

• COMPLETE REFERENCE GUIDE 2026

Master Lovable From A to Z

Methodology, prompts, security, SEO, Stripe, authentication, API integration, extensions and end-to-end workflows. Everything you need to build seriously — without coding.

 End-to-end workflows

 Security & architecture

 Stripe step by step

 APIs & integrations

 SEO & search

 Auth & passwords

 51 tips & checklists

23

sections + bonus

60+

ready-to-use prompts

3

end-to-end workflows

25+

integrations

01	What is Lovable? Platform overview, tech stack, strengths and limits
02	Tool Comparison Lovable vs Claude Code vs Cursor vs Bubble
03	Before You Start Credits, product brief, initial setup
04	The 6-Phase Methodology Sequential approach that eliminates 80% of common problems
05	The Art of Prompting 4 mastery levels, advanced techniques, anti-blocking checklist
06	The Knowledge File Full template — your project's living memory
07	GitHub & Versioning Branch strategy, pinning stable versions
08	Supabase & Database Connection, RLS, schema design, Stripe, OAuth
09	Deployment Cloudflare, Vercel, Netlify, pre-launch checklist
10	Debugging & Errors 5-step workflow, debug prompts, when to restart
11	51 Practical Tips Community best practices, organised by theme
12	Advanced Workflow: Lovable + Claude The combo that cuts costs by 90%
13	Ready-to-Use Prompt Templates Copy-paste into Lovable immediately
14	Full FAQ 10 most-asked questions from the community
15	End-to-End Workflow: SaaS App FitBook booking app — 6 complete phases with all prompts

16	End-to-End Workflow: Professional Website Landing page + blog CMS + qualified form + SEO
17	End-to-End Workflow: Internal Business Tool B2B CRM — pipeline, KPIs, CSV import
18	Advanced Security & Architecture RLS audit, OWASP threats, architecture checklist
19	Authentication & Password Management Supabase Auth, OAuth, MFA, password policy
20	SEO & Organic Search Meta tags, sitemap, Schema.org, Core Web Vitals
21	Integrating Stripe Step by Step Payments, subscriptions, webhooks, customer portal
22	External API Integration REST, GraphQL, OpenAI streaming, webhooks, Make.com, GA4
23	Extensions & Available Integrations 25+ tools: analytics, email, AI, automation, storage, maps

What is Lovable?

Lovable is an AI-driven web application builder. You describe what you want to build in natural language, and the tool generates the corresponding code in real time.

Lovable is neither a classic website generator, a drag-and-drop tool, nor a traditional IDE. It is a **complete AI development platform** that produces React + Tailwind CSS applications connectable to Supabase, GitHub, Stripe and many APIs.



The right definition

What Lovable creates is a real web application with user interface, logic, authentication flows, databases, backend functions, publishing options and GitHub synchronisation — not just a mockup.

What Lovable Does Best

Lovable excels on projects combining pages, forms, tables, filters, charts, authentication and CRUD logic:

- Dashboards and client portals
- Internal business tools
- SaaS interfaces and rapid MVPs
- Lightweight marketplaces and booking flows
- Admin panels
- Landing pages and campaign microsites

Where Lovable Struggles

- Highly customised business rules and complex logic
- Unusual architectures and elaborate state management
- Intensive exception handling within interconnected workflows
- Applications requiring strict unit testing from day one



The right mental model

"Lovable can *accelerate* development of a wide range of modern web applications — it is not a substitute for the entire development chain, but a powerful accelerator at the start of a project."

The Generated Tech Stack

Layer	Technology	Notes
Frontend	React 18 + TypeScript	Modern components with hooks
Styling	Tailwind CSS + ShadCN/UI	Cohesive, responsive design system
Backend	Supabase	PostgreSQL + Auth + Storage + Edge Functions
Version control	GitHub	Automatic sync on every change
Deployment	Lovable / Netlify / Vercel / Cloudflare	Your choice based on needs

Lovable vs Other Tools

Every tool has its place in the ecosystem. Here is how to choose intelligently.

Lovable AI no-code builder <table><tr><td>Code required</td><td>None</td></tr><tr><td>MVP speed</td><td>Very fast</td></tr><tr><td>Backend</td><td>Supabase built-in</td></tr><tr><td>Code control</td><td>Partial</td></tr><tr><td>Pricing</td><td>Credits</td></tr></table>	Code required	None	MVP speed	Very fast	Backend	Supabase built-in	Code control	Partial	Pricing	Credits	Claude Code AI terminal IDE <table><tr><td>Code required</td><td>Basics helpful</td></tr><tr><td>MVP speed</td><td>Medium</td></tr><tr><td>Backend</td><td>Any backend</td></tr><tr><td>Code control</td><td>Full</td></tr><tr><td>Pricing</td><td>Fixed subscription</td></tr></table>	Code required	Basics helpful	MVP speed	Medium	Backend	Any backend	Code control	Full	Pricing	Fixed subscription
Code required	None																				
MVP speed	Very fast																				
Backend	Supabase built-in																				
Code control	Partial																				
Pricing	Credits																				
Code required	Basics helpful																				
MVP speed	Medium																				
Backend	Any backend																				
Code control	Full																				
Pricing	Fixed subscription																				
Cursor AI code-first IDE <table><tr><td>Code required</td><td>Yes</td></tr><tr><td>MVP speed</td><td>Good</td></tr><tr><td>Backend</td><td>Any backend</td></tr><tr><td>Code control</td><td>Full</td></tr><tr><td>Pricing</td><td>Fixed subscription</td></tr></table>	Code required	Yes	MVP speed	Good	Backend	Any backend	Code control	Full	Pricing	Fixed subscription	Bubble Traditional no-code <table><tr><td>Code required</td><td>None</td></tr><tr><td>MVP speed</td><td>Slow</td></tr><tr><td>Backend</td><td>Proprietary</td></tr><tr><td>Code control</td><td>Limited</td></tr><tr><td>Pricing</td><td>Expensive</td></tr></table>	Code required	None	MVP speed	Slow	Backend	Proprietary	Code control	Limited	Pricing	Expensive
Code required	Yes																				
MVP speed	Good																				
Backend	Any backend																				
Code control	Full																				
Pricing	Fixed subscription																				
Code required	None																				
MVP speed	Slow																				
Backend	Proprietary																				
Code control	Limited																				
Pricing	Expensive																				

When to Choose What?

Situation	Recommended Tool	Why
MVP to validate quickly	Lovable	Maximum speed, real-time preview
MVP + complex logic	Lovable + Claude Code	Lovable for frontend, Claude for logic
Solo developer project	Cursor + Claude	Full code control, no credit limits
Marketing / showcase site	Webflow / WordPress	Dedicated tools, better native SEO
Complex production app	Lovable → migrate to GitHub/IDE	Start fast, then take full control



Expert tip

The

Lovable + Claude Code via GitHub

combination is the current standard among serious AI builders. Lovable for speed, Claude for depth. The community reports up to 90% reduction in Lovable credit usage with this combo.

Before You Start: The Fundamentals

The biggest mistake is rushing into Lovable with a vague idea. 20 minutes of preparation prevents hours of corrections.

Understanding the Credit System

Lovable runs on a credit system — each code generation consumes credits. Plan Mode (chat, planning, debugging) does **not** consume any credits.

Plan	Credits/month	Recommended use
Free	5 credits	Testing and discovery only
Starter (~\$20/mo)	100 credits	1–2 MVP projects
Pro (~\$50/mo)	Extended credits	Serious projects, frequent iteration
Teams	Shared credits	Teams and collaboration



Save your credits

Use

Plan Mode 60–70% of the time

to debug, plan and iterate without consuming a single credit. Only click "Implement" when you are confident in the result.

The Product Brief: 4 Fundamental Questions



Product Brief — Starting Checklist

- ☐ **What am I building?** — Describe the product in 1–2 sentences without technical jargon
- ☐ **Who is it for?** — Define personas and user roles (Admin, User, Visitor...)
- ☐ **Why?** — What problem does it solve? What value does it deliver?
- ☐ **What single primary action?** — One action the app must make effortless
- ☐ **Maturity level?** — POC, usable MVP, or full production?
- ☐ **What exclusions?** — What the app will NOT do in this version
- ☐ **Data** — Do you have CSV files, APIs or data to integrate? If so, document each column

Initial Setup: What to Do First

1. **Create a Lovable account** — Choose the right plan for your usage
2. **Connect GitHub immediately** — From the very first MVP, activate sync. Every change becomes an automatic commit
3. **Prepare your Knowledge File** — Your project's reference document (see Section 6)
4. **Write Prompt 0** — The immutable context for the project, before touching the interface

The 6-Phase Sequential Method

The fundamental rule: **each phase must be explicitly validated before moving to the next**. This discipline prevents 80% of classic problems.



The most frequent mistake

Starting with "Create a nice app with a modern dashboard." This type of request pushes Lovable to invent filters, fake data, business rules and pages you never asked for.

PHASE

0



Immutable Context

Define the strategic framework: product objective, target audience, hard constraints, explicit exclusions, maturity level (POC / MVP / Production). At this stage, do not mention interfaces, code or data.

Optional but recommended

Strategy

Zero code

PHASE

1



Architecture & Functional Scope

Define the overall structure: main pages/views, functional modules, flows between modules, inclusions and exclusions. No design, data or detailed business logic yet.

Structure

Navigation

User flows

PHASE

2



Data & Sources

Secure the logical foundation: data sources, exact column names with types and nullability constraints, update frequency, contractual rules. A CSV is never "just imported" — it must be described column by column.

Critical

Contractual columns

No mock data

PHASE

3



Business Logic & Rules

Prevent Lovable from inventing behaviours. Every business rule must be written explicitly: calculation rules, exceptions, priorities, edge cases and error rules. If a rule is not written, it does not exist.

Explicit rules

No inference

Edge cases

PHASE

4



UI/UX & User Journeys

Build the interface only after locking architecture, data and logic. Define screens, states (loading / error / empty), journeys and accessibility. The UI displays and triggers actions — it must not contain core business logic.

Mobile-first

Empty states

Accessibility

PHASE

5

✓ Verification, Testing & Non-Regression

The last prompt does not add features — it verifies that Lovable has not changed scope, renamed columns, invented data, moved logic into the UI, added unrequested filters or broken an existing feature.

Non-regression

Compliance report

Final check

The 10 Golden Rules



Lovable Golden Rules

- Never ask for a complete application in a single prompt
- Never start with the user interface
- Never import a CSV without a contractual column-by-column description
- Never let Lovable infer a business rule
- Never accept silently generated fake data
- Never move to the next phase without explicit validation
- Always restate validated constraints in each new prompt
- Always define exclusions as precisely as inclusions
- Always account for error, loading and empty states
- Always finish with a testing and non-regression prompt

Reliable Prompt Template

STRUCTURED PROMPT TEMPLATE

Copy-paste ready

Step X – [Step Name]

Objective:

[Precise, measurable objective for this step]

Already validated context:

[Reminder of constraints and decisions from previous steps]

Instructions:

1. [Precise instruction]
2. [Precise instruction]
3. [Precise instruction]

Prohibitions:

1. Do not rename contractual columns
2. Do not add features outside the listed scope
3. Do not modify pages X and Y

Validation criteria:

1. [What must be true to validate this step]
2. [Concrete verification]

→ End with: "Do you validate this prompt to proceed to the next step?"

The Art of Prompting: 4 Mastery Levels

The quality of your prompt directly determines the quality of the result. A well-structured 50-word prompt beats a confusing 500-word wall of text.

Level 1

Training Wheels

Beginners, workshops, first draft

Markdown headings: Context → Task → Constraints. Ideal for learning prompting discipline.

```
# Context
Generate a contacts CRUD.

## Task
Create tables + UI.

### Constraints
No custom CSS.
```

Level 2

No Training Wheels

Short prompts, time-pressed users

Same logic, headings disappear. Keep Context-Task-Constraints order in one paragraph.

```
"Generate a contacts CRUD
(Postgres). Tailwind UI,
no refactor without
explicit validation."
```

Level 3

Meta-Prompting

Save time, discover structure

Ask a LLM to write the prompt for you — just give it the topic and let it structure the request.

```
"You are a prompt engineer.
Write the best prompt to
create a simple CRM with
Supabase, specifying all
possible context."
```

Level 4

Reverse Prompting

Documentation, cloning, team retros

At session end, the AI summarises: the prompt + pitfalls avoided = a perfect base for V2 or sharing.

```
"Summarise this project,
list the pitfalls we hit,
and give a ready-to-use
prompt to clone it."
```

Key Prompting Techniques

Locking Files and Components

LOCK PROMPT

Implement modifications to the feature while ensuring core functionality, other features, and processes remain unaffected. Evaluate dependencies and identify potential risks before moving forward. Conduct thorough testing to verify there are no regressions, and highlight any out-of-scope changes. Exercise caution – pause if uncertain.

// Or more simply:
Do not modify /pages/Settings.tsx or /shared/Layout.tsx.
Focus changes only on /components/Dashboard.tsx.

UI-Only Changes (No Logic)

VISUAL CHANGES ONLY

Make solely visual enhancements – ensure functionality and logic remain unaffected. Verify that logic, state management, and APIs stay intact. Test that the app operates exactly as before. Stop if there is any uncertainty about unintended consequences.

Ask for Clarification Before Coding

PRE-FLIGHT CLARIFICATION

Ask me any questions you need in order to fully understand what I want from this feature and how I envision it. Do not write any code until I have answered your questions.

Good vs Bad Prompts

BAD

"Show the best customers."

GOOD

"A top customer is one whose total revenue exceeds \$10,000 over the past 12 months. Cancelled orders are excluded."

✗ BAD

"Use my customer CSV."

✓ GOOD

"The CSV has: customer_id (string, non-null), name (string), email (nullable), status (enum: active/inactive/prospect). These names are contractual."

Anti-Blocking Checklist

Anti-Blocking — Verify Before Every Generation

- ☐ Both input AND expected output clearly defined in the prompt
- ☐ Tech stack named: React, Tailwind, Supabase Edge Functions
- ☐ Anti-refactor rule included: "No refactor without explicit validation"
- ☐ Plan Mode activated and used for clarifications before generation
- ☐ Single topic per prompt (no multi-subject mixing)
- ☐ User role specified if the app has multiple profiles
- ☐ Reverse prompt generated and archived after each important session
- ☐ Version pinned before any significant change

The Knowledge File: Your Project's Memory

The Knowledge File is your project's brain. It is sent with every prompt, giving Lovable full context to avoid inconsistencies.



It's your living mini-PRD

The Knowledge File contains the product vision, personas, journeys, key features, UI guidelines and role-based rules. Update it after every major phase.

Complete Knowledge File Template

knowledge.md — Template to customise

1. PRODUCT VISION

Name: [Application name]
Objective: [What the app does in 1 sentence]
Problem solved: [Precise user problem]
Unique value: [Why it's better/different]
Maturity level: [POC / MVP / Production]

2. USERS AND ROLES

Admin role:

- Permissions: [full list]
- Accessible pages: [list]

User role:

- Permissions: [list]
- Accessible pages: [list]

Visitor role:

- Access: landing page only

3. INCLUDED FEATURES (MVP)

- ✓ [Feature 1] – [Short description]
- ✓ [Feature 2] – [Short description]

✗ EXCLUSIONS (do not implement):

- [Exclusion 1]
- [Exclusion 2]

4. TECH STACK

Frontend: React 18 + TypeScript
Styling: Tailwind CSS + ShadCN/UI
Backend: Supabase (PostgreSQL + Auth + Edge Functions)
Deployment: [Lovable.app / Cloudflare Pages / Vercel]
Version control: GitHub

5. LOVABLE RULES

- Never refactor without explicit validation
- Never rename data columns
- Never add out-of-scope features
- Never use fake or mocked data
- Always specify empty, error and loading states



Generate the Knowledge File automatically

Ask Plan Mode: *"Generate the knowledge file for my project based on the features already implemented."* Update it regularly after each major phase.

GitHub: Versioning and Collaboration

GitHub is your safety net. Connect it from the very first MVP — every Lovable change becomes an automatic commit.

Connecting GitHub — Step by Step

- 1. In Lovable, click the **GitHub** button in the top right
- 2. Authorise access to your GitHub account
- 3. Create a new repository or select an existing one
- 4. From that point, every change generates an automatic commit
- 5. You can clone the repo locally or open it in Claude Code / Cursor

Recommended Branch Strategy

Branch	Use	Rule
main	Stable production	Never push directly from Lovable in prod
dev	Active development	Main Lovable working branch
uat	User acceptance testing	Create once the MVP is stable
feature/xxx	New features	Via Claude Code or Cursor, merge via PR

When to Pin a Version



Pin a version when:

- ☐ After every feature that works correctly
- ☐ Before starting a risky or complex change
- ☐ After a significant refactoring
- ☐ Before connecting Supabase or an external API
- ☐ Before changing the database schema

**Caution with Supabase rollbacks**

Rolling back a Lovable version does not undo Supabase schema changes. Test every data-related feature carefully before publishing.

Supabase: Database and Authentication

Supabase is Lovable's recommended backend — PostgreSQL, authentication, storage and Edge Functions in a single service.



Supabase golden rule

Connect the database

once the frontend is sufficiently stable

. Never touch the schema too early — migrations are costly to fix.

Connecting Supabase to Lovable

1. Create a project at supabase.com
2. In Lovable, go to **Settings** → **Integrations** → **Supabase**
3. Paste the project URL and anonymous key
4. Prompt: *"Configure Supabase with user authentication"*
5. Lovable automatically sets up Auth, base tables and initial RLS policies

Row Level Security (RLS) — The Basics

Policy	Description	Example
SELECT by user	A user only sees their own data	<code>auth.uid() = user_id</code>
Authenticated INSERT	Only logged-in users create records	<code>auth.role() = 'authenticated'</code>
Admin full-access	Admin sees everything	<code>is_admin(auth.uid())</code>

Supabase Pre-Deployment Checklist

Supabase Security Checklist

- ☐ RLS enabled on all tables containing user data
- ☐ Email confirmation disabled in dev, re-enabled in production
- ☐ API keys stored in environment variables (never in code)
- ☐ .env added to .gitignore to prevent credential leaks
- ☐ Staging branch tested before every push to production
- ☐ Lovable 2.0 Security Review run before publishing
- ☐ UUIDs used for all relationships between tables

Advanced Integrations

- **Stripe:** *"Add a checkout button and process payments with Stripe."* Lovable creates an Edge Function that communicates with the Stripe API.
- **Social OAuth (Google, GitHub):** In Supabase → Auth → Providers, enable OAuth then ask Lovable: *"Add a Sign in with Google button."*
- **Make.com / n8n:** Use webhooks to connect Lovable to external services (Slack, CRM, notifications, etc.)

Deployment: From Lovable to Production

Lovable hosts for development, but for production, externalise to Netlify, Vercel or Cloudflare Pages for more control.

Platform	Free tier	Auto-deploy	Custom domain	Best for
Lovable.app	Limited	✓	Paid plans	Dev & rapid testing
Cloudflare Pages	✓ Generous	✓	✓ Included	Recommended for production
Vercel	✓	✓	✓	Complex React apps
Netlify	✓	✓	✓	Sites + forms + functions



Publishing ≠ Production-ready

Publishing means the app is accessible. Going to production requires a full review: access control, security, data behaviour, reliability and manual testing.

Pre-Deployment Checklist



Before Going Live

- ☐ Lovable Security Review run and validated
- ☐ Supabase RLS enabled and tested on all tables
- ☐ Email confirmation re-enabled in Supabase
- ☐ Environment variables configured for production
- ☐ All roles tested (Admin, User, Visitor)
- ☐ Full mobile test (iOS + Android)
- ☐ Authentication flow tested end-to-end
- ☐ Custom domain configured with HTTPS active
- ☐ Supabase staging cloned to separate PROD project

Debugging: Breaking Out of Error Loops

Error loops are the primary source of frustration with Lovable. Here is a systematic method to escape them.

5-Step Debugging Workflow



1. Console
F12 → Logs



2. Describe
Expected vs actual



3. Plan Mode
Diagnose, no code



4. Fix
One targeted fix



5. Rollback
If 3+ attempts fail

Essential Debug Prompts

EFFECTIVE BUG REPORT

On `/settings`, I expected [expected behaviour]
but I get [observed behaviour].

Console error:

[Paste exact error from Chrome DevTools]

Files involved: [List visibly affected components]

// Do not modify unlisted files.

// Start with Plan Mode: identify without coding.

BREAK OUT OF AN ERROR LOOP

I have been stuck on this bug for [N] attempts.
Rather than coding, give me 3 different approaches
to solve this problem without touching the code yet.

What has been tried: [summary]

Current error: [error]



Signs you should Remix (start clean)

Use the

Remix

button to create a clean copy when: the same bug returns after 3+ fix attempts; fixing one bug creates two new ones; accumulated technical debt costs more than a fresh start. Always disconnect Supabase first.

51 Tips to Master Lovable

Community best practices, organised by theme.



Prompting & Instructions (1–10)

01

Context → Task → Constraints

Systematic structure for every prompt. The AI prioritises the start and end — put critical details first.

02

One topic per prompt

Break requests into testable steps. Avoid mixing 5 unrelated changes in one prompt.

03

Ask for questions before code

"Ask me any questions you need before writing any code." Massively reduces back-and-forth.

04

Repeat important instructions

Model memory can be limited between exchanges. Repeat critical constraints regularly.

05

Name the screen and role

Always specify the route (/settings) and the user role involved to avoid side effects.

06

Use real content from the start

Ban "lorem ipsum" and "Feature 1". Real content helps Lovable structure the interface better.

07

Define visual style early

Choose a style word (calm, premium, expressive, minimal) and repeat it in design prompts.

08

Specify the user role

If your app has multiple roles (Admin, User), always indicate which one is involved in the prompt.

09

Build a reusable prompt library

Log your effective prompts in Notion or a MD file. Prompt consistency creates app consistency.

10

Meta-prompting for structure

Ask Claude Chat to write your prompt for you if you don't know where to start.

Workflow & Organisation (11–20)

11 Plan Mode 60–70% of the time

Debug, brainstorm and plan without consuming credits. Click "Implement" only when satisfied.

12 GitHub from the first MVP

Activate immediately. Every change becomes a commit, giving you a full history and backup.

13 Pin stable versions

After each working feature, pin it. Compare versions visually to identify regressions.

14 Mobile-first systematically

Use the responsive mobile-first prompt for every new screen. ShadCN + native Tailwind only.

15 Keep the Knowledge File current

Your project's brain is sent with every prompt. Update after each major phase.

16 Screenshots for design

Capture a design you like and drag it into Lovable for inspiration. Massive time saver for prototyping.

17 Figma + Builder.io

Transform your Figma wireframes into pixel-perfect Lovable projects via the Builder.io integration.

18 Remix to start clean

If the project is too damaged, Remix creates a clean copy. Disconnect Supabase first.

19 MVP $\leq$ 5 core features

Each additional feature exponentially increases complexity. Start with the minimum viable.

20 Work with atomic components

"A form with an email field and a rounded button" works better than "a sign-up section".



Backend, Database & Security (21–30)

21

Configure Supabase first

Lock your backend before adding features. Prompt: "Configure Supabase with user authentication".

22

Email confirmation: off in dev

In Supabase → Auth → Settings, disable "Confirm email" during dev. Re-enable before production.

23

RLS on all sensitive tables

Lovable sets up the basics, but manually verify every table containing user data.

24

UAT branch before production

Once the MVP is stable, create a UAT branch and clone your Supabase project as PROD.

25

Stripe via Edge Functions

"Add a checkout button and process payments with Stripe." Lovable creates the Edge Function automatically.

26

UUIDs for relationships

Verify that foreign keys match the types of their primary keys to avoid silent errors.

27

Social auth in 5 minutes

Supabase → Auth → Providers → enable OAuth, then: "Add a Sign in with Google button."

28

Staging/production toggle

Lovable 2.0 includes a native staging/production toggle. Always test on staging before pushing to prod.

29

Secrets in env variables

Never put API keys in code. Settings → Environment Variables, and .env in .gitignore.

30

Security Review before deploy

Lovable 2.0 includes an AI security scan. Use it systematically and enable API key leak detection.



Debugging & Errors (31–40)

31 Chrome console first

F12 → console logs, network requests, JS errors.
Always before launching a new fix prompt.

32 Bug report format: expected vs actual

"On /page, I expected X but get Y. Console error: [error]." Precision reduces fix attempts.

33 Break loops: 3 approaches

If the AI loops: 1) ask for 3 approaches without coding, 2) roll back to a pinned version, 3) rebuild from scratch.

34 Refactor regularly

"Refactor this file without changing UI or functionality. Document changes and implement gradually."

35 Plan Mode to diagnose

"Identify the problem but don't write code." You stay in control of the solution applied.

36 Maintain a filesExplainer.md

An up-to-date document describing your components and their role prevents the AI from breaking unknown components.

37 Test all roles after every change

A shared component change can break multiple views simultaneously. Check Admin, User, Visitor.

38 Role-specific components

DashboardAdmin, DashboardUser rather than a shared component. Prevents side effects during fixes.

39 "Try to Fix" with discernment

If "Try to Fix" doesn't resolve after 2–3 attempts, stop. Reformulate with exact context rather than looping.

40 Screenshots for UI bugs

Upload a screenshot of the bug. "Why is this UI behaving this way? What's the best fix?" is very effective.



UI, Design & Visual Edits (41–45)

41 Visual Edits for pixel-perfect

Free and instant: click any element to edit it directly like in Figma, no prompt needed.

42 UI reference image

Attach a reference image to your prompt. The AI interprets visuals faster than text descriptions.

43 ShadCN + Tailwind as base

These libraries are perfectly integrated into Lovable. Specify them explicitly for maintainable code.

44 Role-specific components

For multi-role apps, create dedicated components. Avoids shared-component complexity and bugs.

45 Dev Mode for code-first editing

Available on paid plans: edit the generated TypeScript/React directly for fine-grained adjustments.



Performance, Deployment & Community (46–51)

46

Supabase scales without rewriting

Lovable + Supabase supports thousands of users.
Plan your data structure from the start.

47

MVP first, features later

3–5 core features maximum for V1. Each addition exponentially increases complexity.

48

Netlify/Vercel for production

Connect your GitHub repo for automatic deployments. Lovable hosts dev; externalise production.

49

Lovable Discord

A goldmine: prompts shared by Champions, solutions to common problems, access to betas.

50

Iterate like a conversation

The best result comes after 2–3 iterations. Treat Lovable like a creative partner, not an oracle.

51

★ Lovable + Claude Code via GitHub

The combo that cuts costs by 10×. Lovable for generation and previews, Claude for logic and debugging. Community reports –90% in Lovable credit usage.

Lovable + Claude Code: The Winning Combo

The most powerful workflow in the AI dev community — Lovable for speed, Claude for depth.



The Logic of the Combo

Lovable for initial generation and previews — irreplaceable. Claude Code for logic, debugging and refactoring — with no credit system. Combined via GitHub, you get the speed of no-code and the power of AI-assisted code.

The Concrete Workflow

1. **In Lovable:** Click the GitHub button to sync your project. Every change becomes an automatic commit.
2. **Clone the repo:** Locally or directly in Claude Code / Cursor via the GitHub interface.
3. **In Claude Code:** Connect GitHub, select the Lovable repo, create a new branch. Request changes — debugging, refactoring, backend logic — without consuming a single Lovable credit.
4. **Pull Request:** Merge the branch via a GitHub PR.
5. **Lovable detects commits:** Automatically rebuilds the preview — you see the result in Lovable without having generated anything on its side.

Role Distribution

Tool	Optimal use	Avoid
Lovable	Initial generation, rapid UI prototyping, real-time previews, deployment	Massive refactoring, complex business logic
Claude Chat / Plan Mode	Brainstorming, PRD writing, complex debug, schema design	Direct code generation without validation
Claude Code	Refactoring, backend logic, precise fixes, optimisations	UI prototyping (no visual preview)
Cursor + Claude	Code-first editing, API management, automated agents	Initial visual prototypes

Ready-to-Use Prompt Templates

Copy-paste these templates directly into Lovable, adapting the sections in brackets.



Launching a New Project

PROJECT STARTER PROMPT

Level 1

```
# Context
I am building [type of application] for [target audience].
Objective: [what the app does in 1 sentence].

## Tech Stack
- Frontend: React 18 + TypeScript
- Styling: Tailwind CSS + ShadCN/UI
- Backend: Supabase (PostgreSQL + Auth)
- Mobile-first, native ShadCN breakpoints only

## Core Features (MVP only)
1. [Feature 1]
2. [Feature 2]
3. [Feature 3]

## Explicit Exclusions
- No [exclusion 1]
- No [exclusion 2]
- No fake or mocked data

## Visual Style
[calm / premium / expressive / minimal]
Primary colour: [#hex]

## First Step
Start with the home page containing:
[Precise description of the first screen]
```

Supabase Connection

INITIAL SUPABASE CONFIGURATION

Configure Supabase with:

1. User authentication (email + password)
2. Secure user dashboard
3. Row Level Security enabled on all tables
4. Route protection: redirect to /login if not authenticated
5. Do not create any mocked data or fake users

Roles:

- Admin: full access + user management
- User: access to their data only

Only modify files necessary for auth.

No global refactoring.

Targeted Bug Fix

PRECISE BUG FIX

Problem: On page [/route], when [action],
[observed behaviour] instead of [expected behaviour].

Console error:

[Paste exact error message]

Instructions:

1. Start by analysing the root cause in Plan Mode
2. Propose your solution BEFORE writing any code
3. Only modify the [ComponentName] component
4. Do not touch files: [list]
5. Test that all other features remain intact

✓ Non-Regression Verification

FINAL VERIFICATION PROMPT

Perform a complete non-regression check:

1. Verify the scope has not been modified
2. Verify no data columns have been renamed
3. Verify all listed business rules are respected
4. Verify user scenarios: [list]
5. Verify states: loading / error / empty / success
6. Verify no unsolicited features have been added
7. Verify business logic is not in UI components

Produce a compliance report:

- ✓ Compliant
- ✗ Non-compliant + correction needed
- ⚠ Point of attention

↻ Safe Refactoring

REFACTORING WITHOUT REGRESSION

Refactor [component/file] respecting these rules:

- Do not change the visible UI
- Do not change functionality
- Do not change API calls or state
- Document ALL changes made
- Implement gradually, section by section
- Stop and alert if a regression risk is detected

Objective: [improve readability / reduce duplication / optimise performance / etc.]

Frequently Asked Questions

Answers to the most commonly asked questions from the Lovable community.

Do I need to know how to code to use Lovable?

No, but "accessible" doesn't mean "without method". Mastery of Lovable comes from rigorous project preparation and structured prompts, not technical skill. Basic web knowledge (HTML, CSS, React) is a plus for debugging, but not required to get started.

Is the free plan enough?

The free plan (5 credits) may be enough to test the product and learn the workflow. However, for continuous product development, it will quickly prove insufficient. The Starter plan (~\$20/month) is the recommended minimum for a first serious project.

What should I do if Lovable doesn't understand my request?

Break your request down. If Lovable doesn't understand, it's almost always because the request is too vague or too broad. Enable Plan Mode, ask it to pose clarifying questions, then reformulate with one precise objective per prompt.

Can I manually edit the generated code?

Yes, in two ways: via Dev Mode (available on paid plans), which lets you edit the TypeScript/React directly, or via GitHub by cloning the repo and using your preferred editor (VS Code, Cursor, etc.). External changes are detected and automatically rebuilt in Lovable.

Does Lovable handle data security?

Lovable provides tools (Security Review, security scans, automatic RLS configuration), but this does not replace manual testing. For a production application, systematically verify RLS policies, environment variables, and test all user roles.

When should I move from Lovable to GitHub / a traditional IDE?

Move to a more thorough approach when precision, testing, collaboration or advanced customisation outweigh raw generation speed. In practice, many teams do this once the product has validated demand or entered a regular iteration phase. Best approach: use Lovable + GitHub simultaneously to get the best of both.

Is Lovable suitable for production applications?

Yes, but with the same rigour as any other application: backend review, security validation, authorisation testing, dependency management and operational checks. Publishing means the app is accessible — going to production requires a complete validation process.

How do I avoid Lovable "hallucinations"?

The 6-phase methodology is the primary answer: lock the scope, describe data contractually, write every business rule explicitly. The more precise your prompts, the less the AI invents. Also use the absolute rule: "If a rule is not explicitly written, it does not exist."

Can Lovable generate from an image or screenshot?

Yes. You can upload screenshots of existing sites, Figma mockups or Excalidraw sketches directly into Lovable to guide the design. The AI interprets visual inputs and generates code inspired by the provided reference. Always verify the result for accuracy and originality.

Is Lovable worth the paid plan?

If you are actively developing and validating a real product or workflow, it usually is. If you are only casually testing ideas without clear direction, the value is lower. The key question is not the monthly cost, but how quickly it helps you reach a validated product goal compared to alternative approaches.

Concrete Case: SaaS Booking Application

Complete build of a slot-booking tool for a gym — from idea to production deployment, prompt by prompt. Five sequential phases, fully detailed.

FitBook — Booking Application

FitBook — Online class booking for a gym. Members book sessions. Coaches see their schedule. The admin manages everything. Stripe payments included. MVP achievable in 8–12 hours of focused work.

Phase 0 — Immutable Context

PROMPT 0 — STRATEGIC CONTEXT

Plan Mode — zero code

Project: FitBook — slot-booking application for a gym.

Target audience:

- Members (clients): book classes, view their history
- Coaches: see their weekly schedule
- Admin: manage classes, slots, members and payments

Maturity level: MVP — no gamification, no native mobile app

Explicit exclusions:

- No loyalty system in this version
- No real-time chat
- No push notifications (email only)
- No multi-location in the MVP

Hard constraint:

Booking data columns are contractual.
No renaming without explicit validation.

Phase 1 — Architecture & Navigation

PROMPT 1 — APPLICATION STRUCTURE

Pages to create (structure only – no business logic):

```
/ → Public landing (presentation + login/signup)
/login → Sign in
/signup → Member registration
/dashboard → Member dashboard (upcoming bookings)
/classes → Available classes with filters
/booking/:id → Class detail + booking form
/my-account → Profile, history, subscription
/coach/schedule → Coach schedule view (restricted access)
/admin → Admin dashboard
/admin/classes → Class and slot management
/admin/members → Member list and management
```

Navigation:

- Header with logo, main nav, profile button
- Collapsible admin sidebar
- Mobile: bottom navigation for members

Constraints:

- No business logic at this stage
- No real data, structure only
- Do not create form components yet

Validation: generate the file structure and React router only.
Confirm before implementing.

Phase 2 — Supabase Data Schema

Step 2 – Supabase schema for FitBook**Contractual tables and columns (never rename):**

TABLE profiles

```

id: uuid (FK → auth.users, non-null)
role: enum('member','coach','admin') default 'member'
full_name: text (non-null)
phone: text (nullable)
avatar_url: text (nullable)
stripe_customer_id: text (nullable)
created_at: timestamptz default now()

```

TABLE courses

```

id: uuid (PK, default gen_random_uuid())
title: text (non-null)
description: text (nullable)
coach_id: uuid (FK → profiles.id, non-null)
capacity: integer (non-null, default 10)
duration_minutes: integer (non-null)
category: enum('yoga','crossfit','cardio','pilates','strength')
created_at: timestamptz default now()

```

TABLE slots

```

id: uuid (PK)
course_id: uuid (FK → courses.id, non-null)
starts_at: timestamptz (non-null)
ends_at: timestamptz (non-null)
available_spots: integer (non-null)
is_cancelled: boolean default false

```

TABLE bookings

```

id: uuid (PK)
slot_id: uuid (FK → slots.id, non-null)
member_id: uuid (FK → profiles.id, non-null)
status: enum('confirmed','cancelled','waitlist') default 'confirmed'
stripe_payment_intent_id: text (nullable)
booked_at: timestamptz default now()

```

Mandatory RLS policies:

- profiles: SELECT by owner (auth.uid() = id) + admin all
- courses: SELECT public, INSERT/UPDATE coach+admin
- slots: SELECT public, INSERT/UPDATE admin
- bookings: SELECT by member owner + course coach + admin
INSERT by authenticated member only

Generate:

SQL to create tables + RLS policies.
Do not connect to the UI yet.

Phase 3 — Business Logic

PROMPT 3 — EXPLICIT BUSINESS RULES

Step 3 – FitBook business rules (implement strictly)

R1: A member can only book the same slot once.
R2: available_spots decrements by 1 on each confirmed booking.
R3: If available_spots = 0, offer waitlist (status='waitlist').
R4: Member cancellation frees 1 spot AND notifies the 1st member on the waitlist (email via Supabase Edge Function).
R5: Cancelled course (is_cancelled=true) auto-cancels all associated bookings and refunds via Stripe.
R6: A coach can only see their own slots.
R7: Admin can see and modify everything.
R8: No booking allowed less than 2 hours before slot start.

Implement via:

- Supabase Edge Functions for R4 and R5 (server-side logic)
- Postgres triggers for R1 and R2 (database constraint)
- Client-side validation for R8 (UX only, not a security control)

Constraints:

- Do not put R1-R5 only in the frontend
- Always handle error states in the UI

Phase 4 — User Interface

PROMPT 4 — MEMBER UI: CLASS LIST

Step 4a — /classes page (member view)

Style: Modern, energetic — primary colours #6C47FF and #FF5C4D

Components to create:

1. CourseCard: image, title, coach, category, duration, available spots
 - "Full" badge if available_spots = 0
 - "Waitlist" badge on hover if full
 - "Soon" badge if starting in less than 24h
2. CourseFilters: filter by category (chips), by date (datepicker), by coach (dropdown) — all combinable
3. CourseGrid: 3-column desktop, 1-column mobile
4. EmptyState: message if no classes match the filters

Required states:

- Loading: skeleton loader on cards
- Network error: message + "Retry" button
- Empty: illustration + "No classes available"

Constraints:

- Do not modify the data schema
- Do not add out-of-scope features
- Mobile-first, native Tailwind breakpoints only

Phase 5 — Verification & Non-Regression

PROMPT 5 — FITBOOK COMPLIANCE AUDIT

Perform a complete FitBook audit before deployment:

1. DATA

- ✓ All contractual columns present and not renamed?
- ✓ Enums use the defined values?
- ✓ Foreign keys point to the correct tables?

2. SECURITY

- ✓ RLS active on all 4 tables?
- ✓ Can a member access another member's bookings?
- ✓ Can a coach modify courses they don't own?
- ✓ Stripe keys in environment variables?

3. BUSINESS LOGIC

- ✓ Rules R1 to R8 all implemented server-side?
- ✓ available_spots decrement is atomic (Postgres trigger)?
- ✓ Cancellation triggers Stripe refund?

4. UI

- ✓ All states (loading/error/empty) present on every screen?
- ✓ App is responsive on 375px mobile?
- ✓ Status badges display correctly?
- ✓ No booking button shown when slot is full?

Produce a    report for each point.

Realistic Timeline

Phase	Estimated Duration	Estimated Credits
Phase 0 — Context	30 min (Plan Mode)	0
Phase 1 — Architecture	1h	3–5
Phase 2 — Data	1h	4–6
Phase 3 — Business Logic	2h	6–10
Phase 4 — Full UI	3–4h	10–15
Phase 5 — Audit + Fixes	1–2h	5–8
Total MVP	8–12h	28–44 credits

Concrete Case: Professional Website

Website build for an architecture firm — landing page, portfolio, headless CMS blog, qualified contact form, SEO optimised from the very start.

ArchStudio — Architecture Firm

ArchStudio — Showcase site for an independent architecture firm. Goal: attract private and business clients via Google. Blog for SEO content. Contact form with lead qualification. Cloudflare Pages deployment.

Website Starter Prompt

COMPLETE PROMPT — WEBSITE WITH SEO

Project: Showcase site for ArchStudio architecture firm.

Stack:

React 18 + TypeScript + Tailwind + ShadCN
Supabase for blog (headless CMS) and contacts
Deployment: Cloudflare Pages

Pages to create:

/ → Hero + services + featured projects + testimonials + CTA
/projects → Portfolio with filters (residential/commercial/renovation)
/projects/:slug → Project page with photo gallery, description, SEO meta
/blog → Article list with pagination
/blog/:slug → Full article with SEO meta
/contact → Qualified form (project type, budget, location)
/about → Team, values, certifications

SEO — Required on every page:

- Dynamic title: "[Page Title] | ArchStudio Architects"
- Unique meta description 150-160 characters
- og:title, og:description, og:image
- Canonical URL
- JSON-LD Schema.org (Organization on /, Article on /blog/:slug, LocalBusiness on /contact)

Style:

Minimalist premium. Off-white + black + copper (#B87333).
Typography: Georgia for headings, Arial for body.
Subtle scroll animations (no aggressive parallax).

Performance (Core Web Vitals):

- Images in WebP format with lazy loading
- Fonts: system fonts only (no Google Fonts in production)
- Code splitting per route (React lazy imports)

First step only:

The home page with Hero, Services section (3 cards),
and Featured Projects section (3×2 grid).
Real content provided below — no lorem ipsum.

Prompt — Qualified Contact Form

MULTI-STEP FORM WITH LEAD QUALIFICATION

Create the /contact form with these fields:

Step 1 (Project type):

- Type: multiple checkboxes (New Residential / Renovation / Commercial / Other)

Step 2 (Budget and location):

- Estimated budget: select (< \$100k / \$100-300k / \$300k-\$1M / > \$1M)
- Location: text (city + state)
- Desired timeline: select (Urgent < 3 months / 3-12 months / > 1 year)

Step 3 (Contact details):

- First name + Last name: text (non-null)
- Email: email (non-null, format validation)
- Phone: tel (nullable)
- Message: textarea (min 50 characters)

Behaviour:

- Multi-step form with progress bar
- Real-time validation on each field
- Submission → INSERT into Supabase 'leads' table
- Automatic confirmation email via Supabase Edge Function
- Admin notification email (address in environment variable)
- Confirmation page with portfolio CTA

Supabase 'leads' table (contractual columns):

```
id: uuid, project_types: text[], budget_range: text,  
location: text, timeline: text, full_name: text (non-null),  
email: text (non-null), phone: text, message: text,  
created_at: timestamptz, status: enum('new', 'contacted', 'qualified', 'lost')
```

RLS: INSERT public, SELECT admin only.

Prompt — Blog CMS with Supabase

HEADLESS CMS BLOG

Configure the blog using Supabase as a headless CMS.

'posts' table (contractual columns):
id: uuid, slug: text (unique, non-null),
title: text (non-null), excerpt: text (150 chars max),
content: text (Markdown), cover_image_url: text,
author_id: uuid (FK → profiles.id),
published_at: timestamptz (null = draft),
meta_title: text, meta_description: text (160 chars max),
tags: text[], reading_time_minutes: integer

'tags' table: id uuid, name text, slug text unique

Pages to implement:

1. /blog → Paginated article grid (10/page), filter by tag
Meta: "Architecture Blog | ArchStudio"
2. /blog/:slug → Full article
Dynamic meta from meta_title and meta_description
Schema.org Article JSON-LD
Table of contents generated from H2/H3 content headings
Related articles (same tag, max 3)

Exclusions:

- No WYSIWYG editor in MVP (content managed directly in Supabase)
- No comments in this version
- No newsletter in this version

Concrete Case: Internal Business Tool — CRM

Lightweight CRM for a 15-person B2B SMB — client management, sales pipeline, KPI dashboard, CSV import from existing Excel files.

PipeLine — Internal B2B CRM

PipeLine — Internal CRM for a B2B service company. Replaces Excel files. Web access from anywhere. No internal IT required. Roles: Sales Rep (own clients and deals) + Manager (see everything + reports).

CRM Starter Prompt

PROMPT — SMB CRM STARTER

Project: PipeLine – Internal B2B service company CRM.

Users:

- Sales Rep: manages their clients and deals, sees their pipeline
- Manager: sees everything, generates reports

MVP Modules:

1. Clients: profile, interaction history, status
2. Contacts: people linked to a client (multiple contacts/client)
3. Opportunities: amount, probability, pipeline stage, closing date
4. Activities: calls, emails, meetings linked to an opportunity
5. Dashboard: sales rep + manager KPIs

MVP Exclusions:

- No PDF quote generation
- No email sending from the app
- No CSV import in MVP
- No native mobile app

Sales pipeline (fixed stages):

Prospect → Qualified → Proposal → Negotiation → Won/Lost

Start with:

File structure + Supabase schema only.
Present the schema before implementing.

Prompt — Manager KPI Dashboard

PROMPT — DASHBOARD WITH KPIS

Create the manager dashboard /dashboard/manager with these KPIs:

Row 1 — Global metrics (4 cards):

- Total pipeline revenue (sum opportunities.amount where status != 'lost')
- Active opportunities (count, excluding 'won'/'lost')
- Conversion rate (won / total closed × 100, as %)
- Average deal size (won revenue / count won, rolling 90d)

Row 2 — Visual pipeline:

- Read-only Kanban (columns = pipeline stages)
- Each card: client, amount, sales rep, probability
- Card colour by probability:
green > 70%, orange 40-70%, red < 40%

Row 3 — Sales rep table:

Columns: Name | Active Opps | Pipeline | Won (90d) | Conv. %
Sorted by won revenue descending

Rules:

- All data from Supabase (no mock data)
- Filters: period (7d/30d/90d/custom) + sales rep
- Auto-refresh every 5 minutes
- RLS: manager view only

Required states:

- Skeleton loader during loading
- Message if no data in selected period

Prompt — CSV Import of Existing Clients

PROMPT — MIGRATING EXISTING CSV DATA

Add /admin/import page to migrate existing clients from Excel.

Expected CSV format (contractual columns):

company_name, siret, industry, website, city, postal_code,
country (default 'US'), annual_revenue, employee_count,
main_contact_name, main_contact_email, main_contact_phone,
sales_rep_name, status (prospect/client/inactive), notes

Behaviour:

1. Upload CSV file
2. Preview first 5 rows with validation
3. Error detection: invalid email, non-numeric SIRET,
duplicate on company_name + siret
4. Pre-import report: X valid rows, Y errors detected
5. Explicit confirmation required before INSERT
6. Import as a transaction (all or nothing)
7. Log imports in 'import_logs' table

Constraints:

- No import if blocking errors are detected
- No silent data modification from CSV
- Never overwrite an existing client without confirmation

Advanced Security & Architecture Review

Security is not a feature — it is a cross-cutting layer. Here is everything to verify before any production deployment.

The 4 Main Threats to Know

- SQL Injection & RLS Bypass
 - Table without RLS policy = all data readable
 - Unsanitised parameters in queries
 - service_role key exposed client-side
 - Edge Functions without JWT token verification
- Credential Exposure
 - API keys hardcoded in source code
 - .env file committed to GitHub
 - Stripe publishable vs secret key confused
 - Supabase service_role variables client-side
- Authentication Failures
 - Protected routes accessible without token
 - JWT not verified in Edge Functions
 - Persistent session with no expiry
 - No rate limiting on /login
- Client-Side Data Leaks
 - Sensitive data in URL query params
 - console.log of tokens in production
 - Overly verbose API responses
 - LocalStorage for sensitive tokens

Complete RLS Checklist — Supabase

RLS Audit — Check table by table

- ☐ RLS enabled on **every table** — verify in Supabase → Table Editor → RLS
- ☐ SELECT policy: a user can only read their own data (`auth.uid() = user_id`)
- ☐ INSERT policy: only authenticated users can create (`auth.role() = 'authenticated'`)
- ☐ UPDATE/DELETE policy: only on the user's own records
- ☐ Admin role verified via `is_admin()` function or `profiles.role` column
- ☐ Test in incognito mode with a standard user account: no admin access possible
- ☐ Test with 2 different accounts: user A cannot see user B's data
- ☐ Edge Functions: verify JWT with `supabase.auth.getUser(token)` before any processing

Prompt — Security Audit

COMPLETE SECURITY AUDIT

Perform a complete security audit of the application:

1. SUPABASE RLS

- List all tables without an active RLS policy
- Identify overly permissive policies (SELECT without auth)
- Verify service_role is not used client-side

2. CREDENTIALS

- Search for any API key, token or secret hardcoded in code
- Verify .env is in .gitignore
- List all required environment variables

3. ROUTES & AUTHORISATION

- List all protected routes
- Verify each route checks authentication
- Verify admin routes check the admin role

4. EDGE FUNCTIONS

- Do they verify the incoming JWT?
- Do errors return generic messages (no stack trace in production)?

5. SENSITIVE DATA

- Any sensitive data in URLs?
- Any console.log of sensitive data in production?

Report: list of vulnerabilities by level

CRITICAL / HIGH / MEDIUM / LOW + recommended fix.

Recommended Architecture for a Production App

Layer	Responsibility	Security Rule
React Frontend	Display, UX, client-side validation	Never the sole security layer for anything
Supabase Auth	JWT, sessions, OAuth	Verify token in every Edge Function
Supabase RLS	Data access control	Always active, never disabled in production
Edge Functions	Sensitive logic, Stripe, emails	Env vars for all secrets
PostgreSQL	Persistence + constraints	DB constraints as the ultimate safety net



Never use the `service_role` key client-side

The `service_role` key bypasses all RLS policies. It must only exist as a server-side environment variable (Edge Functions). The `anon` key is the only one to use in the frontend.

Complete Architecture Checklist



Architecture — Checklist before production

- ☐ **Separation of concerns** — Business logic is in Edge Functions, not React components
- ☐ **Single source of truth** — Data always comes from Supabase, never from un-synced local state
- ☐ **Error boundaries** — Each critical section has a React ErrorBoundary component
- ☐ **Loading states** — Every async call has a visible loading state
- ☐ **Optimistic updates** — Mutations update UI immediately, then confirm from server
- ☐ **Pagination** — No query ever returns an entire table (always use LIMIT)
- ☐ **DB indexes** — Columns used in WHERE and JOIN are indexed
- ☐ **Versioned migrations** — All schema changes in versioned SQL files
- ☐ **Env variables** — Zero secrets in code, all in environment variables
- ☐ **CORS configured** — Edge Functions only accept authorised origins

Password Management & Authentication

Supabase Auth handles the cryptography — but you must correctly configure policies, flows and recovery mechanisms for a secure, production-ready experience.

Recommended Supabase Auth Configuration

Parameter	Recommended Value	Where
Email confirmation	OFF in dev, ON in prod	Auth → Settings → Email
JWT expiry	3600s (1h) in production	Auth → Settings → JWT
Refresh token rotation	Enabled	Auth → Settings → Security
Minimum password length	8 characters	Auth → Settings → Password
Login rate limiting	Enabled by default	Managed by Supabase automatically
MFA (TOTP)	Optional but recommended for admins	Auth → Multi-Factor Auth

Prompt — Full Auth Flow with Password Reset

COMPLETE AUTHENTICATION FLOW

Implement the complete authentication flow with Supabase:

1. SIGN UP (/signup)

Fields: email, password, confirmation, first name, last name

Validation:

- Email: valid format
- Password: min 8 chars, 1 uppercase, 1 digit, 1 symbol
- Confirmation: must match password

After submission: Supabase confirmation email

Message: "Check your inbox to confirm your account"

2. LOGIN (/login)

Fields: email, password

Option: "Remember me" (persistent 30d session)

"Forgot password" link → /reset-password

Error: generic "Incorrect email or password"

(never reveal whether the email exists)

3. PASSWORD RESET (/reset-password)

Step 1: email only → Supabase sends magic link

Step 2: new password + confirmation (from email link)

Validation: same rules as sign up

Success: redirect to /login with confirmation message

4. ROUTE PROTECTION

ProtectedRoute HOC: checks Supabase session

AdminRoute: checks profiles.role = 'admin'

Redirect to /login if not authenticated

5. SIGN OUT

Button in the profile menu

supabase.auth.signOut() + redirect to /

Clear local state

Constraints:

- Never store password in plain text or hashed locally
- Never log JWT tokens to the console
- Never reveal whether an email is already registered

Prompt — Social Authentication (OAuth)

GOOGLE + GITHUB OAUTH

Add social authentication to /login and /signup:

Providers to enable:

- Google OAuth (prerequisite: app configured in Google Console)
- GitHub OAuth (prerequisite: app configured in GitHub Settings)

UI:

- Separate buttons: "Continue with Google" + "Continue with GitHub"
- "or" separator between OAuth and email form
- Official provider icons (inline SVG)

Behaviour:

- Callback URL: /auth/callback
- On first OAuth login: auto-create profile in 'profiles' table with role='member'
- If email already exists (classic account): offer to link accounts (informational message)

Security:

- /auth/callback verifies the code and creates the session
- Redirect to /dashboard on success
- Redirect to /login?error=oauth_failed on failure

Configure in Supabase:

Authentication → Providers → Google/GitHub → enable

Redirect URL: [https://\[your-project\].supabase.co/auth/v1/callback](https://[your-project].supabase.co/auth/v1/callback)

Prompt — MFA for Admin Accounts

TOTP MFA FOR ADMIN USERS

Implement TOTP two-factor authentication for users with role='admin'.

Enrolment flow:

1. On first admin login: mandatory MFA activation popup
2. Display Supabase MFA QR code (Google Authenticator / Authy)
3. Request TOTP code entry to confirm enrolment
4. Display and force saving of recovery codes

Admin login flow:

1. Email + password (step 1)
2. 6-digit TOTP code (step 2)
3. "Trust this device for 30 days" option

Recovery flow:

- "Use a recovery code" button on the TOTP step
- Each recovery code is single-use (managed by Supabase)

Supabase API:

- `supabase.auth.mfa.enroll()` for the QR code
- `supabase.auth.mfa.challenge()` to initiate verification
- `supabase.auth.mfa.verify()` to validate the TOTP code

Do not touch the authentication flow for non-admin users.

2026 Password Policy Standards

Recommended Password Policy

- Minimum length: 8 characters (12 recommended for admins)
- Complexity: at least 1 uppercase, 1 digit, 1 special character
- Block common passwords (NIST list) — Supabase handles this natively
- Never store the password — only bcrypt hash (Supabase native)
- Session expiry: 1h (JWT) with 30d refresh token
- Refresh token rotation on every use
- Mandatory TOTP MFA for admin accounts in production
- Temporary lockout after 5 failed attempts (managed by Supabase)
- Never reveal whether an email address is already registered

SEO & Organic Search

Lovable generates a React SPA — SEO requires explicit configuration. Here is how to structure your app so that Google can index it efficiently.



SPA Limitation

By default, Google can index SPAs but with a delay. For optimal SEO on a showcase site or blog, consider SSG deployment via Cloudflare Pages. For a SaaS app, SEO mainly concerns the public landing page.

Prompt — Meta Tags & Open Graph

SEO BASELINE — ALL PAGES

Install `react-helmet-async`.

Create a reusable `SEOHead` component with these props:

```
title: string (required)
description: string (150-160 chars)
canonical?: string (full canonical URL)
ogImage?: string (1200×630px image URL)
noIndex?: boolean (true for private pages)
schema?: object (JSON-LD Schema.org)
```

Tags to generate in `SEOHead`:

```
<title>{title} | SiteName</title>
<meta name="description" content={description} />
<link rel="canonical" href={canonical} />
<meta property="og:title" content={title} />
<meta property="og:description" content={description} />
<meta property="og:image" content={ogImage} />
<meta property="og:type" content="website" />
<meta name="twitter:card" content="summary_large_image" />
{noIndex && <meta name="robots" content="noindex,nofollow" />}
{schema && <script type="application/ld+json">
  {JSON.stringify(schema)}
</script>}
```

Private pages (`/dashboard`, `/admin`, `/profile`, `/auth`): `noIndex={true}`

Public pages: all tags populated with real content

Prompt — Automatic XML Sitemap

DYNAMIC SITEMAP.XML VIA EDGE FUNCTION

Create a dynamic sitemap.xml via a Supabase Edge Function GET /sitemap.xml:

1. Static pages (high priority):

- / → priority 1.0, changefreq weekly
- /projects → priority 0.8, changefreq weekly
- /blog → priority 0.8, changefreq daily
- /contact → priority 0.6, changefreq monthly
- /about → priority 0.5, changefreq monthly

2. Dynamic pages (generated from Supabase):

- /projects/:slug → SELECT slug, updated_at FROM projects
WHERE published = true
priority 0.7, changefreq monthly
- /blog/:slug → SELECT slug, published_at FROM posts
WHERE published_at IS NOT NULL
priority 0.6, changefreq never

Output format:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://mysite.com/</loc>
    <lastmod>2026-01-15</lastmod>
    <changefreq>weekly</changefreq>
    <priority>1.0</priority>
  </url>
</urlset>
```

Headers: Content-Type: application/xml

Accessible without authentication (public SELECT on tables)

Prompt — Schema.org JSON-LD Structured Data

SCHEMA.ORG ON KEY PAGES

Add Schema.org structured data on key pages:

HOME PAGE — LocalBusiness:

```
{
  "@context": "https://schema.org",
  "@type": "LocalBusiness",
  "name": "[Company name]",
  "url": "https://[domain]",
  "telephone": "[phone]",
  "address": {
    "@type": "PostalAddress",
    "streetAddress": "[address]",
    "addressLocality": "[city]",
    "postalCode": "[zip]",
    "addressCountry": "US"
  },
  "openingHours": "Mo-Fr 09:00-18:00"
}
```

BLOG ARTICLE — Article:

```
{
  "@context": "https://schema.org",
  "@type": "Article",
  "headline": "[article title]",
  "datePublished": "[published_at ISO]",
  "dateModified": "[updated_at ISO]",
  "author": { "@type": "Person", "name": "[author]" },
  "image": "[cover_image_url]",
  "description": "[excerpt]"
}
```

FAQ PAGE (if applicable) — FAQPage:

```
{
  "@context": "https://schema.org",
  "@type": "FAQPage",
  "mainEntity": [
    { "@type": "Question",
      "name": "[question]",
      "acceptedAnswer": { "@type": "Answer", "text": "[answer]" }
    }
  ]
}
```

Integrate via the SEOHead component with the schema={} prop.

Validate with Google's Rich Results Test.

Prompt — robots.txt

ROBOTS.TXT FOR LOVABLE APPS

Create public/robots.txt with this content:

```
User-agent: *  
Allow: /  
Allow: /blog/  
Allow: /projects/  
Disallow: /dashboard/  
Disallow: /admin/  
Disallow: /auth/  
Disallow: /account/  
Disallow: /api/
```

Sitemap: https://[your-domain]/sitemap.xml

Ensure the file is served statically from the public/ folder and accessible without authentication.

SEO Checklist — Before Going Live

Complete SEO Checklist

- ☐ **Unique title tags** on every page (50–60 characters)
- ☐ **Unique meta descriptions** on every page (150–160 characters)
- ☐ **Canonical URLs** on all pages to prevent duplicate content
- ☐ **Full Open Graph** — og:title, og:description, og:image (1200×630px)
- ☐ **noindex on all private pages**: dashboard, profile, admin, /auth/*
- ☐ **sitemap.xml** submitted in Google Search Console
- ☐ **robots.txt** configured (block /admin, /api, /auth)
- ☐ **WebP images** with descriptive alt attribute on every image
- ☐ **Core Web Vitals**: LCP < 2.5s, FID < 100ms, CLS < 0.1 (test with PageSpeed)
- ☐ **HTTPS active** with automatic HTTP → HTTPS redirect
- ☐ **Clean URLs**: /blog/my-article-title (kebab-case, no numeric IDs)
- ☐ **Schema.org JSON-LD** on key pages (validate with Rich Results Test)

Integrating Stripe Step by Step

Stripe is the most common payment integration in Lovable. Full guide: one-time payments, subscriptions, webhooks and customer portal.



Stripe Architecture in Lovable

All Stripe logic lives server-side (Supabase Edge Functions). The Stripe secret key must never appear in frontend code. The frontend only uses the publishable key for Stripe.js.

Initial Setup Steps

1. Create an account at **stripe.com** and activate test mode
2. In Stripe Dashboard → Developers → API Keys: copy the test secret key
3. In Lovable → Settings → Environment Variables: add `STRIPE_SECRET_KEY`
4. Add the publishable key to frontend code: `VITE_STRIPE_PUBLISHABLE_KEY`
5. Configure the webhook endpoint (see below)

Prompt — One-Time Payment (Checkout Session)

ONE-TIME PAYMENT WITH STRIPE CHECKOUT

1. EDGE FUNCTION create-checkout-session:
 - Receives: { amount_cents: number, product_name: string, user_id: string, metadata: object }
 - Verifies the user JWT
 - Creates a Stripe Checkout Session:
 - mode: 'payment'
 - success_url: https://[domain]/payment/success?session_id={CHECKOUT_SESSION_ID}
 - cancel_url: https://[domain]/payment/cancelled
 - line_items: [{ price_data: { currency: 'usd', product_data: { name: product_name }, unit_amount: amount_cents }, quantity: 1 }]
 - metadata: { user_id, ...metadata }
 - Returns { url: session.url }
 2. FRONTEND — "Pay" button:
 - Calls the Edge Function with fetch
 - Redirects to session.url (window.location.href = url)
 - Loading state during session creation
 3. /payment/success PAGE:
 - Retrieves session_id from query params
 - Calls verify-payment Edge Function to confirm
 - Displays confirmation and updates status in DB
 4. EDGE FUNCTION verify-payment:
 - stripe.checkout.sessions.retrieve(session_id)
 - Verify payment_status === 'paid'
 - Update DB accordingly
 - Return { success: true, details }
- Env vars: STRIPE_SECRET_KEY (server only)
VITE_STRIPE_PUBLISHABLE_KEY (frontend)

Prompt — Recurring Subscriptions

STRIPE SUBSCRIPTIONS WITH TRIAL

Prerequisite: create products and prices in Stripe Dashboard

Starter plan: price_XXXX (\$9.99/month)

Pro plan: price_YYYY (\$29.99/month)

1. subscriptions TABLE (contractual columns):
user_id: uuid (FK profiles), stripe_customer_id: text,
stripe_subscription_id: text, plan: enum('starter','pro'),
status: enum('active','past_due','cancelled','trialing'),
current_period_end: timestamptz
2. EDGE FUNCTION create-subscription-checkout:
 - Create or retrieve the Stripe Customer (via stripe_customer_id)
 - Create Checkout Session mode='subscription'
 - price_id according to chosen plan
 - trial_period_days: 14 (free trial period)
 - success_url + cancel_url
3. EDGE FUNCTION stripe-webhook:
 - Endpoint: /functions/v1/stripe-webhook
 - Verify Stripe signature (STRIPE_WEBHOOK_SECRET)
 - Handle events:
 - customer.subscription.created → INSERT subscriptions
 - customer.subscription.updated → UPDATE subscriptions
 - customer.subscription.deleted → status='cancelled'
 - invoice.payment_failed → status='past_due' + alert email
4. /my-subscription PAGE:
 - Current plan + renewal date
 - "Manage subscription" button → Stripe Customer Portal
 - "Change plan" button
5. EDGE FUNCTION create-portal-session:
 - stripe.billingPortal.sessions.create()
 - return_url: https://[domain]/my-subscription
 - Redirects to Stripe portal (cancellation, card update)

Prompt — Secure Stripe Webhook

STRIPE WEBHOOK — SECURE CONFIGURATION

In Stripe Dashboard → Webhooks → Add endpoint:

URL: `https://[project].supabase.co/functions/v1/stripe-webhook`

Events to listen:

```
payment_intent.succeeded
payment_intent.payment_failed
customer.subscription.created
customer.subscription.updated
customer.subscription.deleted
invoice.payment_succeeded
invoice.payment_failed
```

In Supabase Edge Function `stripe-webhook`:

1. Read signature: `request.headers.get('stripe-signature')`
2. Verify: `stripe.webhooks.constructEvent(body, signature, Deno.env.get('STRIPE_WEBHOOK_SECRET'))`
3. If signature error → return 400 immediately
4. Process event by `event.type`
5. Return 200 in all success cases
(Stripe retries on 5xx or timeout)

Env var: `STRIPE_WEBHOOK_SECRET`

(available in Stripe Dashboard → Webhook → Signing secret)

IMPORTANT:

- Always return 200 even for unhandled events
- Idempotency: check if the event has already been processed
(store `stripe_event_id` in DB to prevent duplicates)

Stripe Test Cards

Card Number	Result	Use Case
4242 4242 4242 4242	✓ Payment succeeds	Happy path test
4000 0000 0000 0002	✗ Card declined	Failure test
4000 0025 0000 3155	⚡ 3D Secure required	Strong auth test
4000 0000 0000 9995	✗ Insufficient funds	Business error test



Going live

Simply replace test keys with live keys in your production environment variables. Never mix test and live keys in the same environment.

Integrating External APIs

Lovable can consume any REST or GraphQL API. Sensitive calls go through Supabase Edge Functions to protect your keys. Here is the complete method, with reusable patterns.

API Call Architecture Principle

Call Type	From	Why
Public APIs without key (weather, exchange rates...)	React frontend directly	No secret to protect
APIs with secret key (OpenAI, Stripe, Twilio...)	Supabase Edge Function	Key never exposed client-side
User OAuth APIs (Google, GitHub...)	Supabase Auth + Edge Function	User token managed server-side
Incoming webhooks (Stripe, Make, GitHub...)	Supabase Edge Function	Secured public endpoint

Prompt — Public REST API (Client-Side)

PUBLIC API WITHOUT SECRET KEY

Integrate the [Name] API (documentation: [URL]) in the [ComponentName] component.

Endpoint: GET https://api.example.com/v1/resource?param={value}

Required headers: { 'Accept': 'application/json' }

Expected response: { data: [...], total: number }

Behaviour:

- Fetch on component mount (useEffect with [param] dependencies)
- Loading state: skeleton loader during request
- Error state: message + "Retry" button (exponential backoff retry)
- Empty state: illustration + descriptive message
- Cache: store result in local state, re-fetch if parameter changes

Never call this API from an Edge Function (it is public).

Do not store the response in Supabase unless explicitly needed.

Prompt — API with Secret Key via Edge Function

SECURED API — GENERIC REUSABLE PATTERN

Create an Edge Function call-[api-name] to call [API Name].

Function structure:

1. Verify user JWT: `const { data: { user } } = await supabase.auth.getUser(token)`
If no user → return 401

2. Validate incoming parameters (never trust the client)

3. API call with secret key:

```
const response = await fetch('https://api.example.com/endpoint', {
  method: 'POST',
  headers: {
    'Authorization': `Bearer ${Deno.env.get('API_SECRET_KEY')}`,
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ ...params })
})
```

4. Handle API errors: `if response.ok === false` → log + return generic error
(never expose the exact error message from the third-party API to the client)

5. Return only the data the frontend needs (not the full response)

Env var: `API_SECRET_KEY`

Frontend: `supabase.functions.invoke('call-[api-name]', { body: params })`

Prompt — OpenAI GPT with Streaming

EDGE FUNCTION OPENAI GPT-4O WITH SSE STREAMING

Edge Function openai-chat:

1. Verify user JWT
2. Rate limiting: max 10 calls per user per hour
(store in table 'api_rate_limits': user_id, endpoint, count, reset_at)
3. OpenAI call with streaming:

```
const stream = await openai.chat.completions.create({
  model: 'gpt-4o',
  messages: [
    { role: 'system', content: Deno.env.get('SYSTEM_PROMPT') },
    { role: 'user', content: userMessage }
  ],
  stream: true,
  max_tokens: 1000
})
```
4. Return stream to frontend (Server-Sent Events)

Frontend — AIChat component:

- User input + Send button
- Display stream in real time (token by token)
- "Stop" button to cancel generation
- Conversation history in local state (not in DB for MVP)

Env vars: OPENAI_API_KEY, SYSTEM_PROMPT

Prompt — GraphQL API with Apollo Client

GRAPHQL QUERIES FROM LOVABLE

Integrate the [Name] GraphQL API (endpoint: `https://api.example.com/graphql`).
Install: `@apollo/client` (or native `fetch` for a lighter footprint)

Configuration in `src/lib/apollo.ts`:

```
const client = new ApolloClient({
  uri: 'https://api.example.com/graphql',
  cache: new InMemoryCache(),
  headers: { 'Authorization': `Bearer ${token}` }
})
```

Query to implement:

```
query GetItems($filter: String, $limit: Int) {
  items(filter: $filter, limit: $limit) {
    id, name, description, createdAt
  }
}
```

Usage in [ComponentName]:

```
const { data, loading, error } = useQuery(GET_ITEMS, {
  variables: { filter, limit: 20 }
})
```

If the API key is secret: use an Edge Function proxy
that makes the GraphQL request server-side.

Prompt — Secure Incoming Webhooks with Idempotency

GENERIC SECURE WEBHOOK ENDPOINT

Edge Function webhook-[service]:

1. Verify signature:

```
const signature = req.headers.get('[service]-signature')
const isValid = await verifySignature(body, signature, Deno.env.get('WEBHOOK_SECRET'))
If invalid → return 401 immediately (do not process)
```

2. Idempotency: check if event_id already processed

```
SELECT id FROM processed_webhooks WHERE event_id = event.id
If already processed → return 200 without reprocessing
```

3. Route by event.type:

```
switch(event.type) {
  case 'event.created': await handleCreated(event.data); break;
  case 'event.updated': await handleUpdated(event.data); break;
  default: console.log('Unhandled event:', event.type);
}
```

4. Log in 'webhook_logs' table: event_id, type, status, payload, processed_at

5. Always return 200 for known events

(returning 4xx or 5xx triggers retries from the service)

Env var: WEBHOOK_SECRET

Prompt — Retry with Exponential Backoff

ROBUST_FETCHWITHRETRY UTILITY

```
async function fetchWithRetry(url, options, maxRetries = 3) {
  for (let attempt = 0; attempt <= maxRetries; attempt++) {
    try {
      const response = await fetch(url, options)

      // Retry only on server errors (5xx) and rate limiting (429)
      if (response.status === 429 || response.status >= 500) {
        if (attempt === maxRetries) throw new Error(`API failed after ${maxRetries} retries`)
        const delay = Math.pow(2, attempt) * 1000 + Math.random() * 500 // exponential + jitter
        await new Promise(resolve => setTimeout(resolve, delay))
        continue
      }

      return response // Return immediately for 2xx, 4xx (except 429)
    } catch (networkError) {
      if (attempt === maxRetries) throw networkError
      await new Promise(resolve => setTimeout(resolve, Math.pow(2, attempt) * 1000))
    }
  }
}
```

Use in all Edge Functions that call critical external APIs.

Never retry 4xx errors (except 429) – those are client-side errors.

Prompt — Make.com Integration via Webhook

MAKE.COM AUTOMATION TRIGGER

When a user submits the contact form, trigger a Make.com webhook to automate lead qualification.

1. Create Edge Function send-to-make:
 - Receives form data
 - Makes a POST to MAKE_WEBHOOK_URL (env var)
 - JSON payload with all form fields
 - Handles errors gracefully (do not block submission if Make fails)
2. In the contact form:
 - Call send-to-make in parallel with the Supabase INSERT
 - Make result is not awaited (fire-and-forget)

Env var: MAKE_WEBHOOK_URL
(available in Make.com → Webhooks → Custom webhook → URL)

This pattern works identically for Zapier and n8n webhooks.

Prompt — File Upload with Supabase Storage

AVATAR UPLOAD ON PROFILE PAGE

Implement avatar upload on the profile page.

1. Supabase Storage configuration:

Bucket 'avatars': public, limited to jpg/png/webp, max 5MB

2. AvatarUpload component:

- "Change photo" button with hidden file input
- Image preview before upload
- Progress bar during upload
- Client-side compression (browser-image-compression library)
- Filename: {user_id}/avatar.{ext}

3. After upload:

```
UPDATE profiles SET avatar_url = [Supabase Storage public URL]
WHERE id = auth.uid()
```

4. Storage RLS policy:

```
SELECT: public (avatars are visible to all)
INSERT/UPDATE/DELETE: auth.uid()::text = (storage.foldername(name))[1]
```

Error handling:

- File too large: "File too large (max 5MB)"
- Unsupported format: "Unsupported format (JPG, PNG, WebP)"
- Network error: "Upload failed, please try again"

Prompt — Google Analytics 4 Integration

GA4 WITH EVENT TRACKING

Integrate Google Analytics 4 in the application.

1. Add in App.tsx (via react-helmet-async):
gtag.js script with MEASUREMENT_ID from env var
(VITE_GA_MEASUREMENT_ID)
2. Create useAnalytics() hook with these methods:
trackPageView(path: string)
trackEvent(name: string, params: object)
trackConversion(value: number, currency: string)
3. Auto-trigger:
trackPageView() on every React Router route change
4. Events to track manually:
trackEvent('sign_up', { method: 'email'|'google'|'github' })
trackEvent('login', { method: 'email'|'google' })
trackEvent('contact_form_submit', { project_type, budget })
trackEvent('checkout_started', { plan, amount })
trackEvent('purchase', { transaction_id, value, currency })
5. Private pages (/dashboard, /admin):
Do not track internal actions

GDPR: display a cookie consent banner (required in EU)
before initialising gtag.

Env var: VITE_GA_MEASUREMENT_ID=G-XXXXXXXXXX

API Integration Checklist



Before deploying an API integration

- ☐ API key stored in environment variable — never in code or committed
- ☐ Calls with secret keys go through an Edge Function (never client-side)
- ☐ JWT verification in every Edge Function before the API call
- ☐ Rate limiting on expensive APIs (OpenAI, image generation...)
- ☐ API errors returned as generic messages (no stack trace to client)
- ☐ Retry with backoff on 5xx and 429 errors
- ☐ Idempotency for webhooks (event_id stored to avoid duplicates)
- ☐ All UI states handled: loading, error, empty
- ☐ Tested in sandbox/test mode before activating production

Extensions & Available Integrations

25+ services referenced by category, with integration complexity, free tier and recommended prompts.



Analytics & Tracking

GA4

Google Analytics 4
Prompt: "Integrate GA4 with gtag.js via react-helmet-async. Track signup, checkout, cta_click events. noindex on private pages." Free and unlimited.

PLB

Plausible Analytics
Privacy-first alternative to GA4, GDPR-native by default, no cookies. 1 script line. Ideal for European apps. From \$9/month.

MXP

Mixpanel
Advanced product analytics: funnels, retention, cohorts. Prompt: "Track signup, first_action, conversion. Use Mixpanel SDK via react-helmet."

CLR

Microsoft Clarity
Heatmaps and session recordings. Free and unlimited. 1 script to inject. Perfect for UX debugging with zero effort.



Transactional Emails

RSN

Resend (recommended)
Modern email API, native Supabase integration. 100 emails/day free. Prompt: "Configure Resend in an Edge Function for signup confirmation and admin notifications."

SG

SendGrid
Historical leader. 100 emails/day free. Drag-and-drop templates. Node.js SDK compatible with Supabase Edge Functions Deno runtime.

LPS

Loops
Transactional + marketing email for SaaS. Onboarding sequences, segments based on Supabase events. Excellent developer experience.

MG

Mailgun
Excellent B2B deliverability. Simple API. SDK compatible with Supabase Edge Functions (Deno). Good reputation for cold email avoidance.



Artificial Intelligence

OAI

OpenAI API

GPT-4o, DALL-E, Whisper, Embeddings. Always via Edge Function. Prompt: "openai-chat Edge Function with SSE streaming and per-user rate limiting."

ANT

Anthropic Claude API

Claude Sonnet 4 and Opus. Same Edge Function pattern as OpenAI. Key in ANTHROPIC_API_KEY env var. Excellent for long-context tasks.

PGV

pgvector (Supabase)

PostgreSQL vector search extension, natively available in Supabase. Ideal for RAG chatbots on your own documents. Combine with OpenAI Embeddings.

RPL

Replicate

Image generation (FLUX, Stable Diffusion) via API. Pay-per-use. Prompt: "Edge Function generate-image, store result URL in Supabase Storage."



Notifications & Communication

TWL

Twilio SMS

Send SMS from an Edge Function. Custom OTPs, appointment reminders, critical alerts. Very simple REST API. Test credits available.

SLK

Slack Webhooks

Notify your team of key events (new signup, error, payment). Incoming webhook set up in 2 minutes. Fire-and-forget from Edge Function.

ISB

Supabase Realtime

Native Supabase. Real-time notifications via WebSocket. Prompt: "Subscribe the NotifBell component to changes in the notifications table via Supabase Realtime."

OSG

OneSignal (Push)

Web push notifications for desktop and mobile. JS SDK + Edge Function to trigger. Generous free tier. Good cross-browser support.



No-Code Automation

MK

Make.com

1,000+ app connections. Trigger via Edge Function webhook. Prompt: "Edge Function send-to-make: fire-and-forget POST to MAKE_WEBHOOK_URL after each new lead or event."

N8N

n8n (open source)

Self-hosted Make.com alternative. Full data control, no per-operation cost. Same webhook integration pattern as Make.

ZAP

Zapier

Connect Supabase webhooks to Gmail, HubSpot, Salesforce, and 6,000+ apps. 100 tasks/month on free tier.

ART

Airtable API

Supabase ↔ Airtable sync via Edge Function. Useful when a non-technical team manages records in Airtable that need to feed your app.

Storage & Files

SBS

Supabase Storage

Native, always available. Prompt: "Create bucket 'uploads', user profile upload, 5MB limit, jpg/png/webp only, RLS policy filtering by user_id."

CDY

Cloudinary

On-the-fly image optimisation and transformation (resize, crop, auto-WebP). 25GB free. Direct JS SDK — no Edge Function needed for public images.

UPD

UploadThing

Modern file upload library built for React apps. Type enforcement, size limits, callbacks. Full integration in under 30 minutes.

R2

Cloudflare R2

S3-compatible object storage with no egress fees. Best for high-volume file storage. Configure via Edge Function using AWS SDK S3-compatible API.

Maps & Geolocation

MBX

Mapbox

Customisable interactive maps. Prompt: "Integrate Mapbox GL JS to display [points from locations table] on a map with clickable markers." 50k free requests/month.

GMP

Google Maps API

Address autocomplete, geocoding, route calculation. Key in env var. \$200/month free credit. Prompt: "Add Places Autocomplete on the address field."

Summary Table

Service	Category	Complexity	Free Tier	Integrated Via
Stripe	Payment	Medium	Test only	Edge Function
Resend	Email	Easy	100/day	Edge Function
OpenAI	AI	Medium	No	Edge Function
Google Analytics 4	Analytics	Easy	Yes	JS Script
Plausible	Analytics	Very easy	From \$9/mo	JS Script
Supabase Realtime	Real-time	Easy	Included	Native SDK
Make.com	Automation	Easy	1000 ops/mo	Webhook
Twilio	SMS	Medium	Test credits	Edge Function
Mapbox	Maps	Medium	50k req/mo	JS SDK
Cloudinary	Media	Easy	25GB	JS SDK
Replicate	AI Generation	Medium	Pay-per-use	Edge Function

Additional Advanced Prompts

Reference prompts for the most common advanced use cases — copy, adapt, iterate.

Performance & Optimisation

CORE WEB VITALS OPTIMISATION

Optimise application performance to improve Core Web Vitals without touching business logic.

Actions to perform:

1. Code splitting: add `React.lazy()` and `Suspense` on all routes except the home page
2. Images: replace `<img>` tags with a `LazyImage` component with `loading="lazy"` and WebP format
3. Fonts: remove Google Fonts imports, use system fonts (`font-family: system-ui, sans-serif`)
4. Bundle size: audit imports and replace global imports with targeted ones (e.g. `import { Button } from '@shadcn/ui'`)
5. Memoisation: identify components with unnecessary re-renders and add `React.memo()` where relevant

Do not modify:

- Business logic or API calls
- Existing styles
- Route structure

Expected output: `console.log` of bundle size before and after.

INFINITE SCROLL PAGINATION

Implement infinite scroll on [ListName].

Behaviour:

- Load 20 items on first render
- Detect scroll to bottom (Intersection Observer)
- Auto-load 20 more items
- "Loading..." indicator during fetch
- "All items loaded" message at end of list
- No "Load more" button (scroll-only)

Supabase query:

```
.from('table').select('*')  
.range(page * 20, (page + 1) * 20 - 1)  
.order('created_at', { ascending: false })
```

Local state:

```
items: [] (cumulative list)  
page: 0 (current page)  
hasMore: true (more items available)  
loading: false
```

Only modify [ListName] component.

REAL-TIME SEARCH WITH DEBOUNCE

Add real-time search on [PageName].

Behaviour:

- Search input with placeholder "Search..."
- 300ms debounce before Supabase query
- Search across columns: [col1], [col2], [col3]
(use `ilike '%term%'` for each column)
- Highlight search term in results
- Counter "X result(s) found"
- Clear button (x) if field is not empty
- Empty state: "No results for '[term]'"

Supabase query:

```
.or('col1.ilike.%term%', 'col2.ilike.%term%')
```

Performance:

- Cancel previous request if a new one is fired (AbortController)
- Don't search if fewer than 2 characters

Only modify [PageName] component.

Add an "Export CSV" button on [PageName].

Behaviour:

- Fetch ALL filtered data (not just the current page)
- Columns to export: [col1: "Label 1"], [col2: "Label 2"]...
- UTF-8 encoding with BOM for Excel compatibility
- Filename: [export-name]-YYYY-MM-DD.csv
- Escape quotes: `cell.replace(/"/g, '"')`
- Confirmation toast after download

CSV generation code:

```
const BOM = '\uFEFF'
const headers = ['Label 1', 'Label 2', ...]
const rows = data.map(item => [item.col1, item.col2, ...])
const csv = BOM + [headers, ...rows]
  .map(row => row.map(cell =>
    `${String(cell ?? '').replace(/"/g, '"')}`
  )).join(',')).join('\n')
```

RLS: use the same filters as the current view.

Admin only if the table contains sensitive data.

Accessibility & Dark Mode

Implement dark mode with Tailwind CSS.

1. `tailwind.config.js`:
 - darkMode: 'class' (based on 'dark' class on <html>)
2. ThemeProvider context:
 - Read system preference: `window.matchMedia('prefers-color-scheme: dark')`
 - Persist choice in localStorage (`'theme': 'light'|'dark'|'system'`)
 - Apply 'dark' class on `document.documentElement`
3. Header toggle:
 - Sun/moon icon based on current theme
 - 3 options in a dropdown: Light / Dark / System
4. Component update:
 - Replace hardcoded colours with Tailwind dark-aware variables:
 - `bg-white dark:bg-gray-900`
 - Check text contrast in dark mode (WCAG AA)

Do not modify business logic.

Visually test on all main pages before validating.

Improve application accessibility (WCAG AA level).

Actions to perform:

1. Images: add descriptive alt on all `<img>`
(`alt=""` for decorative images)
2. Forms: associate each `<label>` with its `<input>`
via `htmlFor/id`
3. Buttons: ensure no button has only an icon without `aria-label`
4. Keyboard navigation: verify Tab/Shift+Tab
follows logical order
5. Colours: verify contrast ratio $> 4.5:1$ for body text
6. Visible focus: never remove outline without an alternative
7. Headings: H1 $\rightarrow$ H2 $\rightarrow$ H3 hierarchy respected on each page
8. Skip link: add "Skip to main content" as first
focusable element on each page

Test with: axe DevTools (free Chrome extension)

Only modify HTML/ARIA attributes, not business logic.

Add multilingual support EN/FR with `react-i18next`.

Install: `react-i18next`, `i18next`, `i18next-browser-languagedetector`

File structure:

- `src/locales/en/common.json` $\rightarrow$ English texts
- `src/locales/fr/common.json` $\rightarrow$ French texts

`i18n.ts` configuration:

- Auto-detect browser language
- Fallback to 'en' if language not supported
- Persist choice in `localStorage`

Text migration:

- Replace all hardcoded UI text with `t('key')`
- Keep Supabase database content as-is
(do not translate data stored in the DB)

5. Language toggle in header:

Flags EN / FR, persisted in `localStorage`

Start with components:

Header, Footer, LoginPage, SignupPage

Do not migrate admin pages in this step.