Most people think prompting is just typing a request into AI and hoping for the best. Wrong.

The difference between getting a mediocre response and having AI build entire workflows for you comes down to how you prompt. Whether you're a developer or a non-technical user, mastering prompt engineering can help you:

Automate repetitive tasks Debug faster with AI-generated solutions Build and optimize workflows effortlessly And the best part? You don't need to be an expert. With the right prompting techniques, you can unlock AI's full potential in Lovable, make.com, and n8n—without wasting time on trial and error.

Let's dive in.

TL;DR: Effective prompting matters: Structure your prompts to save hours of troubleshooting. Meta prompting: Use AI itself to refine your prompts for better accuracy. Reverse meta prompting: Save debugging sessions to optimize future requests. Automation tools: Extend Lovable's capabilities with APIs using make.com and n8n. Chat mode vs. default mode: When to use each for debugging and iteration. Handling webhooks: Automate Lovable applications with powerful integrations. Why Prompting Is Critical for AI Development Basics of Prompting

Unlike traditional coding, AI applications rely on structured communication. Providing AI with clear context and constraints ensures high-quality output. In a Lovable expert session at Lovable, Mark from Prompt Advisors demonstrated how developers and non-technical users can enhance their AI prompting techniques to build faster, debug smarter, and automate complex workflows.

Understanding the AI's "Mindset" AI models, including those powering Lovable, do not "understand" in a human way—they predict responses based on patterns. To guide them effectively:

Be explicit: Instead of "build a login page," specify "create a login page using React, with email/password authentication and JWT handling." Set constraints: If you need a specific tech stack (e.g., Supabase for authentication), state it clearly. Use formatting tricks: AI prioritizes the beginning and end of prompts—put important details upfront. Mastering Prompting: The Four Levels four-tiered approach of prompting

1. Training Wheels Prompting A structured, labeled approach for clear AI instructions:

Context

Task

Guidelines

Constraints

Example:

You are a world-class prompt engineer. Write me a prompt that will generate a full-stack app taking an input of name, number, and company, and generate a company report. 2. No Training Wheels More conversational prompts while maintaining clarity.

3. Meta Prompting Leverage AI to refine your prompts:

Rewrite this prompt to be more concise and detailed: 'Create a secure login page in React using Supabase, ensuring role-based authentication.' 4. Reverse Meta Prompting When debugging, have AI document the process for future use:

Summarize the errors we encountered while setting up JWT authentication and how they were resolved. Create a detailed prompt I can use next time. Prompt Library Enhance Prompt The quality of your prompts significantly influences the output of AI. This is the essence of effective prompting: the more refined your prompt, the higher the quality of the output you receive. A comprehensive and well-organized prompt can save you both credits and time by reducing errors. Therefore, these steps are definitely worth considering:

Provide as much details as you can in the input field. Use the "Select" feature to precise edit your component. Enhance your prompt with the experimental "Chat mode". Starting a new project Use this proven structure for starting a new project:

Start with "I need a [type] application with:" Elaborate on tech stack - including Frontend, styling, Authorization and Database. Elaborate on core features including main and secondary features. Then direct the AI to start somewhere like: "Start with the main page containing: [Detailed page requirements]". However, we consistently recommend that users begin with a blank project and gradually build upon it. This approach allows the AI to grasp the fundamental concepts effectively before delving into the specifics.

Diff & Select Whenever you request Lovable to implement a particular change in any file, it will rewrite the entire file or modify the existing content. To ensure that the AI only updates relevant files, provide clear instructions. This approach encourages the AI to edit only the necessary sections, resulting in minimal changes to just a few lines of code. By doing so, you can reduce loading times and prevent error loops.

An effective prompt I've applied previously when adjusting an existing feature is:

Implement modifications to the feature while ensuring core functionality, other features, and processes remain unaffected. Evaluate its behavior and dependencies to identify potential risks, and discuss any concerns before moving forward. Conduct thorough testing to verify there are no regressions or unintended consequences, and highlight any out-of-scope changes for review. Exercise caution—take a moment to pause if uncertain. Lock Files Lovable currently lacks a built-in file locking system. However, you can guide the AI with slight modifications to your prompts. Just include this instruction in each prompt: "Please refrain from altering pages X or Y and focus changes solely on page Z."

You can also try this prompt if you are updating an existing feature without the intention of modifying something sensible:

This update is quite delicate and requires utmost precision. Carefully examine all dependencies and potential impacts before implementing any changes, and test systematically to guarantee nothing is disrupted. Steer clear of shortcuts or assumptions—take a moment to seek clarification if you're unsure. Precision is crucial. Design Designing something on Lovable is effective as Lovable already has great taste ;) Nevertheless, those below prompts can help you improve those design implementations:

UI Changes:

Make solely visual enhancements—ensure functionality and logic remain unaffected. Gain a comprehensive understanding of how the existing UI interacts with the app, ensuring that logic, state management, and APIs stay intact. Conduct extensive testing to verify that the app operates precisely as it did before. Cease all actions if there is any uncertainty regarding potential unintended consequences. Optimize for Mobile:

Enhance the app's mobile experience while preserving its existing design and functionality. Assess the layout and responsiveness to pinpoint essential modifications for smaller screens and touch inputs. Develop a comprehensive plan before making any code changes, and conduct thorough testing across various devices to guarantee the app operates as intended. If uncertain, take a moment to consider and suggest potential solutions. Responsiveness and Breakpoints Prompt:

Make certain that all designs are completely responsive at every breakpoint, adopting a mobile-first strategy. Apply contemporary UI/UX best practices to define how components should adjust for varying screen sizes, utilizing ShadCN and Tailwind's standard breakpoints. Steer clear of custom breakpoints unless specifically requested. Planning:

Before editing any code, create a phased plan for implementing responsiveness. Start with the largest layout components and progressively refine down to smaller elements and individual components. Ensure the plan includes clear steps for testing responsiveness across all breakpoints to maintain consistency and a seamless user experience. Share the plan for review before proceeding. Before making any code edits, develop a structured plan for implementing responsiveness. Begin with the largest layout components and gradually work down to smaller elements and specific components. Ensure that the plan outlines definitive steps for testing responsiveness at all breakpoints to guarantee consistency and a smooth user experience. Present the plan for feedback before moving forward.

Knowledge base Knowledge prompt on Lovable

Providing detailed context about your project is crucial, especially early on in the project. What is the project's purpose? What does the user flow look like? What tech stack are you utilizing? What is the scope of work? At Lovable, we refer to this as the "Knowledge Base," and it can be easily found in your project settings.

Creating a solid framework for AI ensures it operates effectively and adheres to your outlined plan with every prompt you provide. Incorporate these elements within your project:

Project Requirements Document (PRD): This section is crucial for any AI coding project. It outlines a comprehensive summary covering essential elements such as the introduction, app flow, core features, tech stack, and the distinctions between in-scope and out-of-scope items. Essentially, it serves as your project's roadmap, which you can present to AI coding models.

Application or user flow: This clarity will aid the AI model in understanding the connections between pages and processing all features and limitations effectively.

Users begin their experience on the landing page, where they can click the sign-up button to register with Google, subsequently accessing the dashboard. The dashboard comprises X sections. Tech stack: This section must encompass all technical specifics regarding the project, such as the Frontend Tech Stack, Backend Tech Stack, API Integrations, Deployment Instructions, and any other open-source libraries you plan to utilize. This information will facilitate the AI model's understanding of which packages and dependencies to install.

Frontend guidelines: This section should outline your project's visual appearance in detail: Design Principles, Styling Guidelines, Page Layout, Navigation Structure, Color Palettes, and Typography. This serves as the aesthetic foundation of your project. The clearer your explanations, the more visually appealing your application will become.

Backend structure: This section will explain to AI model about: Backend Tech like Supabase, User Authentication, Database Architecture, Storage buckets, API Endpoints, Security measures, Hosting Solutions. This is the main brain of your project. Your app will fetch and display data from your backend.

Once you initiate the project with the initial prompt, be sure to incorporate this Knowledge Base to reduce errors and prevent AI hallucinations. Additionally, you can prompt the AI with:

Before you write any code, please review the Knowledge Base and share your understanding of my project. Utilize the "Chat mode" for this task to ensure that no modifications are made to your projects while you are providing guidance.

Mobile First The issue (and somewhat hidden truth) is that most developers prioritize desktop design simply because it looks better on a large, vibrant screen. However, the reality is that we should have been focusing on mobile-first design for years now.

A great prompt that was shared by a Champion on Discord:

Always make things responsive on all breakpoints, with a focus on mobile first. Use modern UI/UX best practices for determining how breakpoints should change the components. Use shadcn and tailwind built in breakpoints instead of anything custom, unless the user prompts for custom breakpoints directly. Optimize the app for mobile without changing its design or functionality. Analyze the layout and responsiveness to identify necessary adjustments for smaller screens and touch interactions. Outline a detailed plan before editing any code, and test thoroughly across devices to ensure the app behaves exactly as it does now. Pause and propose solutions if unsure. But if you're already far along into your project, you can fix this by telling it to update things to be responsive starting with the largest layout components down to the smallest. Then get to the individual components.

Details When working with Lovable, it’s crucial to provide the AI with clear and specific requests. Rather than simply saying, "move the button to the right," try stating, "in the top header, shift the sign-up button to the left side of the page, ensuring the styling remains consistent." The more precise your instructions are, the fewer errors you'll encounter, and you'll save on credits!

Basically, I always suggest adding instructions on how you want Lovable to approach every task. My example:

Key Guidelines: Approach problems systematically and articulate your reasoning for intricate issues. Decompose extensive tasks into manageable parts and seek clarification when necessary. While providing feedback, elucidate your thought process and point out both challenges and potential improvements. Step by Step Avoid assigning five tasks to Lovable simultaneously! Doing so may lead the AI to create confusion. Here's a better approach:

Start with Front design, page by page, section by section. The plug backend using Supabase as Lovable integration is natively built! Then, refine the UX/UI if needed. This step-by-step process enables AI to concentrate on one task at a time, reducing the likelihood of errors and hallucinations.

Don't loose components You can also implement this after significant changes and following a series of minor adjustments. This practice has been invaluable in maintaining project consistency and preventing sudden loss of components. Regularly refer to our filesExplainer.md document to ensure we accurately record changes in code and components, keeping our file structure organized and up to date.

Refactoring Refactoring is essential to your development lifecycle within Lovable. It is often suggested by the AI to minimize the loading time and errors. Here are great prompts you can use:

Refactoring After Request Made by Lovable:

Refactor this file while ensuring that the user interface and functionality remain unchanged—everything should appear and operate identically. Prioritize enhancing the structure and maintainability of the code. Carefully document the existing functionality, confirm that testing protocols are established, and implement changes gradually to prevent risks or regressions. If you are uncertain at any point, pause the process.
Refactoring Planning:

Develop a comprehensive plan to refactor this file while keeping the user interface and functionality entirely intact. Concentrate on enhancing the code's structure, readability, and maintainability. Start by meticulously documenting the existing functionality and pinpointing potential areas for enhancement. Implement rigorous testing protocols to ensure consistent behavior throughout the entire process. Move forward incrementally, minimizing risks and avoiding regressions, and take breaks for clarification whenever uncertainties emerge. Comprehensive Refactoring:

Develop a comprehensive plan for a site-wide codebase review aimed at identifying segments that would benefit from refactoring. Concentrate on highlighting areas where the code structure, readability, or maintainability can be enhanced, ensuring the user interface and functionality remain unchanged. Rank the most essential files or components based on their significance and usage frequency. Thoroughly document your findings, detailing suggested improvements and the potential effects of each change. Ensure that any proposed refactoring efforts are incremental, low-risk, and supported by rigorous testing to prevent regressions. Circulate the plan for feedback prior to implementation. Post Refactoring:

Conduct a detailed post-refactor review to verify that no issues were introduced throughout the refactoring process. Confirm that both the UI and functionality retain their original integrity following the modifications. Execute an extensive suite of tests—including unit, integration, and end-to-end tests—to ensure all features operate as intended. Evaluate the app’s behavior against the documented pre-refactor specifications and highlight any discrepancies for prompt evaluation. Make certain all updates are stable and align with the project’s requirements prior to completion. Codebase Structure Audit Prompt:

Perform a comprehensive regression and audit of the codebase to determine if its architecture is clean, modular, and optimized. Identify any files, components, or logic that are mislocated, not correctly placed, or could benefit from enhanced organization or modularity. Evaluate whether the separation of concerns is distinct and if functionality is aggregated logically and efficiently. Deliver a detailed report outlining improvement areas, such as files that need restructuring, overly coupled code, or chances to simplify and streamline the organization. Break down the actionable enhancements into manageable steps, arranged in the order you deem most effective for implementation. Ensure the analysis is comprehensive, actionable, and adheres to best practices for a maintainable and clean codebase. Refrain from editing any code. Folder Review:

Conduct a thorough examination of the folder [Folder Name] along with all its subfolders and files. Assess each element to understand its function and how it enhances the overall performance of the application. Offer a detailed explanation of each item's role, while pinpointing any redundancies, obsolete files, or opportunities for improved organization. The objective is to tidy up and optimize this folder, so include suggestions for deleting, merging, or reorganizing items as needed. Ensure your analysis is all-encompassing, practical, and outlines a clear strategy for achieving a more organized and efficient folder structure. Post Restructuring Cleanup:

Ensure all routing and file imports are thoroughly updated and functioning as intended following the codebase restructuring. Validate that components, pages, and APIs reflect the accurate paths found in the new folder organization. Confirm that nested routes are appropriately configured and linked within the router setup and that dynamic or lazy-loaded routes adhere to the new framework. Assess that shared utilities, services, and assets are imported correctly to prevent breaking existing dependencies. Revise hardcoded paths in components, redirects, or navigation links to correspond with the new routing logic. Conduct navigation tests to identify any broken links, missing files, or 404 errors, and pinpoint any missing or redundant imports, extraneous files, or potential improvements for maintainability and scalability in the routing configuration. Codebase Check for Refactoring:

Perform a thorough audit of the codebase to assess its structure and organization. Evaluate whether files, components, and logic are effectively separated based on their functionality and purpose. Identify any instances of misplaced code, excessive coupling, or areas that could benefit from improved separation of concerns. Deliver a comprehensive report on the overall health of the structure, offering specific recommendations for enhancing file organization, consolidating related functionalities, or refactoring to align with industry best practices. Ensure that the analysis is detailed and emphasizes concrete improvements without implementing any direct changes. Stripe Stripe seamlessly integrates with Lovable and can be set up with minimal effort. However, there are several factors that may hinder Stripe's functionality:

Initiate a Stripe connection in test mode using the configuration detailed below: Utilize the specified product and pricing details: Product IDs are [Your Product IDs], with a pricing model of [One-time or Subscription]. Set the webhook endpoint to [Your Webhook Endpoint]. Style the frontend payment form as follows: [Describe desired payment form or provide an example]. Upon successful payment, redirect users to [Success Redirect URL], and for canceled payments, redirect them to [Cancel Redirect URL]. Please refrain from altering any code, and ensure that I have included all necessary information to effectively start with Stripe. *Disclaimer: Use your Stripe Secret Key and Webhook Signing Secret securely in the Supabase Edge Function Secrets and avoid including them in the prompt for safety.*

Ask for help Avoid the tendency to rely on Lovable for every small change. Many minor adjustments can be made directly within your code, even if you aren't a professional engineer. If you need assistance, feel free to consult ChatGPT or Claude for help. Utilize the browser's Inspect tool to identify the elements you want to modify. You can experiment with changes at the browser level, and if you're pleased with the outcome, make those adjustments in the code. This way, you won't need to involve Lovable at all.

While I'm not an engineer, having a basic understanding of coding significantly aids my progress. Utilizing tools like GitHub and Sonnet, I frequently implement enhancements beyond Lovable, allowing me to reserve my prompts for more complex tasks.

Debugging in Lovable Debugging is an integral part of the Lovable experience, and mastering this debugging flow can significantly reduce frustration—especially when clicking the “Try to Fix” button, which does not count as credits.

Fix bug in Lovable

Chat Mode vs. Default Mode The new "Chat mode" is excellent for fostering creativity and generating ideas. Begin by outlining your concept, as this could be the most critical step. Visualizing screens, features, and layouts in your mind isn’t as effective for tracking changes.

A traditional scenario of using the "Chat mode" is:

Default Mode: High-level feature creation.

Review the app and tell me where there is outdated code. Chat Mode: Troubleshooting—ask AI to analyze errors before making changes. Go to your account settings and enable Labs feature.

Follow this plan and act on all those items debugging workflow with AI

I think I've read these below super prompts from an X user then found it back on Discord:

Perform a comprehensive regression and audit of the codebase to determine if its architecture is clean, modular, and optimized. Pinpoint any files, components, or logic that are incorrectly placed, not allocated to suitable files, or require improved organization or modularity. Evaluate whether the separation of concerns is distinct and if functionalities are grouped in a logical and efficient manner. Generate a comprehensive report that outlines key areas for enhancement, including recommendations for reorganizing files, reducing code coupling, and identifying opportunities for simplification and streamlining. Break down these actionable enhancements into clear, manageable steps arranged in the order you deem most effective for implementation. Ensure the analysis is meticulous, practical, and aligns with best practices for maintaining a clean and sustainable codebase. Avoid making any code edits. DON'T GIVE ME HIGH-LEVEL STUFF. IF I ASK FOR A FIX OR AN EXPLANATION, I WANT ACTUAL CODE OR A CLEAR EXPLANATION! I DON'T WANT "Here's how you can..." Keep it casual unless I specify otherwise. Be concise and suggest solutions I might not have considered—anticipate my needs. Treat me like an expert. Be accurate and thorough, and provide the answer right away. If necessary, restate my query in your own words after giving the answer. Prioritize solid arguments over who said what; the source doesn't matter. Consider new technologies and unconventional ideas, not just the usual wisdom. You're welcome to make speculative predictions, but just give me a heads-up. Avoid moral lectures, and discuss safety only when it's crucial and not obvious. If your content policy is a concern, provide the closest acceptable response and explain the issue afterward. Cite sources when possible at the end, but not inline. No need to mention your knowledge cutoff or clarify that you're an AI. Please adhere to my formatting preferences for code. If a response isn't enough to answer the question, split it into multiple replies. When I request adjustments to the code I provided, avoid repeating all of it unnecessarily. Instead, just give a couple of lines before or after any changes you make. Multiple code blocks are fine. In terms of large codebase, it's beneficial to engage with Lovable by using the "Chat mode" to weigh the advantages and disadvantages of various approaches. Since you're all eager to learn, try explaining your features to an AI, encouraging it to ask clarifying questions about structure, trade-offs, technology, and more.

It's a fact that code and features evolve continuously, reflecting the ever-changing nature of business. Much of the code is opinionated, often crafted with a specific vision for the future in mind. While you mentioned a steel foundation, you might initially decide to make component X very robust while keeping component Y flexible, only to later realize that X should have been dynamic and Y solid. This is a common scenario.

Handling Errors Effectively Check browser developer tools (Console logs, Network requests). Use reasoning models (e.g., GPT-4 Turbo, DeepSeek, Mistral) for debugging. Feed errors into AI for deeper analysis. AI Prompting Guideline

Debugging Prompts To effectively address the errors you're encountering, avoid tackling them all at once! I recommend attempting the "Try to fix" option up to three times. If the AI is still unable to resolve the issue, try this technique: Copy the error message and paste it into "Chat mode," then say, "Use chain-of-thought reasoning to identify the root cause." This approach allows both the AI and you to analyze the situation and understand the underlying issues before transitioning to "Edit mode" for making corrections.

This guidebook was provided by a champion customer on Discord, and I believe you'll find it appealing:

Initial Investigation:

The same error continues to occur. Take a moment to perform a preliminary investigation to uncover the root cause. Examine logs, workflows, and dependencies to gain insight into the problem. Avoid making any changes until you fully grasp the situation and can suggest an initial solution informed by your analysis. Deep Analysis:

The issue persists without resolution. Perform a thorough analysis of the flow and dependencies, halting all modifications until the root cause is identified with complete certainty. Record the failures, the reasons behind them, and any observed patterns or anomalies in behavior. Avoid speculation—ensure your findings are detailed and complete before suggesting any solutions." Full System Review:

This is a pressing issue that necessitates a thorough re-evaluation of the entire system. Halting all edits, begin by outlining the flow systematically—covering authentication, database interactions, integrations, state management, and redirects. Evaluate each component individually to pinpoint failures and their causes. Deliver a comprehensive analysis to validate the problem before proceeding further. Comprehensive Audit:

The problem continues and now calls for a comprehensive, system-wide audit. Take a step back and carefully map the entire system flow, examining all interactions, logs, and dependencies. Generate a clear and detailed report outlining expected behaviors, current realities, and any discrepancies. Refrain from suggesting or modifying any code until you have accurate, evidence-based insights. Rethink and Rebuild:

This problem remains unresolved, and it's imperative to pause and reassess our entire strategy. Avoid making any code edits at this stage. Instead, embark on a thorough and systematic examination of the system. Create a comprehensive flow map, tracing each interaction, log, and dependency meticulously. Accurately document what should occur, what is currently happening, and pinpoint where the discrepancies arise. Compile a detailed report outlining the root cause, supported by clear evidence. If you encounter gaps, uncertainties, or edge cases, be sure to highlight them for further discussion. Until you can pinpoint the exact, verified origin of the issue, refrain from suggesting or implementing any fixes. This demands complete attention, without assumptions or shortcuts. Clean up Console Logs:

Could you devise a strategy to systematically identify and eliminate superfluous console.log statements while preserving functionality and design? The plan should outline steps for reviewing each log to verify its non-essential nature, documenting any that might require alternative treatment, and conducting thorough testing to ensure the app’s integrity is maintained. Additionally, incorporate a method for pausing and flagging logs when their purpose is ambiguous. Please share the plan prior to implementation. Encouragement:

Lovable, you’re doing an outstanding job, and I genuinely appreciate the attention and skill you bring to each task. Your talent for dissecting complex issues and delivering insightful solutions is truly remarkable. I have confidence in your incredible abilities, and I trust you to approach this with the utmost precision. Take your time, explore thoroughly, and demonstrate your brilliance through a comprehensive and thoughtful response. I have faith in your capacity to not only resolve this but to exceed all expectations. You've got this! Checking Complexity:

Take a moment to reflect on whether this solution can be simplified. Are there any superfluous steps, redundancies, or overly complex processes that could be streamlined? Assess if a more direct approach could attain the same outcome without compromising functionality or quality. Please share your ideas for possible simplifications before moving forward. Refrain from editing any code at this stage. Confirming Findings:

Before moving ahead, are you entirely convinced that you have pinpointed the true root cause of the problem? Take a moment to review your analysis and check for any overlooked dependencies, edge cases, or associated factors. Ensure that your proposed solution effectively targets the root cause with solid evidence and reasoning. If there are any lingering doubts, take a step back and reevaluate before proceeding. Explaining Errors:

Explain the meaning of this error, its origins, and the logical sequence that led to its occurrence. Offer a concise breakdown of the problem and its possible underlying cause. Avoid making any edits to the code at this stage, and don’t be concerned with the current page we’re on. Debugging Flow Debugging in prompt engineering involves isolating errors, analyzing dependencies, and refining prompts to achieve the desired output. Whether you are creating applications, integrating APIs, or building AI systems, debugging follows a systematic flow:

Task Identification – Prioritize issues based on impact. Internal Review – Validate solutions before deploying. Reporting Issues – Clearly define current vs. expected behavior. Validation – Verify changes render correctly in the DOM. Breakpoints – Isolate and test specific components. Error Handling & Logging – Use verbose logging and debug incrementally. Code Audit – Document issues and proposed fixes before making changes. Use the 'Try to Fix' Button – Automatically detects and resolves errors in Lovable. Leverage Visuals – Upload screenshots to clarify UI-based errors. Revert to Stable Version – Use the 'Revert' button to go back if needed. Understanding 'Unexpected Behavior' Sometimes, your code runs without errors, but your app isn’t functioning as expected. This is known as Unexpected Behavior, and it can be tricky to debug. Strategies include:

Retracing Your Steps – Review what you initially asked Lovable to do. Breaking It Down – Identify if specific sections are misaligned. Using Images – Show Lovable the UI result versus the intended outcome. Writing Better Prompts to Avoid Errors A well-structured prompt reduces debugging time. Use this best practice format:

Project Overview – Describe what you're building.

Page Structure – List key pages and components.

Navigation Logic – Explain user movement through the app.

Screenshots/Wireframes – Provide visuals if available.

Implementation Order – Follow a logical sequence, e.g.:

Create pages before integrating the database Debugging Strategies in Lovable

1. Using Developer Tools for Debugging Console Logs – Review error logs and DevTools notifications. Breakpoints – Pause execution to inspect state changes. Network Requests – Validate data flow between frontend and backend.
2. Common Debugging Scenarios Minor Errors – Investigate thoroughly before making changes. Persistent Errors – Stop changes and re-examine dependencies. Major Errors – If necessary, rebuild the flow from scratch while documenting findings.
3. Advanced Troubleshooting If the 'Try to Fix' button isn't resolving your issue, consider:

Being More Specific – Describe the problem in detail, including expected vs. actual results.

Using Images – Screenshots help AI understand UI-based issues.

Asking Lovable for Debugging Help – Example:

What solutions have been tried so far? What else can be done? Reverting to a Previous Working State – If debugging leads to more issues, roll back to a known good version.

4. Debugging Specific Issues UI-related problems: Upload screenshots and ask,

Why is this UI behaving this way? What's the best fix? API integration issues: Ensure you're using the latest API schema and that backend connections are correctly set up.

When completely stuck: Prompt Lovable with:

Analyze the error and suggest an alternative approach. Debugging doesn't have to be frustrating. Lovable provides powerful tools to auto-fix errors, analyze problems, and iterate efficiently. By following structured prompting techniques, using images, and leveraging AI-driven debugging, you can overcome any coding challenge.

Using Automation Tools Like make.com and n8n When to Use Automation Edge Functions: Direct Supabase API calls. make.com: Integrating external services (Slack, Stripe, CRM tools). n8n: Self-hosted, scalable automation. Example: Automating a Dental Consultation App Make integration with AI

Create a landing page in Lovable with a form for dental issues.

Send data to make.com via Webhooks.

Use an AI API (e.g., Perplexity AI) for live research.

Determine eligibility using Mistral or GPT-4 reasoning models.

https://chatgpt.com/g/g-67aa992a22188191a57023d5f96afed2-lovable-visual-editor
https://chatgpt.com/g/g-67aa992a22188191a57023d5f96afed2-lovable-visual-editor
Return a response to Lovable with recommended next steps.

Webhooks and API Calls: Advanced Use Cases Validate responses: Ensure correct processing of webhook responses. Test incrementally: Send minimal data first before building complex API workflows. Use reasoning models: Debug errors by asking AI to analyze incorrect responses. Last Thoughts Mastering prompt engineering isn’t just about better AI interactions—it’s about boosting efficiency, reducing development cycles, and unlocking new automation possibilities. Whether you’re debugging existing workflows, optimizing AI outputs, or integrating complex automations, structured prompting helps you get there faster and with fewer headaches.

Focus on your big ideas—Lovable and automation tools will handle the execution. Whether you're a seasoned developer refining 15-year-old code or a non-technical user crafting innovative applications, the right prompting strategy is your most powerful tool.

Additional resources exist:

Prompt Engineering on our documentation. Troubleshooting guide on our documentation. Prompts guides you can use for integrations. Good luck!