Comprehensive Guide to Vibe Coding using Claude Code

A field-tested, practical guide by Michał Gołębiowski

Introduction

This isn't another generic AI tutorial. It's a field-tested playbook for builders-for anyone with an idea who wants to go from spark to a shipped application, fast.

Vibe Coding changes the rules: you code with an AI - not as a tool, but as a coding partner. But to make it work, you need to know how to guide the conversation strategically and avoid the pitfalls that trap 90% of practitioners.

This guide walks through each stage of building an AI-written application using Claude Code by Anthropic - my preferred tool after extensive testing against Cursor, Windsurf, and others. It's built for both beginners who want clarity and experienced developers who want to 10x their velocity.

This guide provides practical insights on:

- What you need to do - with specific actions
- Common traps to avoid - learned through painful experience
- How to use Claude Code effectively - beyond basic prompting
- Real examples and templates - copy-paste ready

I've condensed lessons from hundreds of hours using Claude Code, building real applications, and refining prompt engineering as both a UX manager and AI practitioner.

Note: If you think Vibe Coding is a matter of simple prompts, you might end with code that is as large as an operating system, writes only "Hello World", and can't be debugged without a Quantum Computer.

What is Vibe Coding and Who is it For?

Vibe Coding is the practice of building software through conversational collaboration with a generative AI, treating it like a highly capable junior developer (at least for the moment of writing this guide) who needs clear goals, structured context, and iterative feedback. You are the business, AI is the executor.

Vibe Coding IS:	**Vibe Coding is NOT:**
Translating intent into executable instructions	Copying code snippets from ChatGPT or any other LLM
Using AI to test hypotheses and iterate rapidly	Asking AI to "build me an app" - it will do so based on a generic approach
Building fast while maintaining enough structure to avoid technical debt	Replacing your technical judgment - you are in charge, if you are a skilled, experienced developer that is your superpower, use it
Amplifying your capabilities, not replacing your decision-making	Magic that works without understanding

Who This Guide is For

- Designers who want to prototype interactive experiences without full-stack expertise
- Product managers who need working MVPs for validation
- Developers who want to 10x their prototyping velocity and focus on architecture over boilerplate
- Founders and tinkerers who are short on time but full of ideas
- Technical professionals transitioning to AI-augmented workflows

Core requirement: If you can describe what you want clearly and follow structured processes, you can build it with Claude Code. But... you need to be skilled in communication and straightforward instruction (remember: garbage-in, garbage-out).

Why Vibe Coding Matters?

Traditional development looks like this: **Discover - Ideate (and usability testing) - Weeks of coding - Testing MVP**.

Vibe Coding looks like this: **Discover - Ideate - Hours of structured prompting - Working MVP for testing purpose and iterations**

This starkly highlights the speed advantage, freeing up more time for iteration.

Real Examples I've Witnessed and Time Savings:

- Data dashboard built during a Zoom call - 5 prompts, live preview, exported to production
- API wrapper + frontend generated with zero boilerplate setup
- Internal tool rebuilt in one evening with better UX than the original
- PDF processing pipeline implemented in 2 hours vs. estimated 2 weeks
- App to help gather insightful stock exchange decisions making in 30 minutes
- Real-time social media aggregator in ~30 prompts
- Framework for a large scale news website with Agentic Architecture in 12 days.

So if someone says it is impossible, he/she probably didn't have a chance to make a full use of Agentic Coders like Claude Code.

Why This Speed Matters

I'm sure you are aware of this but let's say it out loud:

- For busy professionals, this eliminates context switching. You stay in a flow state from concept to working software because it is not a few-month project. You have quick gratification and motivation to go further.

- For experimentation, you can validate ideas before committing significant resources. You are the unicorn. You can quickly validate your hypothesis before going to stakeholders.
- For learning, you see working code immediately, understanding patterns faster than traditional tutorials. You can check what works, and what's not in the real examples with the full context of the app.

So Vibe Coding allows you to:

- Skip boilerplate and focus on business logic
- Describe behavior in natural language
- Get real-time feedback and iterate instantly
- Prototype ideas that would normally require a dev team

Getting Started with Claude Code

Before diving into the Vibe Coding stages, you'll need to set up your environment with Claude Code. Below information is my enchanced installation instructions based on Getting started with Claude Code

System Requirements

Ensure your system meets the following prerequisites:

- **Operating Systems**: macOS 10.15+, Ubuntu 20.04+/Debian 10+, or Windows via WSL (Windows Subsystem for Linux).
- **Hardware**: Minimum 4GB RAM.
- **Software**: Node.js 18+ (Node Package Manager `npm` is included with Node.js), `git` 2.23+ (optional, but highly recommended for version control), GitHub or GitLab CLI for PR workflows (optional).
- **Network**: Active internet connection for authentication and AI processing.
- **Location**: Claude Code is only available in supported countries.

Installation

1. **Install Node.js (if not already installed or if you have an older version)**: Ensure you have Node.js version 18 or higher. If you need to install or update, you can use your distribution's package manager or Node Version Manager (nvm). For example, on Ubuntu:

   ```
   sudo apt update && sudo apt upgrade -y
   curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
   sudo apt install -y nodejs
   ```

   Verify installation: `node --version` and `npm --version`

2. **Install Claude Code**: Open your terminal and run the following command. **Important**: Do NOT use `sudo` with this command unless absolutely necessary and you understand the potential permission issues.

   ```
   npm install -g @anthropic-ai/claude-code
   ```

   If you encounter permission errors, consider configuring npm to install global packages in your user directory:

   ```
   mkdir -p ~/.npm-global
   npm config set prefix ~/.npm-global
   echo 'export PATH=~/.npm-global/bin:$PATH' >> ~/.bashrc
   source ~/.bashrc
   ```

Authentication and Using the Claude API

1. **Start Claude Code**: Navigate to your project directory and simply run:

   ```
   cd your-project-directory
   claude
   ```

2. **Complete Authentication**: On the first launch, Claude Code will guide you through an OAuth process to connect to your Anthropic account.

    - **Anthropic Console**: This is the default option. A browser window will open, prompting you to log in to your Anthropic Console (console.anthropic.com). You will need an active billing setup (e.g., sufficient credits) in your Anthropic account to use Claude Code. Follow the prompts to complete the connection.
    - **Claude App (Pro or Max plan)**: If you have a Pro or Max plan, you can log in with your Claude.ai account for a unified subscription.
    - **Enterprise platforms**: Claude Code can also be configured to use Amazon Bedrock or Google Vertex AI for enterprise deployments.

3. **Initial Project Setup (Recommended)**: Once authenticated, you can use Claude Code to initialize your project:

   ```
   claude > /init
   ```

   This command will generate a `CLAUDE.md` file in your project root, which serves as a project guide and context for Claude Code. It's highly recommended to commit this file to your version control.

Define the Purpose

Before writing a single prompt, you need absolute clarity on one thing: **Why are you building this?** And this is crucial. Without knowing "why" don't go any further. The results will demotivate you.

Claude Code mirrors your intention. Vague input = vague output (trash in - trash out). Precise and purposeful input = clean, focused results.

The Purpose Framework

Ask yourself these three questions and write down the answers:

1. **What problem is this solving?** Be specific. "Make invoicing easier" is vague. "Allow small business owners to generate and send invoices in under 2 minutes without accounting software" is clear.
2. **What change do I want to see?** Focus on the outcome, not the features. "Users can complete task X 10x faster" vs. "Users have a dashboard".

3. **Is it worth solving with software?** Could this be solved with a spreadsheet, existing tool, or manual process? If yes, maybe software isn't needed.

Note: You need to stop thinking about solutions, think about the purpose and results. It is very important at this stage.

Vision File Template

Create a file called `0_vision.md` in your project root:

```
# Project Vision

## Problem
[Specific problem you're solving]

## Target User
[Who exactly will use this]

## Success Metric
[How you'll know it's working]

## Why Software?
[Why this needs to be built vs. using existing tools]
```

Pro Tip: Challenge Your Assumptions After writing your initial vision, paste it into Claude (any interface) and ask:

> Challenge this project idea with 5 hard questions. Help me identify potential flaws or missing pieces.

This forces you to complete gaps before coding starts, saving hours of rework later. This foundation isn't optional. Whether you're designing a checkout flow or prompting Claude Code, unclear purpose leads to unfocused results. This vision file becomes your north star when prompts start drifting.

Know Your User

Before prompting Claude to build anything, you need to know who it's for - not as a vague persona, but with enough specificity that Claude can make appropriate UX decisions. This is crucial to have in mind: you are not prompting until you finish describing your app.

The User Clarity Framework

Answer these questions with specificity:

1. **Who will use this?** If it's you: describe your needs, habits, expectations, and frustrations clearly. If it's others: be specific about their technical comfort, time constraints, and goals.
2. **What do they want to achieve?** Focus on the job-to-be-done, not features they might want.
3. **What frustrates them in current tools?** This guides what to avoid and what to optimize for.

User Profile Template

```
## User Profile

**Primary User**: [Specific description]
**Goal**: [What they're trying to accomplish]
**Context**: [When/where they'll use this]
**Frustrations**: [What slows them down or annoys them]
**Technical Comfort**: [Low/Medium/High - affects UI complexity]
**Time Constraints**: [How much time they have for this task]
```

Example: Invoice Tool for Small Business

- **Bad user description**: "Business owners who need invoicing"

- **Good user description**: "Solo consultants and freelancers who bill 5-20 clients monthly. They're not accounting experts, work from laptops, and get frustrated with bloated software that requires tutorials. They need to create and send invoices in under 2 minutes between client calls."

Feeding This to Claude

Once you have clarity, give this context to Claude directly:

> **Context**: This tool is for solo consultants who need to create invoices quickly between client calls. They prioritize speed over features, get overwhelmed by complex interfaces, and work primarily on laptops. **Task**: Help me design the user flow and interface priorities for an invoice generator, keeping this user's needs and constraints in mind.

Why this matters: Claude will make thousands of micro-decisions about UX, naming, flow, and complexity. If you don't anchor it to a specific user, it defaults to "generic tech user" - which may be wrong for your audience.

Define Outcome and Success Metrics

If you can't define success, neither can Claude. Before any code gets written, establish clear criteria for what "working" means and how you'll measure it.

The Success Definition Framework

1. **Define "Working"** Choose your target:
    - **Proof of concept**: Demonstrates core functionality, may have rough edges.
    - **MVP**: Minimally viable product that real users can test.
    - **Polished demo**: Ready for presentation or investor demo.
    - **Production ready**: Can handle real users and edge cases.
2. **Set Specific Success Criteria** Instead of vague goals, define measurable outcomes:

Bad success criteria	Good success criteria
"Users like it"	"90% of test users can complete core task in under 2 minutes"
"It works well"	"Handles 1000 concurrent users without performance issues"
"People find it useful"	"Zero crashes during demo with realistic data volume"
	"Users complete onboarding without asking questions"

Success Metrics Template

```
## Success Definition

**Target Level**: [Proof of concept/MVP/Demo/Production]

**Core Success Criteria**:
1. [Specific, measurable outcome]
2. [Performance or usability threshold]
3. [Technical requirement]

**Validation Method**:
- [ ] Will you demo it to someone?
- [ ] Will you run usability tests?
- [ ] Will you measure usage/conversion?
- [ ] Will you deploy it for real users?

**Timeline**: [When do you need this working]
```

Example: PDF Data Extraction Tool

```
## Success Definition
**Target Level**: MVP

**Core Success Criteria**:
1. Processes 90% of uploaded PDFs without errors
2. Extracts tabular data to usable Excel format in <60 seconds
```

```
3. Works on mobile and desktop browsers
4. Users can complete full workflow without instructions

**Validation Method**:
- [ ] Test with 20 different PDF types
- [ ] Time 5 users completing the full workflow
- [x] Deploy publicly and monitor usage analytics

**Timeline**: Working version in 2 weeks
```

Feeding This to Claude

Include your success criteria in Claude conversations:

> **Goal**: Build a PDF data extraction tool that processes 90% of uploaded PDFs without errors and outputs Excel files in under 60 seconds. Please keep these success criteria in mind when suggesting features or implementation approaches.

Why this works: Claude can optimize for your specific goals rather than building generic features. It will suggest implementations that align with your success metrics.

Benchmarking and Inspiration

Before building, spend 30 minutes understanding what already exists. Don't reinvent wheels, but understand which wheels work best for your use case. Claude works significantly better when it understands the solution space you're operating in and has concrete examples to reference or differentiate from.

The Research Framework

1. **Find Existing Solutions** Identify 3-5 tools that solve similar problems:
   - Direct competitors
   - Adjacent solutions
   - Tools that handle part of your workflow
2. **Analyze What Works/Doesn't Work** For each tool, document:

    - Good patterns to borrow: UI flows, features, interactions
    - Bad patterns to avoid: Complexity, confusion points, missing features
    - Gaps to fill: What they don't handle that you could
3. **Identify UI/UX Patterns** Screenshots are invaluable:
    - Clean, minimal interfaces that work for your user type
    - Interaction patterns (drag-drop, wizards, dashboards)
    - Layout approaches (mobile-first, sidebar navigation, etc.)

Research Template

```
## Competitive Analysis

### Tool 1: [Name]
**What it does well**: [Specific strengths]
**What it does poorly**: [Specific weaknesses]
**Key insight**: [What to learn from this]

### Tool 2: [Name]
**What it does well**: [Specific strengths]
**What it does poorly**: [Specific weaknesses]
**Key insight**: [What to learn from this]

### Design Patterns to Use
- [Specific UI pattern with description]
- [Interaction model that works well]

### Patterns to Avoid
- [Common mistakes in this space]
- [Complexity that doesn't add value]

### Opportunity Gap
[What none of these tools handle well that you could solve]
```

Feeding Research to Claude

Use this research to guide Claude's decisions:

> **Research Context**: I've analyzed 3 invoicing tools: Tool A: Great mobile UX but too complex for quick tasks Tool B: Fast but lacks professional invoice templates Tool C: Good templates but requires too many form fields **Reference**: Build something that combines Tool A's mobile-first approach with Tool B's speed, but uses better visual design than either. **Task**: Design the core user flow for invoice creation that learns from these insights.

Advanced: Screenshots as Context If you have screenshots of interfaces you like:

- Upload them to Claude Code's context
- Reference them specifically: "Use a layout similar to the attached screenshot but adapted for [your specific use case]"

Pro tip: Even simple sketches or wireframes help Claude understand your visual intent better than purely text descriptions.

Technical Foundation and AI Rules

This is where you establish the guardrails and guidelines that keep Claude focused and consistent throughout development. The goal isn't to over-specify, but to provide enough structure that Claude can make confident decisions within defined boundaries.

The Technical Stack Decision

Option A: Let Claude Help You Choose If you're not sure about technical choices:

> **Context**: Building [your app description] for [your user type] **Constraints**: [Any requirements like mobile-first, offline capability, etc.] **Task**: Recommend a technical stack (frontend, backend, database, deployment) that optimizes for rapid prototyping and ease of maintenance. Explain your reasoning.

Option B: Define Stack Upfront If you have preferences or constraints:

```
## Technical Foundation
**Frontend**: [React/Vue/Vanilla JS + styling approach]
**Backend**: [Node.js/Python/None for client-only apps]
**Database**: [PostgreSQL/SQLite/Firebase/None]
**Deployment**: [Vercel/Netlify/Railway/Local only]
**Key Libraries**: [Specific tools you want to use]
```

Claude Code Specific: The `CLAUDE.md` File

Critical: Create a `CLAUDE.md` file in your project root. Claude Code automatically pulls this into context for every conversation.

```
# Project Guidelines for Claude

## Project Overview
[Brief description of what you're building and why]

## Target User
[Your user profile from Stage 2]

## Technical Stack
- Frontend: Next.js 14 + Tailwind CSS
- Database: SQLite (local development)
- Deployment: Vercel

## Coding Standards
- Use TypeScript for all new files
- Follow functional component patterns in React
- Add comments for complex business logic
- No external dependencies without approval
- Mobile-first responsive design

## Architecture Principles
- Keep components small and focused
- Separate business logic from UI components
- Use consistent error handling patterns
- Follow RESTful API conventions

## Testing Requirements
- Add unit tests for utility functions
- Include integration tests for API endpoints
- Test mobile responsiveness manually

## Constraints
- Must work offline for core features
- Support modern browsers only (no IE)
- Keep bundle size under 500KB
- No external API dependencies in MVP
```

Note on Tool Curation: When defining the project in `CLAUDE.md`, it's also a good practice to explicitly curate Claude's list of allowed tools to ensure safety and efficiency. This means specifying which external commands or integrations Claude is permitted to use.

Rules for AI Behavior

Include specific instructions for how Claude should behave:

```
## AI Instructions

### Code Generation
- Always explain architectural decisions
- Ask clarifying questions if requirements are ambiguous
- Suggest improvements but don't implement without approval
- Generate complete, working code blocks (no placeholders)

### Problem Solving
- Start with the simplest solution that works
- Only add complexity when specifically requested
- Highlight potential issues or edge cases
- Suggest testing approaches for new features
```

```
    ### Communication
    - Use clear, structured explanations
    - Provide code comments in generated files
    - Flag when you're making assumptions
    - Offer alternatives when multiple approaches exist
```

Advanced: Custom Slash Commands and External Tools Store reusable prompts in `.claude/commands/` folder. You can also leverage `bash` tools and other command-line utilities by instructing Claude to use them.

File: `.claude/commands/test.md`

```
    # Test Generation Command
    Please generate comprehensive tests for: $ARGUMENTS
    Include:
    - Unit tests for core functions
    - Integration tests for API endpoints
    - Error handling test cases
    - Edge case scenarios
    Use our established testing patterns and ensure mobile compatibility.
```

Usage: Type `/test user authentication system` in Claude Code.

Why This Stage Matters: These guidelines become Claude's "working memory"-ensuring consistency across sessions and preventing common mistakes from recurring.

Prototyping with Claude Code

This is where intent becomes reality. Your approach here determines whether you get clean, maintainable code or an unmaintainable mess.

Key insight: With Claude Code, your IDE is a conversation. Success depends on structured communication, not just clear requests.

Session Setup Strategy

Environment Preparation

```
# Navigate to your project directory
cd your-project-name
# Initialize git (critical for version control)
git init
git add .
git commit -m "Initial project setup"
# Start Claude Code
claude-code
```

Opening Context Start each major session with a context-setting prompt:

> **Context**: Project: [Brief description from your vision doc] User: [User profile summary] Current goal: [What you want to accomplish this session] Files to consider: [Specific files if relevant] **Task**: [Specific, actionable request] Please ask any clarifying questions before starting.

The Structured Development Approach

Start Small and Specific. Don't ask for "the entire app." Break down into logical, testable pieces:

❌ **Bad first prompt**: "Build a PDF data extraction tool with drag-drop upload, table detection, and Excel export"

✅ **Good first prompt**: "Create a basic file upload component with drag-and-drop functionality. Style it with Tailwind CSS and show upload progress. Focus on the happy path for now."

Follow the Build-Test-Iterate Cycle

The core of Vibe Coding is a rapid feedback loop. Embrace common workflows:

- **Build**: Request a specific feature or component

- **Test**: Run the code, check the output
- **Iterate**: Refine based on what you see

This can be conceptualized as:

- **Explore, Plan, Code, Commit**: Understand the problem, outline a solution, implement, then save your progress.
- **Write Tests, Commit; Code, Iterate, Commit**: A test-driven approach where tests are written first, then code is developed to pass them.

Build - Test - Iterate - Build - Test - Iterate

Effective Prompting Patterns

Pattern 1: Feature-by-Feature Development

> **Session Goal**: Implement PDF upload and preview **Step 1**: "Create a file upload component that accepts PDFs and shows a preview" [Test and verify] **Step 2**: "Add table detection logic that identifies tabular data in PDFs" [Test with sample files] **Step 3**: "Connect the components into a complete upload-to-preview flow" [Test end-to-end]

Pattern 2: The Options Approach When you're not sure about implementation:

> I need to implement [specific feature]. Please provide 3 different approaches: The simplest solution that works A more robust solution with better error handling The most user-friendly solution For each approach, explain the tradeoffs.

Pattern 3: Context-Aware Modifications

> Looking at the current [component/file], I need to: [Specific change 1] [Specific change 2] Please modify the existing code while maintaining the current patterns and style.

Managing Context and Sessions

Claude Code has extensive context awareness, but it can get overwhelmed:

- Start fresh sessions for major new features
- Summarize progress when starting a new session

Create new Claude Code sessions when:

- Switching to a completely different feature
- The conversation gets too long (20+ exchanges)
- You want to change technical direction significantly
- The current session is making repetitive mistakes

Version Control Integration

Commit frequently during development:

```
# After each working feature
git add .
git commit -m "feat: Add PDF upload component with drag-drop"

# Before major changes
git tag v0.1 -m "Working file upload"
```

Claude Code can help with Git:

> Please review my current changes and suggest an appropriate commit message that follows conventional commit format.

Code Review and Quality Control

Don't accept everything blindly. You're still the architect.

Review checklist for Claude's output:

- ☐ Does it follow your coding standards?
- ☐ Are there obvious security issues?
- ☐ Is it overengineered for your needs?
- ☐ Does it match your user experience goals?
- ☐ Can you understand and maintain this code?

Common things to challenge:

"Can you simplify this implementation?" "Is this dependency necessary for our use case?" "How would this handle [specific edge case]?" "Can you add comments explaining the business logic?"

The goal of this stage: A working prototype that demonstrates your core value proposition and sets the foundation for iterative improvement.

Prompt Engineering and Iteration Strategy

Your first prompt will never be your last. Vibe Coding means iterating not just code, but the prompts themselves into increasingly effective instructions.

Core principle: Each conversation round should bring you closer to your intended outcome through strategic refinement.

The Prompt Evolution Framework

- **Level 1: Basic Request**

  > Build a user authentication system. *Problem*: Too vague, Claude will make many assumptions

- **Level 2: Specific Requirements**

  > Build a user authentication system with: Email/password registration and login JWT-based sessions Password reset functionality Input validation and error handling *Better*, but still missing context

- **Level 3: Context-Rich Instructions**

> **Context**: Building a B2B invoice tool for small business owners who need quick, secure access. **Task**: Implement user authentication with: Email/password registration and login JWT-based sessions (7-day expiry) Password reset via email Client-side validation with clear error messages Mobile-first responsive design **Constraints**: Use existing design system (Tailwind classes) Follow security best practices Keep forms simple (max 3 fields per step) No social login for MVP Please implement the registration flow first, then ask before proceeding to login. *Optimal*: Clear context, specific requirements, defined boundaries

Prompt Engineering Patterns

The Specification Pattern

```
## Feature Specification: Data Export
### User Story
As a [user type], I want to [action] so that [benefit].
### Acceptance Criteria
- [ ] [Specific testable requirement]
- [ ] [Performance requirement]
- [ ] [Edge case handling]
### Technical Requirements
- [Implementation constraints]
- [Integration points]
### Out of Scope
- [What NOT to build]
```

The Options Pattern

> I need to implement [specific feature]. Please analyze these requirements and suggest 3 approaches: **Requirements**: [List specific needs] For each approach, explain: Implementation complexity User experience impact Maintenance considerations Pros and cons I'll choose one and you can implement it.

Pattern 3: Context-Aware Modifications

> Looking at the current [component/file], I need to: [Specific change 1] [Specific change 2] Please modify the existing code while maintaining the current patterns and style.

Testing and Validation

You have a working prototype. Now verify it actually solves the problem you defined in Stage 1. This requires a two-layer approach: Technical functionality + Real-world usefulness.

Layer 1: Functional Testing (Does it work?)

Automated Testing with Claude's Help

> Please create comprehensive tests for the current application: Unit tests for core business logic functions Integration tests for the main user workflows Error handling tests for edge cases Performance tests for file processing Use [your preferred testing framework] and include both positive and negative test cases.

Manual Testing Checklist

```
## Testing Checklist
### Core Functionality
- [ ] Happy path works end-to-end
- [ ] Each feature works in isolation
- [ ] Error states display appropriately
### Cross-Platform Compatibility
- [ ] Works on desktop browsers (Chrome, Firefox, Safari)
- [ ] Functions properly on mobile devices
- [ ] Responsive design adapts correctly
### Performance Testing
- [ ] Loads within acceptable time limits
- [ ] Handles expected data volumes
```

```
### Security Validation
- [ ] Input validation prevents malicious data
- [ ] Authentication works correctly
```

Layer 2: User Validation (Does it solve the problem?)

Get the app in front of 1-3 actual users.

User Testing Script Template

```
## User Test Script
### Setup
"I'd like you to try using this tool. I'm testing the tool, not you,
so there are no wrong answers. Please think out loud as you use it."
### Task
"Your goal is to [realistic task that matches your user story]. Take
your time and let me know if anything is confusing."
### Observation Points
- Do they understand what the tool does?
- Can they complete the primary task without help?
- Where do they hesitate or show confusion?
```

Analyzing User Feedback with Claude

> Based on user testing, here's what I observed: User 1: [Specific behaviors and comments] User 2: [Specific behaviors and comments] Common patterns: [Issue that multiple users hit] [Unexpected user behavior] Please analyze this feedback and suggest: Critical UX improvements needed Changes to improve user success rate

The goal: Confidence that your application solves the intended problem for real users, with acceptable technical quality.

Deployment and Integration

Transform your validated prototype into a publicly accessible application.

Deployment Strategy with Claude's Help

> Please prepare this application for deployment: **Target platform**: [Vercel/Netlify/Railway/etc.] **Requirements**: Environment variable setup for [specific configs] Build optimization for production Error monitoring setup Generate: Deployment configuration files Environment variable template Build and deploy instructions

CI/CD Pipeline with Claude

> Create a simple CI/CD pipeline using GitHub Actions that: Runs tests on every push Builds the application for production Deploys automatically on the main branch Sends notifications on deploy success/failure

Example GitHub Actions Workflow:

```
name: Deploy to Production
on:
  push:
    branches: [ main ]
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Setup Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '18'
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
      - name: Build application
        run: npm run build
      - name: Deploy to Vercel
        uses: amondnet/vercel-action@v20
        with:
          vercel-token: ${{ secrets.VERCEL_TOKEN }}
          vercel-org-id: ${{ secrets.ORG_ID }}
          vercel-project-id: ${{ secrets.PROJECT_ID }}
```

You now have:

✅ A live, publicly accessible application ✅ Monitoring to detect issues ✅ Automated deployment process

Advanced Deployment and Automation

For more complex scenarios, consider these advanced concepts:

- **Headless Mode for Automation**: Claude Code can be run in non-interactive (headless) mode, allowing you to integrate it into automated scripts or CI/CD pipelines for tasks like automated code generation, refactoring, or bug fixes.
- **Multi-Claude Workflows (MCP)**: For highly complex problems, you can orchestrate multiple instances of Claude (or other AI agents) to work collaboratively on different parts of a problem. This involves breaking down a large task into smaller, interconnected sub-tasks, with each AI focusing on its specialized area, and then integrating their outputs.

Maintenance and Growth

Shipping is the beginning, not the end. Now you transition from building to improving based on real-world usage.

Continuous Improvement Framework

User Feedback Collection

- Create a lightweight in-app feedback widget.
- Set up user behavior tracking to understand feature usage.

Weekly Review Process

```
## Weekly Maintenance Review
### User Feedback Summary
- Positive feedback themes: [What's working well]
- Pain points: [What needs fixing]
- Feature requests: [What users want added]
### Technical Health
- Error rates: [Any spikes or patterns]
- Performance metrics: [Load times, API response]
### Priority Actions
1. [Critical fixes needed this week]
2. [Quick wins that improve user experience]
```

Using Claude for Analysis

> Based on this week's user feedback and analytics, please analyze the data and suggest: Immediate fixes that would have the biggest user impact. Potential root causes for common issues. Feature opportunities based on user behavior.

The goal: A stable, improving product that consistently delivers value to users while remaining enjoyable to maintain and extend.

Best Practices and Critical Pitfalls

These are the lessons that separate successful vibe coders from those who struggle.

Critical Best Practices

- **Foundation-First Approach**: Before opening Claude Code, write your `0_vision.md`, `1_user_profile.md`, and `CLAUDE.md`.
- **Prompt Engineering Excellence**: Use structured prompts (Context, Task, Constraints) and break down complex features into incremental stages.
- **Session and Context Management**: Start new chat sessions for new features to avoid context overload.
- **Quality Control**: You are the architect. Review every output. Challenge Claude to simplify, justify dependencies, and handle edge cases.

Critical Pitfalls to Avoid

Foundational Mistakes:

- **Wandering Prompts**: Starting without clear goals leads to feature creep.
- **Vague User Definition**: "For business users" leads to generic UX. Be specific.

Technical Pitfalls:

- **Over-Engineering from the Start**: Ask for "the simplest solution that works" first.
- **Ignoring Code Quality**: After a major feature, ask Claude to refactor and clean up the code.
- **Not Using Version Control**: Commit frequently. `git` is your safety net.

Process Pitfalls:

- **Accepting First Output**: The magic happens in iteration rounds 2-3.
- **Using Claude Like Google**: It's a collaborator, not a search engine. Guide the conversation.

UX and Product Pitfalls:

- **Building Without User Validation**: Get real users to try your prototype as early as possible.

Last but not least: Vibe Coding as a Core Skill

Vibe Coding isn't just a new development method-it's a fundamental shift in how software gets made and who gets to make it.

The New Skill Stack

Traditional Developer Skills	Vibe Coding Skills
Syntax mastery	Problem decomposition
Algorithm knowledge	Prompt engineering
Framework expertise	System thinking
Database design	User empathy & Quality judgment

The Force Multiplier Effect

Experienced developers don't become obsolete-they become force multipliers. A senior developer with vibe coding skills can:

- Prototype 10x faster.
- Focus on architecture while AI handles implementation.
- Validate ideas before committing team resources.

Your Journey Forward

This guide gives you the foundation, but mastery comes through practice. Start with small projects, build systematically, and focus on creating real value for real users. The opportunity is immense, and your discerning judgment will be the key to harnessing its full potential.

Happy building!

Michał Gołębiowski - linkedin.com/in/hello-michal-golebiowski/