Comprehensive debugging manual organized by error types for Lovable

Before you begin

1. Download the file as markdown (name the file as Comprehensive_debugging_manual.md)
2. Upload to your project GitHub repository
3. Prompt Lovable

   "Read this document - Comprehensive_debugging_manual.md. Tell me if it helps you and create a prompt which I should ask you to help with troubleshooting this issue please:

   {paste your issue}"
4. When it gives you the prompt or a suggested fix, paste it in the chat but add this to the very end of the message

   "Before you proceed, answer me in great detail - Why do you think this will work? Wait for my approval."

AND GOOD LUCK!

Bonus:

When looking to debug a persistent issue, there are a few approaches you can take aside from the one above:

1. **Refactor the file -** this will split the original file into smaller chunks and that is usually making things easier for Lovable to "chew" on your requests, including bug fixes
2. **Load the file into Claude 3.7 or ChatGPT -** Go to Github, download the file(s) in question, load them into Claude or ChatGPT and tell them what the issue is and attach the file and ask them to write the full code, line by line, so that you can copy/paste it back into the original file
3. **Add more console logs -** Simply ask Lovable to add more console logs after each critical step of the function you are trying to build and that will help it diagnose the error more granulary.
4. **Ask Lovable to help you create a better prompt -** I will best illustrate this using an example from one of the apps I was building:

“Ok, think this through please, deeply analyze.

1. You scrape the recipe
2. You find 9 ingredients
3. You parse just 1 of them, even though all data is readily available in the edge function logs

Audit the code responsible for displaying the edge function "result" into the UI, and see if it has limited capabilities of writing just 1 ingredient vs all 9 or as many as the edge function returns?

DONT CODE, LET'S DIAGNOSE THIS.

GIVE ME A BETTER PROMPT SO THAT I CAN ASK YOU HOW TO INVESTIGATE THE ISSUE HERE”

The last sentence is critical as Lovable will return an explanation and a prompt you can copy/paste in the chat to help it diagnose the problem more deeply.

Keep Shipping!

Type Mismatches & TypeScript Errors

Prompt template:
Please analyze this TypeScript error:
1. Show the relevant type definitions from both DB schema and TypeScript
2. Trace the data transformation pipeline
3. Identify where types diverge
4. Suggest fixes that maintain type safety

Understanding Type Errors

When you encounter a TypeScript error, it's crucial to understand what the compiler is telling you. Here's how to interpret common errors:

1. Property Missing Errors
When TypeScript says "Property X is missing", it means your object shape doesn't match what's expected. Think of it like a puzzle piece missing a tab that should fit into another piece.

2. Possible Undefined Errors
These occur when TypeScript thinks a value might not exist. It's like reaching into a cookie jar - you need to check if there's actually a cookie there before trying to eat it.

3. Type Assignment Errors
When one type doesn't match another, it's like trying to put a square peg in a round hole. TypeScript is telling you the shapes don't match.

Debugging Strategy

Step 1: Identify the Source

- Look at where the data originates (database, API, user input)
- Track how it flows through your application
- Find where the type mismatch first occurs

Step 2: Compare Types

- Look at your database schema
- Check your TypeScript interfaces
- Compare them side by side
- Note any mismatches in:
- Property names
- Data types
- Optional vs required fields

Step 3: Check Data Transformations

- Look for places where data is transformed
- Verify that transformations maintain type safety

- Check for implicit type conversions

Step 4: Common Patterns
Think about these common scenarios:

- Database numbers coming in as strings
- Dates needing parsing
- Nullable fields not being handled
- Optional properties being treated as required
- Real-World Examples
- Database Types vs TypeScript
- Instead of showing code, think about it this way:
- Database numbers might be "numeric" but TypeScript expects "number"
- Database timestamps come as strings but you might want Date objects
- Database UUIDs are strings in TypeScript
- JSON fields need specific TypeScript shapes

4. Best Practices for Debugging Database Issues:

- **Always check database configuration (triggers, RLS, functions) first**
- **Review logs from both client and server**
- **Look for any automatic processes that might override manual changes**

For future reference, when dealing with database-related issues, starting with "Please check all database triggers, functions, and RLS policies that might affect [specific operation]" will help me identify such issues much faster.

Handling Optional Data

Think of it like a form:

Some fields are required (must be filled)
Some are optional (can be left blank)
Your types should reflect this reality

Best Practices

Type Safety First

- Always define expected shapes
- Don't use 'any' as an escape hatch
- Think about null and undefined

Consistent Naming

- Use the same property names throughout
- Be consistent with casing
- Match database column names when possible

Documentation

- Comment complex type relationships
- Explain why certain types are chosen
- Document any special handling

Testing

- Test with real data shapes
- Verify edge cases
- Check null/undefined handling

Data Flow Issues (UI not reflecting DB)

Prompt template:
I have a mismatch between database and UI values. Please:
1. Show the data flow: DB → API → State → UI
2. Check query/mutation invalidation patterns
3. Verify data transformations at each step
4. Add console.logs at critical points
5. Review RLS policies that might affect data access

Understanding Data Flow Problems

The Data Pipeline

Think of data flow like a river system:

- The Database is the source (lake)
- The API calls are the rivers
- The State management is like reservoirs
- The UI components are like irrigation systems

When your UI isn't showing the right data, there's a blockage somewhere in this system.

Common Data Flow Issues

1. Stale Data
    - Like looking at yesterday's weather forecast
    - Data in UI doesn't update when database changes
    - Cache might be holding onto old information
    - Query invalidation might not be triggering
2. Data Transformation Problems
    - Like a translation getting garbled
    - Data changes shape as it moves through the app
    - Numbers become strings
    - Dates lose their formatting
    - Arrays get flattened unexpectedly
3. State Management Issues
    - Like a broken gauge showing wrong readings
    - State updates don't trigger re-renders
    - Multiple sources of truth conflict

- Race conditions in async updates

How to Debug Data Flow

1. Follow the Data Journey
Start at the database and check each step:
    - Is the query correct?
    - Is the API returning expected data?
    - Is state management capturing updates?
    - Is the UI receiving the right props?
2. Check the Timing
    - When should data update?
    - What triggers the updates?
    - Are updates happening in the right order?
    - Is there a delay that matters?
3. Verify Data Shapes
    - What does the data look like leaving the database?
    - How does it change through transformations?
    - What shape does the UI expect?
    - Where might transformations fail?

Real-World Examples

Admin Dashboard Scenarios

- User list not refreshing after updates
- Statistics showing wrong numbers
- Filters not affecting displayed data
- Sorting not persisting
- Pagination losing state

Common Solutions

Think about:

- Query invalidation timing
- Cache management
- State synchronization
- Error boundary handling
- Loading state management

Best Practices

1. Monitoring
    - Add strategic console logs
    - Track state changes
    - Monitor network requests
    - Watch for error patterns
2. Testing
    - Test data transformations
    - Verify state updates
    - Check edge cases
    - Validate error handling
3. Error Handling
    - Graceful degradation
    - User feedback
    - Recovery strategies
    - Error boundaries

Supabase-Specific Issues

Prompt template:
I'm encountering a Supabase error. Please:
1. Show the relevant table schemas and RLS policies
2. Check foreign key relationships
3. Verify data types between Supabase and TypeScript
4. Review query syntax for Supabase client
5. Check edge function logs if relevant

Understanding Supabase Architecture

1. Database Layer
Think of Supabase like a house:
- Tables are rooms
- RLS policies are security guards
- Foreign keys are doorways between rooms
- Triggers are automatic systems

2. Common Supabase Issues
- Authentication Problems
  - Like having the wrong key for a door
- Token expiration
- Missing RLS policies
- Wrong role permissions
- Data Access Issues
- RLS blocking legitimate requests
- Foreign key constraints failing
- Unique constraint violations
- Type mismatches between Postgres and JavaScript

3. Debugging Steps
- Check Access Rights
- Review RLS policies
- Verify user roles
- Check token validity
- Confirm table permissions
- Data Integrity
- Examine table relationships
- Verify constraint rules
- Check data types
- Review default values

Real-World Scenarios

1. Data Not Showing Up
Common causes:
- RLS blocking access
- Wrong table relationships
- Missing JOIN conditions
- Incorrect foreign keys

2. Insert/Update Failures
Typical issues:
- Unique constraint violations
- Foreign key mismatches
- Type conversion errors
- Missing required fields

3. Authentication Problems
Common scenarios:
- Token expiration
- Wrong role assignment
- Missing permissions
- Incorrect policy definitions

Best Practices

1. RLS Policy Design
- Start restrictive
- Add permissions gradually
- Test each policy
- Document policy logic

2. Data Modeling
- Plan relationships carefully
- Use appropriate data types
- Consider constraints early
- Document schema decisions

3. Error Handling
- Catch specific Supabase errors
- Provide meaningful feedback
- Log important details
- Plan recovery strategies

State Management Problems

Prompt template:
My component state isn't working as expected. Please:
1. Show the state management flow
2. Check React Query invalidation patterns
3. Verify effect dependencies
4. Review component re-render triggers
5. Check for race conditions in async operations

Understanding React State Flow

1. State Architecture Fundamentals

Think of state like a building's electrical system:

- Global state is the main power supply
- Local state is like individual room circuits
- Props are like power outlets
- Effects are like automatic switches

2. Common State Issues

State Updates Not Reflecting

- Like flipping a switch but light doesn't change
- Component not re-rendering
- Stale closures
- Batched updates confusion

React Query Specific Problems

- Cache invalidation timing
- Stale data handling
- Background updates
- Optimistic updates failing

3. State Synchronization

Problems like:

- Multiple sources of truth
- Race conditions
- Waterfalls
- Unnecessary re-renders

Debugging Approach

1. Component State

Check:

- Initial state values
- Update triggers
- Re-render conditions
- Effect dependencies

2. Query Management

Verify:

- Cache configuration
- Invalidation rules
- Refetch policies
- Error boundaries

3. Performance Impact

Monitor:

- Re-render frequency
- Memory usage
- Network calls
- Bundle size

Real-World Examples

1. Form State Issues

Common problems:

- Input lag
- Lost focus
- Validation timing
- Submit race conditions

2. List/Grid Updates

Typical issues:

- Item updates not showing
- Sort/filter state loss
- Pagination state reset
- Selection state inconsistency

3. Real-time Updates

Challenges with:

- Websocket state sync
- Optimistic updates
- Conflict resolution
- Error recovery

Best Practices

1. State Organization

- Keep state close to usage
- Split complex state
- Document state purpose
- Plan state shape

2. Performance Optimization

- Use appropriate hooks
- Implement memoization
- Batch updates
- Lazy load state

3. Testing Strategy

- Test state transitions
- Verify update flows
- Check edge cases
- Monitor performance

Performance Issues

Prompt template:
My application is experiencing performance issues. Please:
1. Review query caching strategy
2. Check unnecessary re-renders
3. Analyze data fetching patterns
4. Review component memoization
5. Check for N+1 query problems

Understanding Performance Bottlenecks

1. Core Performance Concepts

Think of performance like a car's efficiency:

- CPU usage is like engine power
- Memory usage is like fuel consumption
- Network calls are like pit stops
- Rendering is like acceleration

2. Common Performance Problems

Rendering Issues

- Too many re-renders (engine revving needlessly)
- Heavy components (carrying too much weight)
- Unoptimized lists (dragging extra cargo)
- Memory leaks (fuel leaks)

Data Loading Problems

- Waterfall requests (sequential pit stops)
- Over-fetching (filling up too much)
- Under-fetching (too many small fills)
- Cache mismanagement (wasting fuel)

3. Network Performance

Issues like:

- Too many API calls
- Large payload sizes
- Slow response times
- Connection overhead

Debugging Approach

1. Identify Bottlenecks

Use tools like:

- React DevTools
- Performance tab
- Network panel
- Memory profiler

2. Measure Impact

Check:

- Load times
- Time to interactive
- First contentful paint
- Largest contentful paint

3. Common Patterns

Look for:

- Unnecessary re-renders
- Expensive calculations
- Unoptimized images
- Heavy third-party scripts

Real-World Examples

1. List Performance

Problems with:

- Large datasets
- Complex item rendering
- Scroll performance
- Dynamic updates

2. Form Performance

Issues like:

- Input lag
- Validation delays
- Submit handling
- State updates

3. Image Loading

Challenges with:

- Large images
- Many images
- Lazy loading
- Progressive loading

Best Practices

1. Code Optimization

- Use proper hooks
- Implement virtualization
- Optimize bundle size
- Code splitting

2. Data Management

- Implement caching
- Batch requests
- Use pagination
- Optimize queries

3. Asset Optimization

- Compress images
- Lazy load resources
- Use CDNs
- Implement caching

Authentication/Authorization Issues

Prompt template:
I'm having auth-related problems. Please:
1. Show the auth flow
2. Check RLS policies
3. Verify token handling
4. Review auth state management
5. Check role-based access control

Understanding Auth Flow

1. Auth Architecture

Think of auth like a secure building:

- Authentication is the ID check at entrance
- Authorization is access cards for different areas
- Tokens are like temporary visitor badges
- Roles are like security clearance levels

2. Common Auth Problems

Authentication Issues

- Failed login attempts
- Token expiration
- Session management
- Social auth integration

Authorization Problems

- Missing permissions
- Role confusion
- Access control gaps
- Policy enforcement

3. Supabase Auth Specifics

Think about:

- JWT token handling
- Role-based access
- Row-level security
- Policy enforcement

Debugging Approach

1. Auth Flow Check

Verify:

- Login process
- Token storage
- Session persistence
- Logout cleanup

2. Permission Verification

Check:

- User roles
- Access policies
- Table permissions
- Function access

3. Common Patterns

Look for:

- Token expiration
- Missing headers
- Invalid credentials
- Policy conflicts

Real-World Examples

1. Login Issues

Problems with:

- Credential validation
- Social auth flow
- Password reset
- Email verification

2. Access Control

Issues like:

- Missing data access
- Unauthorized actions
- Role confusion
- Policy conflicts

3. Session Management

Challenges with:

- Token refresh
- Session timeout
- Multiple devices
- Concurrent access

Best Practices

1. Security First

- Implement proper validation
- Use secure token storage
- Handle errors gracefully
- Log security events

2. User Experience

- Clear error messages
- Smooth auth flow
- Proper redirects
- Loading states

3. Maintenance

- Regular token cleanup
- Policy reviews
- Access audits
- Security updates

Edge Function Problems

Prompt template:
My edge function isn't working correctly. Please:
1. Show the edge function logs
2. Verify CORS configuration
3. Check secret/environment variable access
4. Review error handling
5. Verify function deployment status

Understanding Edge Functions

1. Edge Computing Concept

Think of edge functions like local branches of a business:

- Closer to the customer (user)
- Faster response times
- Local processing
- Distributed computing

2. Common Edge Function Issues

Deployment Problems

- Function not updating
- Environment variables missing
- Dependencies not included
- Runtime errors

Integration Issues

- CORS configuration
- API gateway problems
- Authentication headers
- Rate limiting

3. Debugging Edge Functions

Logs and Monitoring

- Check deployment logs
- Monitor execution times
- Track error rates
- Review request patterns

Common Failure Points

- Secret access
- Memory limits
- Timeout issues
- Cold starts

Real-World Scenarios

1. Data Processing

Problems with:

- Large payloads
- Timeout limits
- Memory constraints
- Processing errors

2. External API Integration

Issues like:

- API rate limits
- Authentication
- Response handling
- Error recovery

3. Real-time Processing

Challenges with:

- WebSocket connections
- Stream processing
- Event handling
- State management

Best Practices

1. Development

- Local testing
- Error handling
- Logging strategy
- Performance monitoring

2. Deployment

- Environment management

- Secret handling
- Version control
- Rollback plans

3. Maintenance

- Regular updates
- Performance monitoring
- Error tracking
- Usage analytics

Database Constraints

Prompt template:
I'm getting database constraint errors. Please:
1. Show the relevant table schemas
2. Check foreign key relationships
3. Verify unique constraints
4. Review default values
5. Check null constraints

Understanding Database Constraints

Think of database constraints like rules in a board game:

- Primary keys are like unique player IDs
- Foreign keys are like card relationships
- Unique constraints are like one-of-a-kind items
- Not null constraints are like required moves

1. Types of Constraints

Primary Key Constraints

- Like a social security number
- Must be unique
- Cannot be null
- Identifies each record

Foreign Key Constraints

- Like family relationships
- References other tables
- Maintains data integrity
- Prevents orphaned records

Unique Constraints

- Like email addresses
- No duplicates allowed
- Can be single or multiple columns
- Optional nullability

Check Constraints

- Like game rules
- Validates data values

- Enforces business rules
- Maintains data quality

Common Constraint Issues

1. Violation Scenarios

- Duplicate unique values
- Missing required data
- Invalid foreign keys
- Check constraint failures

2. Impact Areas

- Data insertion
- Updates
- Deletions
- Bulk operations

Debugging Approach

1. Identify Constraint

- Check error message
- Review constraint definition
- Understand business rule
- Verify data requirements

2. Data Validation

- Check input data
- Verify existing data
- Test edge cases
- Validate transformations

3. Resolution Strategies

- Data cleanup
- Constraint modification
- Error handling
- Recovery procedures

Best Practices

1. Design Phase

- Plan constraints early
- Document requirements
- Consider scalability
- Test thoroughly

2. Implementation

- Clear naming conventions
- Proper error messages
- Efficient indexing
- Performance consideration

3. Maintenance

- Regular validation
- Performance monitoring
- Documentation updates
- Regular reviews

Component Structure Issues

Prompt template:
My component structure isn't working. Please:
1. Show the component hierarchy
2. Review prop drilling patterns
3. Check context usage
4. Verify event handling
5. Review lifecycle methods

Understanding Component Architecture

Think of components like building blocks:

- Parent components are like foundation blocks
- Child components are like specialized pieces
- Props are like connecting pins
- State is like internal wiring

1. Component Hierarchy

Think of it as a family tree:

- Root components (grandparents)
- Container components (parents)
- Presentational components (children)
- Shared components (cousins)

2. Common Component Issues

Prop Drilling

- Like passing a message through many people
- Props going through multiple levels
- Unnecessary component coupling
- Maintenance complexity

Component Communication

Problems like:

- Parent-child communication
- Sibling communication
- Distant component updates
- Event bubbling

3. State Management

Issues with:

- Local vs global state
- State lifting
- Shared state
- Derived state

Debugging Approach

1. Component Tree Analysis

Check:

- Component relationships
- Prop flow
- State location
- Event handlers

2. Performance Impact

Look for:

- Unnecessary renders
- Prop changes
- State updates
- Effect triggers

3. Common Patterns

Identify:

- Repeated logic
- Shared functionality
- Component coupling
- State dependencies

Best Practices

1. Component Design

- Single responsibility
- Clear interfaces
- Consistent patterns
- Proper documentation

2. State Management

- Minimize prop drilling
- Use appropriate context
- Plan state location
- Consider composition

3. Performance

- Implement memoization
- Lazy loading
- Code splitting
- Bundle optimization

API Integration Problems

Prompt template:
My API integration isn't working. Please:
1. Show the API call flow
2. Check error handling
3. Verify request/response types
4. Review authentication headers
5. Check CORS configuration

+Understanding API Architecture

Think of APIs like a restaurant service system:

- Endpoints are like menu items
- Parameters are like customization options
- Headers are like order specifications
- Responses are like the delivered meals

1. Common API Issues

Request Problems

- Authentication failures (wrong credentials)
- Invalid parameters
- Missing headers
- Rate limiting
- CORS issues

Response Handling

- Parsing errors
- Unexpected data formats
- Missing error handling
- Timeout issues
- Status code confusion

2. Supabase-Specific API Issues

Authentication

- JWT token issues
- Session management
- Role permissions

- Policy conflicts

Data Operations

- Query structure
- Filter conditions
- Join relationships
- Real-time subscriptions

Debugging Approach

1. Request Inspection

Check:

- URL construction
- Query parameters
- Header values
- Body format
- Authentication tokens

2. Response Analysis

Verify:

- Status codes
- Response format
- Error messages
- Data structure
- Performance metrics

3. Common Patterns

Look for:

- Network timeouts
- Cache issues
- Race conditions
- Error handling gaps

Best Practices

1. Error Handling

- Catch specific errors
- Provide feedback
- Implement retries
- Log failures

2. Performance

- Implement caching
- Optimize queries
- Batch requests
- Monitor timing

3. Security

- Secure credentials
- Validate input
- Sanitize output
- Use HTTPS

4. Testing

- Unit tests

- Integration tests
- Error scenarios
- Edge cases

Understanding Error Management

Think of error handling like a safety net system:

- Try/catch blocks are like safety nets
- Error boundaries are like containment zones
- Error messages are like warning signs
- Recovery strategies are like emergency procedures

1. Types of Errors

Runtime Errors

- JavaScript exceptions
- Network failures
- State mutations
- Async operation failures

UI Errors

- Rendering failures
- Component crashes
- Props validation
- State inconsistencies

Data Errors

- API failures
- Database errors
- Validation errors
- Type mismatches

2. Error Handling Strategies

Global Error Handling

- Error boundaries
- Global error listeners
- Logging systems
- Recovery mechanisms

Local Error Handling

- Try/catch blocks
- Error states
- Fallback UI
- Recovery options

3. User Experience

Consider:

- Clear error messages
- Recovery options
- Graceful degradation
- User feedback

Best Practices

1. Error Prevention

- Input validation
- Type checking
- Proper initialization
- Guard clauses

2. Error Recovery

- Retry mechanisms
- Fallback options
- State reset
- Cache clearing

3. Error Reporting

- Error logging
- User feedback
- Developer alerts
- Analytics tracking

4. Maintenance

- Error monitoring
- Pattern analysis
- Regular reviews
- Documentation updates

General Debugging Guide

General Debugging Checklist:
Identify the error type from above categories
Use the relevant prompt template
Follow the step-by-step analysis
Check console logs at each step
Verify types match throughout the pipeline
Review relevant documentation
Test proposed solutions in isolation
Verify fix doesn't break other functionality
Best Practices for Error Reports:
Include full error message
Show relevant code snippets
Include console logs
Describe expected vs actual behavior
List recent changes that might have caused the issue
Specify the environment (dev/prod)

1. Systematic Approach

The Scientific Method for Debugging

Think of debugging like being a detective:

- Observe the problem (collect evidence)
- Form a hypothesis (what might be wrong)
- Test the hypothesis (investigate)
- Analyze results (evaluate findings)
- Draw conclusions (solve the case)

Step-by-Step Process

1. Reproduce the Issue
    - Identify exact steps to recreate
    - Note environment conditions
    - Document browser/device specifics
    - Record user actions
2. Isolate the Problem
    - Narrow down affected components
    - Identify trigger conditions
    - Note related state changes
    - Check network activity
3. Gather Information
    - Console logs

        - Network requests
        - State changes
        - Component renders
        - Error messages

2. Tools and Techniques

Browser Developer Tools

- Console tab for logs
- Network tab for requests
- Elements tab for DOM
- Sources for breakpoints
- Performance for timing

React Developer Tools

- Components tab
    - Props inspection
    - State viewing
    - Hook examination
    - Render counts
- Profiler tab
    - Render timing
    - Component updates
    - Performance metrics

Code-Level Tools

- Debugger statements
- Console logging
- Error boundaries
- Performance monitoring

3. Common Patterns

Symptom Recognition

- UI not updating
- Slow performance
- Memory leaks
- Network failures
- State inconsistencies

Quick Checks

1. Console Errors
    - Syntax errors
    - Runtime errors
    - Promise rejections
    - Network failures
2. React Warnings
    - Key warnings
    - Hook violations
    - Deprecated features
    - Memory leaks
3. Network Activity
    - Failed requests
    - Slow responses
    - Wrong data
    - Authentication issues

4. Advanced Debugging

Performance Debugging

- React Profiler
- Chrome Performance tab
- Memory usage
- Network waterfall

State Debugging

- Redux DevTools
- React Query DevTools
- Local Storage
- Session Storage

Network Debugging

- Network tab
- Request/Response inspection
- Headers examination
- Payload analysis

5. Best Practices

Code Organization

- Clear component structure
- Consistent naming
- Proper error handling
- Clean code principles

Documentation

- Error logs
- Bug reports
- Solution documentation
- Pattern recognition

Prevention

- Unit tests
- Integration tests
- Error boundaries
- Type checking