless function) and rotate the exposed key. (2) Render comments with `textContent` or sanitise to prevent XSS. (3) `localStorage` means each customer sees only their own browser's data — if shared data is intended, add a backend with access rules. (4) Add a prototype banner, privacy notice, and get engineering/security review because real customers are involved.

Exercise 2 · Write the debugging prompt

The page shows the cart but the total never appears. Console: `TypeError: Cannot read properties of undefined (reading 'price')`. Write your prompt to the AI.

Model answer

“When I add 'Paneer wrap' the total never appears. Console error: `TypeError: Cannot read properties of undefined (reading 'price')` at `subtotal()` in app.js. Before changing code, explain the likely cause. I suspect the cart contains an id not present in MENU. Propose the smallest fix and a test I can run to confirm. Don't change any other behaviour.”

Exercise 3 · Prototype or production?

Your vibe-coded internal dashboard reads the company's live orders database. Who needs to be involved and why?

Model answer

Engineering (data access method, read-only replica, performance), security/IT (credentials, access control, approved tools), data/privacy (PII exposure, retention), and the data owner (permission). Even read-only access to production data is a production concern; use an approved read-only replica or an extract with minimised fields.

11.15 Interview perspective

“How do you use AI coding tools as a PM?” — 60-second answer

“I use them to make ideas testable quickly — clickable prototypes with fictional data that I put in front of users within days. I write a short brief with the hypothesis, flows, states and accessibility needs, build in small steps, test each one, and keep everything in Git. I label what's simulated, never use real customer data, and I hand off to engineering for anything with real data, auth, payments or customers relying on it. The value is faster learning, not shipping AI-written code to production unreviewed.”

“Walk me through a prototype you built.” — structure

Problem and hypothesis → users → what you built (stack, scope, what's simulated) → how you tested it (sessions, metrics) → results (real numbers or “not measured”) → what you'd do next → what engineering would need to productionise it. Be explicit about your contribution versus the AI's.

11.16 Key takeaways

- Vibe coding accelerates **learning**; it doesn't remove the need for **engineering judgment**.
- Work in **small, tested, committed steps**; describe symptoms precisely when debugging.
- Know **HTML/CSS/JS, Git, APIs and storage** well enough to review what the AI produced.
- **Never put secrets in frontends**, never use real customer data in unreviewed prototypes, and label simulations.
- Bring in **engineering review** before real data, auth, payments, integrations or public company use.

11.17 Check your understanding

CHECK YOUR UNDERSTANDING

Where should a secret API key live in a web prototype?

A. In app.js so it's easy to find

B. In a server-side function or secret manager

C. In the README

D. In localStorage

Why: Anything in frontend code is visible to users; secrets must be server-side.

A form appears to save data, but after reload it's gone. Most likely...

A. The AI tool is broken

B. The UI isn't connected to persistent storage

C. The browser is too old

D. It's an accessibility issue

Why: A screen with a form isn't necessarily wired to storage; verify the data flow.

Which practice makes AI-assisted coding mistakes cheapest to undo?

A. One huge prompt for the whole app

B. Small steps with a commit after each working step

C. Never testing

D. Editing production directly

Why: Small committed increments let you revert precisely.

When is engineering review required?

A. Never for prototypes

B. Before real user data, auth, payments, integrations or public company use

C. Only for mobile apps

D. Only if the AI says so

Why: These introduce security, privacy, reliability and legal obligations.

Rendering user-entered text with innerHTML risks...

A. Slow performance only

B. Cross-site scripting (XSS)

C. Nothing

D. Rate limiting

Why: innerHTML can execute injected markup; use textContent or sanitisation.

11.18 Recommended resources

- MDN: Learn web development — HTML, CSS, JavaScript basics.
- GitHub Docs: Hello World and GitHub Pages quickstart.
- Claude Code documentation · Cursor docs · GitHub Copilot docs.
- Lovable docs and Replit Agent docs (from your course notes).
- OWASP Top 10 — common web security risks.