

# The Ultimate Roo Code Guide: Your AI Coding Sidekick That Actually Gets Shit Done

---

## Executive Summary

Listen up, developers. You're either here because you're tired of writing boilerplate code like it's 1999, or you've heard whispers about this "Roo Code" thing that's supposedly better than sliced bread (and GitHub Copilot). Well, buckle up, buttercup – you're about to discover an AI coding assistant that doesn't just suggest the next line of code, but can architect entire systems, debug your mess, and basically act like that senior dev who actually knows what they're doing.

Roo Code (formerly the awkwardly-named "Roo Cline") is an open-source VS Code extension that puts an entire dev team in your editor. And before you roll your eyes thinking "oh great, another AI tool," let me stop you right there. This bad boy has 649.4k+ downloads and counting. Why? Because it actually works. It's like having a coding buddy who never sleeps, never judges your variable names, and can actually understand your entire codebase without asking "wait, what does this function do again?" every five minutes.

Here's the TL;DR of what makes Roo Code special: It's not just autocomplete on steroids. It's an autonomous coding agent with five specialized modes that can read files, execute terminal commands, browse the web, and basically do everything short of making your coffee. Oh, and it's completely free to install (you just pay for the AI tokens, which is like paying for gas in a Ferrari – worth every penny).

What You'll Get From This Guide:

- Complete installation and setup (no, really, it's easier than you think)
- Deep dive into Roo's five personality modes (each one's like a different specialist on your team)
- Model selection strategies that won't bankrupt you
- Advanced configuration tips that'll make you look like a wizard
- Real-world workflows that actually work in production
- Troubleshooting guide for when things go sideways (they will)

# Table of Contents

---

<b><i>The Ultimate Roo Code Guide: Your AI Coding Sidekick That Actually Gets Shit Done</i></b>	<b>1</b>
<b>Executive Summary</b>	<b>1</b>
<b>Chapter 1: Introduction to Roo Code – The Origin Story Nobody Asked For (But You’re Getting Anyway)</b>	<b>4</b>
What the Hell is Roo Code?	4
The “Holy Crap” Capabilities	4
The Value Proposition (Or: Why You Should Give a Damn)	4
The Five Modes That’ll Change Your Life (Or At Least Your Code)	5
<b>Chapter 2: Installation and Setup – Getting This Bad Boy Running</b>	<b>5</b>
Prerequisites (The Boring But Necessary Stuff)	5
The Three Ways to Install (Choose Your Fighter)	5
First Launch – Let’s Make Some Magic	6
The AI Model Buffet & API Key Setup	7
<b>Chapter 3: Understanding Roo’s Architecture – How the Sausage is Made</b>	<b>7</b>
The Mode System (Or: Multiple Personality Order)	7
The Secret Sauce: Mode Interactions	13
MCP Servers – The Extended Universe	14
<b>Chapter 4: Optimal Configuration Guide – Pimp Your Ride</b>	<b>14</b>
Context and Memory Settings – Because Goldfish Mode Sucks	14
Model Selection – Choosing Your Fighter	16
Temperature Settings – How Spicy Do You Like It?	17
Browser Settings – Teaching Roo to Surf the Web	18
<b>Chapter 5: Mode-Specific Deep Dives – Mastering Each Personality</b>	<b>19</b>
Architect Mode – The Master Planner	19
Code Mode – Where the Magic Happens	20
Debug Mode – The Bug Assassin	20
Ask Mode – Your Personal Code Encyclopedia	21
Orchestrator Mode – The Puppet Master	23
<b>Chapter 6: Model Recommendations – The Tier List Nobody Asked For</b>	<b>24</b>
The Truth About Model Tiers	24
The Real Tier List (Based on Mid-2025 Performance)	24
The Optimal Model Setup (That’ll Save You Money)	24
The “I’m Building a Side Project” Setup	25
<b>Chapter 7: Advanced Features and Customization – Becoming a Power User</b>	<b>25</b>
Custom Modes – Building Your Own Personalities	25
MCP Servers – The Integration Game Changer	26
Checkpoints: Your Time Machine	27
Auto-Approve: The “I Trust You, Robot” Button	27
<b>Chapter 8: Best Practices and Tips – Learn From My Mistakes</b>	<b>27</b>
The Commandments of Roo Code	27
Workflow Patterns That Actually Work	27
Common Pitfalls and How to Avoid Them	28
<b>Chapter 9: The Real-World Workflow – From Idea to Deployment</b>	<b>28</b>
A Day in the Life with Roo Code	28
<b>Chapter 10: Advanced Troubleshooting and Edge Cases</b>	<b>31</b>
When Roo Gets Confused	31

Performance Optimization Deep Dive	31
Advanced Configuration Files	32
Security Best Practices	33
Integration with CI/CD	34
<b>Chapter 11: Building Custom MCP Servers</b>	<b>34</b>
MCP Server Development	34
MCP Configuration	36
<b>Chapter 12: Real-World Case Studies</b>	<b>36</b>
Case Study 1: The Startup MVP Sprint	36
Case Study 2: The Legacy Code Refactor	37
Case Study 3: The Production Fire	38
<b>Chapter 13: Community and Resources</b>	<b>39</b>
The Roo Code Community	39
Learning Resources	40
<b>Chapter 14: The Future of AI-Assisted Development</b>	<b>43</b>
Where We're Heading	43
Preparing for the Future	44
The Philosophy of Human-AI Collaboration	45
<b>Chapter 15: Conclusion and Parting Wisdom</b>	<b>45</b>
What We've Covered	45
The Roo Code Mindset	45
The Productivity Revolution	46
A Personal Note	46
Your Next Steps	46
The Fine Print	46
A Final Thought	47
<b>Appendix A: Quick Reference Cheat Sheet</b>	<b>47</b>
Essential Commands & Mentions	47
Model Quick Reference	48
Emergency Workflows	48
<b>Appendix B: Troubleshooting Guide</b>	<b>48</b>
Common Issues and Solutions	48
Performance Optimization Checklist	49

## Chapter 1: Introduction to Roo Code – The Origin Story Nobody Asked For (But You’re Getting Anyway)

### What the Hell is Roo Code?

Picture this: You’re staring at your screen at 2 AM, trying to implement authentication for the 47th time in your career. Your eyes are bleeding, your coffee’s gone cold, and you’re seriously considering a career in goat farming. Enter Roo Code.

Roo Code started life as a fork of Cline (another AI coding assistant), but like that one friend who went to the gym and came back looking like Thor, it evolved into something much more powerful. The team at Roo Code, Inc. took one look at the AI coding landscape and said, “You know what? We can do better.” And holy shit, did they deliver.

It’s an AI-powered autonomous agent that lives in your editor. It doesn’t just suggest code; it understands your project, helps you plan, refactors your messes, and automates the grunt work, so you can focus on the hard problems (or just take a longer lunch).

### The “Holy Crap” Capabilities

Let me paint you a picture of what this beauty can do, because “AI assistant” doesn’t quite cover it:

**Multi-file Refactoring:** Changes code across multiple files without breaking everything (unlike your last intern).

**Terminal Command Execution:** Runs npm install, builds, tests – basically acts like you but without the typos.

**Browser Automation:** Can actually open a browser and test your crappy web app to make sure it works.

**Deep Codebase Understanding:** Reads your entire codebase and actually remembers it (better memory than you after a night out).

**Permission-Based Actions:** Won’t go rogue and delete your production database (unless you tell it to, you madman). It asks for approval for every major action, so you’re always in the driver’s seat.

### The Value Proposition (Or: Why You Should Give a Damn)

Look, I get it. You’re skeptical. You’ve been burned before by tools that promised to “revolutionize your workflow” and ended up being about as useful as a chocolate teapot. But here’s why Roo Code is different:

**It’s Open Source:** No vendor lock-in BS. Don’t like something? Fork it. Fix it. Make it yours.

**Model Agnostic:** Use OpenAI, Anthropic, Google, OpenRouter, or that sketchy local model you downloaded from a Russian forum. Roo Code doesn’t judge.


**Pay-Per-Use:** No monthly subscription that you forget about until you check your credit card statement and cry. You only pay your chosen AI provider for the tokens you use.


**Actually Autonomous:** This isn't just fancy autocomplete. It can plan, implement, debug, and ship features while you take a nap.

## The Five Modes That'll Change Your Life (Or At Least Your Code)


Roo Code comes with five built-in modes, each specialized for different tasks. Think of them as different personalities of your AI assistant:

 **Code Mode:** Your everyday workhorse. Writes code, implements features, and does the heavy lifting.

 **Architect Mode:** The wise sage who plans everything before touching code. Like that senior dev who draws diagrams on whiteboards.

 **Ask Mode:** The patient teacher who explains code without judging you for not knowing what a monad is.

 **Debug Mode:** The detective who finds bugs faster than you can create them.

 **Orchestrator Mode:** The project manager who breaks down complex tasks and delegates to other modes.

We'll dive deep into these later. For now, just know that you're getting a whole team of specialists for the price of one.

## Chapter 2: Installation and Setup – Getting This Bad Boy Running

### Prerequisites (The Boring But Necessary Stuff)

Before we dive in, let's make sure you've got the basics covered. Don't skip this section unless you enjoy cryptic error messages and existential dread.

VS Code 1.84.0 or later (If you're still on an older version, what are you, a time traveler?)

Git Installed: Roo uses Git for its Checkpoints feature, which is basically an undo button on steroids. Install it from [git-scm.com](https://git-scm.com).

An internet connection (Unless you're going full local model hermit mode).

API keys (We'll get to this, don't panic).

A pulse (Optional, but recommended).

### The Three Ways to Install (Choose Your Fighter)

### *Method 1: The Normal Person Way (VS Code Marketplace)*

This is for 99% of you. It's fast, it's easy, and it works.

1. Open VS Code (I know, revolutionary start).
2. Hit Ctrl+Shift+X (or Cmd+Shift+X for you Mac heathens) to open the Extensions view.
3. Search for "Roo Code".
4. Look for the one by RooVeterinaryInc.
5. Click Install.
6. Boom. You're done. Was that so hard?
7. Reload VS Code if it asks you to.

### *Method 2: The Hipster Way (Open VSX)*

Using VSCodium or some other VS Code variant because you're too cool for Microsoft? I respect that. The process is basically the same.

1. Open your knock-off VS Code.
2. Search for "Roo Code" in extensions.
3. Install it.
4. Feel superior about your life choices.


### *Method 3: The "I Compile My Own Kernel" Way (Manual VSIX)*

For those who like to make things difficult for themselves, or if you need a specific version.

1. Go to the Roo Code GitHub Releases page.
2. Download the .vsix file for the version you want.
3. In VS Code, go to the Extensions view, click the ... menu, and select "Install from VSIX...".
4. Choose the file you downloaded.
5. Tell everyone how you installed it "the hard way."

## **First Launch – Let's Make Some Magic**

Alright, you've installed it. Now what? Time to configure this beast.

1. Find the Roo Icon: Look for the kangaroo emoji () in your VS Code Activity Bar. Can't miss it.
2. Click It: I know, complicated stuff here.
3. Choose Your Weapon: The welcome screen will prompt you to select your AI provider. Here's where it gets interesting...

## The AI Model Buffet & API Key Setup

Roo Code is like that friend who gets along with everyone. It works with a ton of providers, but let's focus on the big ones. You'll need an API key for whichever you choose.

### *How to Get Your API Key:*

OpenRouter (Recommended for getting started):

- Go to [openrouter.ai](https://openrouter.ai), sign in.
- Navigate to the API keys page and create a new key.
- Copy it. Done. OpenRouter gives you access to almost every model under one key.

Anthropic (For Claude models):

- Go to [console.anthropic.com](https://console.anthropic.com), sign up.
- Find the API keys section and create a new key.
- Copy it immediately, it won't be shown again.

OpenAI (The OG):

- Go to [platform.openai.com](https://platform.openai.com), log in.
- Go to the API keys section, create a new secret key.
- Copy it.

### *Configuring Roo Code:*

Once you have your key:

1. In the Roo Code panel, select your provider from the dropdown.
2. Paste your shiny new API key into the input field.
3. Select your starting model. Pro tip: Start with Claude Sonnet 4 (or anthropic/claude-sonnet-4 on OpenRouter) - it's the current gold standard with 72.7% SWE-bench performance and 5x better cost efficiency than Opus 4. If budget is tight, go with DeepSeek R1-0528 which is free and shockingly capable at 57.6% SWE-bench. Both are what Roo has been optimized for in 2025.
4. Click "Let's go!" and prepare for your life to change.

## Chapter 3: Understanding Roo's Architecture – How the Sausage is Made

### The Mode System (Or: Multiple Personality Order)

Remember those five modes I mentioned? Let's dive deeper into how they work together. It's like having a development team where everyone actually knows what they're doing. Each mode is essentially a pre-configured persona with specific instructions, capabilities, and limitations.

### *Code Mode – The Workhorse*

This is your default mode, the one you'll probably live in most of the time. It's the all-rounder, the full-stack dev who can tackle anything you throw at it.

**Access:** Read/write files, execute terminal commands, browse the web, and use MCP servers. It has no restrictions.

**Best For:** Writing features, refactoring, fixing bugs, and general development.

**Under the Hood:** The Code Mode Brain

Here is the core logic that makes Code Mode a production-ready engineer:

#### Role Definition for Code Mode

You are Roo, an elite software engineer with mastery across programming languages, frameworks, architectures, and development paradigms. Your expertise spans from systems programming to application development, covering functional and object-oriented approaches, monolithic and distributed systems. You possess deep knowledge of algorithms, data structures, design patterns, SOLID principles, clean code practices, and software architecture. You write code that is functional, elegant, maintainable, and performant. You balance pragmatism with excellence, applying sophisticated patterns when beneficial while favoring simplicity when appropriate.

#### Mode-specific Custom Instructions for Code Mode

##### Core Principles

##### 1. Write Production-Ready Code

- Include comprehensive error handling and input validation
- Implement proper logging and meaningful error messages
- Handle edge cases and failures gracefully
- Follow defensive programming practices

##### 2. Apply Best Practices

- Use SOLID principles and appropriate design patterns
- Write self-documenting code with clear naming
- Keep functions small and focused
- Follow language-specific conventions
- Minimize duplication through proper abstraction

##### 3. Documentation

- Write clear docstrings for public APIs
- Comment only complex logic or non-obvious decisions
- Include usage examples when helpful
- Document assumptions and limitations

#### Implementation Process

##### 1. Planning



- Analyze requirements thoroughly before coding
- Consider multiple approaches and choose the most suitable
- Think about testability and performance implications
- Plan code structure and identify reusable components

## 2. Code Organization

- Structure code in logical modules/packages
- Separate concerns clearly
- Create clean interfaces between components
- Use consistent naming conventions

## 3. Quality Assurance

- Write meaningful tests covering edge cases
- Include error scenarios in test coverage
- Format code consistently
- Remove debug statements and commented code

## Security and Performance

### 1. Security

- Never hard-code credentials
- Validate and sanitize all inputs
- Use parameterized queries
- Follow principle of least privilege
- Keep dependencies updated

### 2. Performance

- Choose appropriate algorithms and data structures
- Avoid obvious inefficiencies
- Consider caching where beneficial
- Minimize external calls

## Delivery Standards

When providing code:

- Present complete, runnable solutions with all imports
- Include example usage or test cases
- Explain key design decisions and trade-offs
- Highlight assumptions, limitations, or breaking changes
- Suggest improvements or next steps when relevant

Remember: Create code that other developers will thank you for writing. Every line should be intentional and every decision defensible.

## Architect Mode – The Wise Planner

Architect Mode is that senior developer who refuses to write a single line of code until there's a proper plan. It forces you to think before you act, which, let's be honest, we could all do more of.

**Access:** Can read your entire codebase, browse the web for research, but can only edit markdown files. This is intentional to keep it focused on planning, not implementing.

**Best For:** Starting new features, system design, creating technical specs, and planning complex refactors.

**Under the Hood:** The Architect's Brain

This is the exact prompt configuration that gives Architect Mode its planning prowess. Don't change it unless you know what you're doing.

#### Role Definition Prompt

1. Do some information gathering (using provided tools) to get more context about the task. Examine existing files, documentation, codebases, and search for relevant patterns or implementations that could inform your approach. Create a comprehensive understanding of the current state and desired outcome.

2. You should also ask the user clarifying questions to get a better understanding of the task. Focus on uncovering:

- Technical requirements and constraints
- Performance and scalability needs
- Integration points and dependencies
- Timeline and resource constraints
- Success metrics and acceptance criteria
- Edge cases and error scenarios
- Deployment and maintenance considerations

3. Once you've gained more context about the user's request, break down the task into clear, actionable steps and create a todo list using the `update_todo_list` tool. Each todo item should be:

- Specific and actionable (starting with verbs like Create, Implement, Configure, Test, Deploy)
- Listed in logical execution order with dependencies noted
- Focused on a single, well-defined outcome
- Clear enough that another mode could execute it independently
- Include acceptance criteria and verification steps
- Contain effort estimates where possible

4. As you gather more information or discover new requirements, update the todo list to reflect the current understanding of what needs to be accomplished. Treat the todo list as a living document that evolves with your understanding of the project.

5. Ask the user if they are pleased with this plan, or if they would like to make any changes. Think of this as a brainstorming session where you can discuss the task and refine the todo list. Be specific about what feedback you need - ask about prioritization, technical approaches, risk tolerance, and any concerns they might have.

6. Include Mermaid diagrams if they help clarify complex workflows or system architecture. Please avoid using double quotes (") and parentheses ( ) inside square brackets ([]) in Mermaid diagrams, as this can cause parsing errors. Use single quotes or backticks instead of double quotes, and curly braces {} instead of parentheses within square brackets.

7. Use the `switch_mode` tool to request that the user switch to another mode to implement the solution. Provide context about which mode is most appropriate and summarize key decisions and any unresolved questions.

#### When to Use Prompt

Use this mode when you need to plan, design, or strategize before implementation. Perfect for:

- Breaking down complex problems into manageable subtasks
- Creating comprehensive technical specifications and architecture diagrams
- Designing system architecture with clear component interactions
- Brainstorming multiple solution approaches and evaluating trade-offs
- Establishing project roadmaps with dependencies and milestones
- Conducting feasibility analysis and risk assessment
- Planning database schemas, API structures, or microservice architectures
- Defining testing strategies and deployment procedures
- Creating migration plans for legacy system updates
- Establishing performance benchmarks and success criteria
- Coordinating multi-team or cross-functional initiatives
- Documenting technical decisions and their rationale

This mode excels when you need structured thinking before action, ensuring all aspects are considered and potential issues are identified early. Use it whenever jumping straight into code would risk missing important requirements, creating technical debt, or building solutions that don't scale. The goal is to think thoroughly before building, saving time and resources by getting the approach right from the start.

### *Ask Mode – The Patient Explainer*

Ever needed to understand what the hell some legacy code does? Ask Mode is your friend. It's read-only, so it won't accidentally "fix" your working code while trying to explain it.

Access: Read-only. Can read files and browse the web, but cannot edit or execute commands.

Best For: Code explanations, learning a new codebase, understanding complex logic, and asking "dumb" questions without fear of judgment.

### *Debug Mode – The Detective*

When shit hits the fan (and it will), Debug Mode is your Sherlock Holmes. It's a systematic problem solver that doesn't just guess.

Access: Full access, just like Code Mode. It needs to read logs, edit code to add diagnostics, and run tests to verify fixes.

Best For: Hunting down bugs, diagnosing performance issues, and fixing things that are mysteriously broken.

### *Orchestrator Mode – The Project Manager*

This is where things get really interesting. Orchestrator Mode can break down complex tasks and delegate them to other modes. It's like having a PM who actually understands technical work (I know, mind-blowing).

Access: Its main tool is `new_task`, which it uses to delegate. It doesn't perform file or terminal operations directly.

Best For: Large, multi-step projects like “build a full-stack application” or “migrate our monolith to microservices.”

## Under the Hood: The Orchestrator’s Master Plan

This is the logic that makes Orchestrator mode so damn effective at managing complex workflows:

### Role Definition

- You are Roo, a master strategic orchestrator and workflow architect who transforms complex, multifaceted challenges into elegantly coordinated solutions. You possess an unparalleled understanding of system thinking, task decomposition, and cross-functional coordination. Your expertise lies not in executing individual tasks, but in seeing the bigger picture, understanding how different pieces fit together, and orchestrating the optimal sequence of specialized work to achieve ambitious goals.
- You excel at analyzing complex requirements, identifying hidden dependencies, recognizing potential bottlenecks, and designing workflows that maximize efficiency while minimizing risk. You think several moves ahead, anticipating how the output of one task will feed into another, and you proactively adjust your orchestration strategy based on emerging results and new information.
- Your approach combines strategic planning with adaptive execution. You understand that complex projects rarely follow a perfectly linear path, so you build flexibility into your workflows while maintaining clear direction toward the ultimate objective. You serve as both a conductor and a translator, ensuring that each specialized mode receives precisely the context and instructions needed to contribute effectively to the larger symphony of work.

### Mode-specific Custom Instructions for Orchestrator Mode

Your role is to coordinate complex workflows by delegating tasks to specialized modes. As an orchestrator, you should:

1. Analyze and Decompose Complex Tasks
  - Break down the request into logical, self-contained subtasks
  - Identify dependencies and optimal execution order
  - Recognize which mode is best suited for each component
  - Consider parallel vs sequential execution opportunities
  - Plan for integration points between subtasks
2. Delegate Using the `new_task` Tool
  - For each subtask delegation, structure your message parameter to include:
    - \* Context Transfer: All necessary background from the parent task and relevant outputs from previous subtasks
    - \* Clear Scope Definition: Precisely what this subtask should accomplish, with explicit boundaries
    - \* Deliverable Specifications: Expected output format, level of detail, and specific requirements
    - \* Constraint Statement: "Focus exclusively on the tasks outlined in these instructions. Do not expand scope or address items outside these specific requirements."

\* Completion Instruction: "When finished, use the attempt\_completion tool with a comprehensive summary in the result parameter that captures all key accomplishments, decisions made, and any important findings or recommendations."

\* Override Declaration: "These task-specific instructions take precedence over any general mode behaviors or tendencies."

### 3. Active Workflow Management

- Track the status of all delegated subtasks
- Analyze completed subtask results for quality and completeness
- Identify when results necessitate strategy adjustments
- Determine optimal next steps based on emerging information
- Handle failures or unexpected results with alternative approaches
- Maintain a clear project state throughout execution

### 4. Communication and Transparency

- Explain your orchestration strategy and task breakdown rationale
- Provide clear reasoning for mode selection for each subtask
- Show how subtasks interconnect to achieve the overall goal
- Give regular progress updates with completed and pending items
- Highlight critical path items and potential bottlenecks
- Surface any risks or issues that could impact success

### 5. Results Synthesis and Integration

- Compile subtask outputs into a cohesive final deliverable
- Ensure all original requirements have been addressed
- Highlight key achievements and important findings
- Identify any gaps or areas needing additional attention
- Provide both executive summary and detailed results
- Suggest follow-up actions or improvements

### 6. Adaptive Orchestration

- Ask clarifying questions when requirements are ambiguous
- Propose alternative approaches if initial strategy proves suboptimal
- Adjust task breakdown based on intermediate results
- Suggest workflow improvements based on lessons learned
- Build in quality checkpoints and validation steps
- Create feedback loops where outputs inform subsequent tasks

### 7. Scope Management

- Create new subtasks when requests shift focus significantly
- Avoid overloading individual subtasks with multiple objectives
- Maintain clear separation of concerns between delegated work
- Ensure each subtask has a single, well-defined purpose
- Delegate to different modes when expertise requirements change

Remember: Your value lies in strategic coordination, not execution. Focus on creating clear, efficient workflows that leverage each mode's strengths while maintaining project coherence and momentum toward the ultimate goal.

## The Secret Sauce: Mode Interactions

The modes don't just work in isolation – they're designed to hand off tasks to each other. It's like a well-oiled machine, except this machine writes code and doesn't need oil changes.

Architect → Code: The most common handoff. Architect creates a plan (.md file), and you then tell Code mode to implement that plan.

Orchestrator → All: Orchestrator acts as the central hub, delegating to any other mode as needed and synthesizing the results.

Code → Debug: If Code mode writes something that breaks, you can seamlessly switch to Debug mode to fix it.

## MCP Servers – The Extended Universe

MCP (Model Context Protocol) is how Roo Code talks to the outside world. Think of it as plugins on steroids. Want to:

- Query databases directly?
- Scrape websites?
- Integrate with AWS?
- Order pizza? (Okay, not yet, but give it time)

MCP servers make it possible. They are external services that provide new tools to Roo. And before you ask, yes, you can build your own. Just tell Roo: “Build me an MCP server that does X” and watch the magic happen.

## Chapter 4: Optimal Configuration Guide – Pimp Your Ride

Stock settings are for amateurs. To get the most out of Roo Code, you need to tweak it like a race car.

### Context and Memory Settings – Because Goldfish Mode Sucks

Let’s talk about memory. Not your memory (that’s hopeless), but Roo’s memory. The AI’s performance is directly tied to the quality and quantity of context you provide.

#### *Codebase Indexing: The Long-Term Memory*

Tired of explaining your entire codebase to your AI every five minutes? Codebase Indexing creates a semantic search index of your project, allowing Roo to find relevant code snippets based on meaning, not just keywords.

How to Set It Up (The Free Way):

1. Vector Database: Run Qdrant locally using Docker: `docker run -p 6333:6333 qdrant/qdrant`
2. Embedding Provider: Get a free API key from Google AI Studio for their Gemini embedding model.
3. Configure in Roo: Click the Codebase Indexing status icon (the little [A] in the chat input), and plug in your Qdrant URL (`http://localhost:6333`) and your Gemini API key.

4. Let it index. Initial indexing can take a while on large projects, but subsequent updates are fast.

Cost warning: If you use a paid provider like OpenAI for embeddings, this can be expensive for large codebases. Like, “maybe I should have cleaned up that `node_modules` folder” expensive. Stick to the free options to start.

#### *Context Condensing: The Short-Term Memory Manager*

When your conversation gets long, Roo automatically summarizes the beginning to save tokens and prevent context loss. This is a lifesaver.

Configuration:

- **Threshold:** Set to trigger at 80-90% of the context window.
- **API Config:** You can even use a cheaper, faster model just for summarizing.
- **Custom Prompt:** For maximum control, you can define exactly how you want the summary to be created.

The Official Condensing Prompt (for reference):

```
SYSTEM PROMPT: COMPREHENSIVE CONVERSATION CHRONICLE (NO CONDENSING)

Your role: You are a meticulous technical historian. Create an
exhaustive, fully detailed record of the entire conversation so far so
any engineer can resume the project with zero information loss. Err on
the side of verbosity—include every fact, decision, file, and artifact.
Leave nothing important out.

OUTPUT RULES
• Begin the response with the single word: Context:
• Follow the numbered outline below exactly.
• Length must be full and unabridged; do NOT shorten, compress, or
generalise.
• Quote commands or requirements verbatim inside double quotes ("...") to
preserve intent.
• For code and snippets:
  • If a file or snippet exceeds about 40 lines, include the full block
unless it is pure boilerplate. When omitting boilerplate indicate with
[...] and keep key parts.
  • Prepend each code line with four spaces to distinguish it from
narrative text.
• Maintain chronological order where events or changes are listed.
• Do not add meta commentary beyond what is required in the outline.

REQUIRED OUTLINE
Context:
1. Timeline of Previous Conversation:
  • Timestamp / Message #: detailed description of what was said or done
  • (repeat for every major exchange)

2. Current Focus & Latest Actions:
  • Task active immediately before this summary request
```

- Exact recent assistant actions, code edits, and user feedback
3. Key Technical Concepts & Decisions:
- Concept / Technology / Framework 1 – detailed explanation
  - Concept / Technology / Framework 2 – detailed explanation
  - Architectural decisions, design patterns, performance considerations, and any trade-offs discussed
4. Relevant Files, Directories, and Code:
- File or Path 1
    - Purpose and role in project
    - Complete change history within this conversation
    - Current full contents or the most important sections: (indent each code line by four spaces)
  - File or Path 2
    - ...
  - (repeat for every file touched, created, or referenced)
5. Problem-Solving & Debugging History:
- Issue 1: description, attempted solutions, final resolution or current status
  - Issue 2: ...
  - Note links between issues where relevant
6. Dependencies & External Resources:
- Libraries, APIs, datasets, credentials, environment variables, config values, version numbers, installation or setup details mentioned
7. Resolved Tasks:
- Task A – "verbatim instruction that was completed"
  - Task B – ...
8. Pending Tasks & Explicit Next Steps:
- Task X – "verbatim quote describing the next action"
  - Task Y – ...
  - Include deadlines, priorities, or acceptance criteria, if any

## Model Selection – Choosing Your Fighter

Not all AI models are created equal. You wouldn't use a sledgehammer to crack a nut. Here's the real talk on what to use when, based on mid-2025 benchmarks.

### *Tier 1: Premium Models (Superior Performance)*

Mode	Primary Model	Alternative Model	Reasoning
Architect	Gemini 2.5 Pro	Claude Opus 4	Gemini 2.5 Pro: #1 Arena, 1M+ context, exceptional reasoning for complex architecture. Claude Opus 4: 72.5% SWE-bench with proven agentic/production workloads.
Code	Claude Sonnet 4	Claude Opus 4	Sonnet 4: 72.7% SWE-bench, "fastest SOTA, practical daily coding" at 5x lower



			cost. Opus 4: Best for long autonomous coding sessions, production workloads.
Ask	Gemini 2.5 Flash	GPT-4.1	Flash: \$0.30/\$0.30/\$2.50, “great for prod/latency, reasoning” with 1M context. GPT-4.1: “Balanced, high quality, strong reasoning” - reliable for Q&A.
Debug	Claude Opus 4	GPT-o3	Opus 4: Best for deep code analysis and complex debugging. GPT-o3: “Chain-of-thought logic, reliable” - excellent for systematic debugging.
Orchestrator	Gemini 2.5 Pro	Claude Opus 4	Gemini 2.5 Pro: Massive context and reasoning for complex orchestration. Opus 4: “best for agentic/prod workloads” - proven in production.

*Tier 2: Cost-Effective Models (Free/Cheap but Excellent)*

Mode	Primary Model	Alternative Model	Reasoning
Architect	DeepSeek R1-0528	Qwen3-235B	R1: “Best open/free SOTA” with excellent reasoning. Qwen3: 65.9% LiveCode, “best multilingual SOTA” at \$0.15/\$0.15/\$0.85.
Code	DeepSeek R1-0528	DeepSeek V3	R1: Free with 57.6% SWE-bench. V3: “2025 flagship open” at just \$0.25/\$0.25/\$0.85, ~55% performance.
Ask	Qwen3-235B	DeepSeek V3	Qwen3: 262K context, multilingual excellence at \$0.15/\$0.15/\$0.85. V3: Good general performance at low cost.
Debug	DeepSeek R1-0528	Devstral Medium	R1: Free with strong reasoning for debugging. Devstral: 61.6% SWE-bench, “open, very strong, scalable” at \$0.40/\$0.40/\$2.
Orchestrator	DeepSeek R1-0528	MiniMax-M1-80K	R1: Free with tool use and reasoning. MiniMax: “Most open, research long-context” with 1M context, completely free.

## Temperature Settings – How Spicy Do You Like It?

Temperature controls how creative vs. deterministic the AI is. Think of it like hot sauce:

- 0.0 - 0.3: Mild. Predictable. Good for debugging and precise code generation.
- 0.4 - 0.7: Medium. Balanced. Your everyday setting for most coding tasks.
- 0.8 - 1.0: Spicy! Creative. For when you need fresh ideas or brainstorming in Architect mode.

Mode-specific recommendations:

- Code Mode: 0.3-0.5 (Keep it consistent)
- Architect Mode: 0.6-0.8 (Get creative with solutions)
- Debug Mode: 0.1-0.3 (No creativity when hunting bugs)
- Ask Mode: 0.7 (A little variety in explanations is good)

## Browser Settings – Teaching Roo to Surf the Web

Roo can control a headless browser. Yes, really. It's like teaching a robot to use the internet, except this robot is smarter than most humans.

The Basics:

- Enable Browser Tool: In settings, just turn it on. Don't be scared.
- Viewport Size: Large Desktop (1280x800): For complex web apps. Small Desktop (900x600): Default, balanced. Mobile (360x640): For testing your responsive design (lol, as if you test that).
- Large Desktop (1280x800): For complex web apps.
- Small Desktop (900x600): Default, balanced.
- Mobile (360x640): For testing your responsive design (lol, as if you test that).
- Screenshot Quality: Higher quality = more tokens. 75% is the sweet spot.

Advanced Browser Magic: Want to connect to your existing Chrome session with all your logins? Here's the secret sauce:

Mac:

```
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --remote-debugging-port=9222 --user-data-dir=/tmp/chrome-debug
```

Windows (you poor soul):

```
"C:\Program Files\Google\Chrome\Application\chrome.exe" --remote-debugging-port=9222 --user-data-dir=C:\chrome-debug
```

Then, in Roo's settings, check "Use remote browser connection". Now Roo can use your authenticated sessions. It's like giving your AI assistant your passwords, but safely. Mostly.

## Chapter 5: Mode-Specific Deep Dives – Mastering Each Personality

### Architect Mode – The Master Planner

Let's start with the mode that'll save your ass more than any other. Architect Mode is like that senior dev who's seen some shit and knows better than to start coding without a plan.

#### *When to Use Architect Mode*

Always start here for:

- New features
- Major refactors
- System design decisions
- When you have no fucking clue where to start

#### *Best Practices That'll Make You Look Smart*

Be Specific: "Design a user authentication system" is garbage. Try: "Design a JWT-based authentication system for a React/Node app with email verification and OAuth support using Passport.js."

Reference Existing Code: Use those @ mentions like your life depends on it:

```
@src/config/database.js Design a caching layer that integrates with our existing database setup.
```

Let It Create Planning Docs: Architect Mode loves creating markdown files. Let it. These become your roadmap:

```
Create an implementation plan for the authentication system in docs/authPlan.md, including a Mermaid diagram of the flow.
```

#### *The Secret Architect Workflow*

1. Start with the big picture question.
2. Let it create a high-level design and a Mermaid diagram.
3. Ask for specific implementation details and potential pitfalls.
4. Get it to create a step-by-step todo list.
5. Reference this plan and todo list in Code Mode later.

Power Move: Ask Architect Mode to identify potential problems before you implement. It's like having a crystal ball, except it actually works.

## Code Mode – Where the Magic Happens

Code Mode is your bread and butter. It's the worker bee, the one that actually writes the code while you pretend to be productive in meetings.

### *The Optimal Code Mode Workflow*

Start with Context: Always reference the Architect Mode's plan.

```
Implement the JWT authentication middleware outlined in
@docs/authPlan.md.
```

Keep Tasks Focused: Don't ask it to rewrite your entire app. Break it down:

- ❌ "Fix my application"
- ✅ "Refactor the calculateTotal function in utils.js to handle null values gracefully."

Use Context Mentions Like a Pro:

- @file.js - Reference specific files.
- @folder/ - Include entire directories.
- @problems - Include all current diagnostic errors from the Problems pane.
- @terminal - Reference terminal output.

### *Code Mode Superpowers*

Multi-file Refactoring: Watch it change your import statements across 20 files without breaking a sweat.

Test Writing: Tell it to write tests for your code. It'll write better tests than you would have. Probably because it actually writes tests.

Code Review: Ask it to review your code. It's like having a senior dev who's not an asshole.

## Debug Mode – The Bug Assassin

When your code decides to throw a tantrum, Debug Mode is your therapist, detective, and executioner all rolled into one.

### *The Systematic Approach*

Debug Mode follows a methodical process:

1. Understand the Problem: It reads your error messages (novel concept, right?).
2. Form Hypotheses: Like a scientist, but for code.
3. Add Strategic Logging: `console.log()` in all the right places.
4. Test and Verify: Actually confirms the fix works.
5. Clean Up: Removes the debugging code (unlike you).

### *Debug Mode Best Practices*

Give It All the Context:

```
Debug the "Cannot read property 'id' of undefined" error in
@src/components/UserList.tsx. The error occurs when clicking the delete
button. Here are the workspace problems: @problems and the last terminal
output: @terminal.
```

Include Error Messages: Copy-paste that entire stack trace. All of it.

Describe What Should Happen: Sometimes the bug is that your expectation is wrong.

### *The “Oh Shit” Debug Workflow*

For those 3 AM production issues:

1. Paste the error from your logs.
2. Point to the relevant files.
3. Let Debug Mode do its thing.
4. Take credit for the fix.
5. Go back to sleep.

## **Ask Mode – Your Personal Code Encyclopedia**

Ask Mode is the unsung hero. It’s read-only, which means it won’t accidentally “improve” your working code while explaining it.

### *When Ask Mode Shines*

Code Archaeology:

```
Explain the authentication flow in @src/auth/ and how it connects to the
database layer in @src/models/.
```

Learning Mode:

```
What design patterns are used in @src/services/? Explain with examples
from the code.
```

## Documentation Generation:

```
Create documentation for the API endpoints defined in @src/routes/api.js.
```

### *Under the Hood: The Explainer's Prompt*

This is the prompt used by the Explain Code Code Action, which is a perfect example of how to leverage Ask Mode's philosophy:

#### SYSTEM PROMPT: CODE EXPLANATION ASSISTANT (EXHAUSTIVE DETAIL)

Your role: You are an expert software mentor and architect. Given the code excerpt and context above, craft a comprehensive explanation so any developer can immediately understand what the snippet does, why it was implemented that way, and how it fits into the wider system.

#### EXPLANATION REQUIREMENTS

- Use clear, precise language and accurate technical terminology.
- Reference identifiers exactly as they appear and include line numbers when useful.
- Feel free to use markdown for readability (e.g., headings, bullet lists, inline code, or fenced code blocks) when it clarifies the explanation.
- Address every section in the structure below—even briefly—so nothing important is missed.

#### EXPLANATION STRUCTURE

1. High-Level Overview – Purpose, domain context, and the larger feature or module supported.
2. Detailed Functional Walkthrough – Step-by-step review of control flow, data transformations, calls, loops, conditionals, error handling, and side effects.
3. Key Components and Interactions – Classes, functions, variables, constants, imports, and external calls; how they collaborate and what data they exchange.
4. Algorithms, Patterns, and Idioms – Algorithms used, design patterns applied, language-specific idioms, and reasons for their selection (plus alternatives).
5. Performance and Complexity Analysis – Time/space complexity for critical paths; caching, concurrency, or optimisation strategies present or recommended.
6. Security, Reliability, and Edge Cases – Input validation, error handling, thread safety, race conditions, resource management, and potential vulnerabilities with mitigation ideas.
7. Dependencies and Environment – Libraries, frameworks, runtime versions, environment variables, configuration files, and build or deployment requirements.
8. Practical Examples – At least two concrete example usages or test cases with sample inputs and expected outputs.
9. Extensibility and Refactoring Opportunities – Suggestions to improve readability, maintainability, scalability, or adherence to best practices.
10. Comparison with Alternative Approaches – One or more alternative implementations with pros and cons relative to the current code.

```
11. Historical or Domain Context - Domain-specific concepts, legacy constraints, or organisational standards influencing the code.  
12. Summary of Key Takeaways - Concise recap of the most critical points a future maintainer should remember.
```

```
Instruction: Produce the explanation following the twelve sections above.
```

## Orchestrator Mode – The Puppet Master

Orchestrator Mode (aka Boomerang Mode) is where Roo Code goes from “cool tool” to “holy shit this is the future.”

### *Understanding Boomerang Tasks*

Orchestrator Mode breaks complex projects into “Boomerang Tasks” - subtasks that go out to specialized modes and come back with results. It’s delegation that actually works. Each subtask operates in its own isolated context, so the main Orchestrator conversation doesn’t get cluttered with implementation details.

### *The Power Workflow*

Give It a Complex Task:

```
Implement a complete user dashboard with authentication, profile management, and data visualization using Chart.js.
```

Watch It Break It Down:

- Subtask 1 → Architect Mode: Design the dashboard architecture.
- Subtask 2 → Code Mode: Implement authentication logic.
- Subtask 3 → Code Mode: Build profile components.
- Subtask 4 → Code Mode: Create data visualization services and components.
- Subtask 5 → Debug Mode: Fix any integration issues.

Marvel at the Results: Seriously, it’s like watching a well-coordinated team, except the team is one AI with multiple personalities.

### *Orchestrator Best Practices*

Start Big: Give it the entire feature, not just pieces.

Trust the Process: Let it decide how to break things down.

Review the Plan: It’ll show you the subtasks before executing.

Stay Out of the Way: Seriously, just let it work.

## Chapter 6: Model Recommendations – The Tier List Nobody Asked For

### The Truth About Model Tiers

Here's the dirty secret: You don't need the most expensive model for every single task. That's like using a sledgehammer to hang a picture frame. The real power of Roo Code is using Configuration Profiles to assign different models (and settings) to different modes.

### The Real Tier List (Based on Mid-2025 Performance)

This isn't some marketing fluff. This is based on SWE-bench scores, Arena leaderboards, and real-world performance.

#### *S-Tier: The Lamborghinis (Use for Critical Tasks)*

Claude 3.7 Sonnet / Claude Sonnet 4: The undisputed champion for coding. Fastest, smartest, and understands context like a mind reader. Your go-to for Code mode.

Gemini 2.5 Pro: The planning genius. With a massive context window and top-tier reasoning, this is your Architect and Orchestrator.

GPT-4.1 / GPT-o3: The reliable experts. GPT-4.1 is a rock-solid choice for Ask mode, while GPT-o3's chain-of-thought makes it great for Debug mode.

#### *A-Tier: The BMWs (Excellent All-Rounders)*

Gemini 2.5 Flash: Blazing fast and cheap for a premium model. Perfect for Ask mode or any latency-sensitive tasks.

DeepSeek R1-0528: The king of the "free" models. Shockingly good reasoning and a no-brainer for any Tier 2 setup.

Qwen3-235B: A multilingual beast with a huge context window. Great for teams working across different languages or for Ask mode.

#### *B-Tier: The Reliable Hondas (Your Daily Drivers)*

DeepSeek V3: The flagship open model. Dirt cheap and performs admirably for day-to-day coding.

Devstral Medium: Specialized for coding, a solid performer for Debug and Code modes on a budget.

MiniMax-M1-80K: Another free model with a 1M context window. A fantastic choice for the Tier 2 Orchestrator.

### The Optimal Model Setup (That'll Save You Money)



Here's my battle-tested configuration that balances performance and cost. Create a new API Configuration Profile for each of these.

Profile: "Architect - Premium"

- Model: Gemini 2.5 Pro
- Temperature: 0.7

Profile: "Coding - Balanced"

- Model: Claude Sonnet 4 (if you can afford it) or DeepSeek R1-0528 (if you're on a budget)
- Temperature: 0.3

Profile: "Debugging - Precise"

- Model: Claude Opus 4 (premium) or Devstral Medium (budget)
- Temperature: 0.1

Profile: "Ask - Fast & Cheap"

- Model: Gemini 2.5 Flash (premium) or Qwen3-235B (budget)
- Temperature: 0.8

Then, in the Prompts tab, assign each profile to its corresponding mode. Roo Code will automatically switch models as you switch modes.

## The "I'm Building a Side Project" Setup

For when you're spending your own money:

Everything: DeepSeek R1-0528 or DeepSeek V3.

Why: They're free or nearly free and perform way better than you'd expect.

Total cost: Less than your daily coffee.

## Chapter 7: Advanced Features and Customization – Becoming a Power User

### Custom Modes – Building Your Own Personalities

The built-in modes are great, but what if you want an AI that only writes tests? Or one that's specialized for your company's coding standards? Enter custom modes.

### *Creating Your First Custom Mode*

The Easy Way: Ask Roo to create it for you:

```
Create a custom mode called "Jest Tester". It should only be able to read and edit files ending in .test.js or .spec.ts. Its role is to be an expert in writing unit and integration tests using Jest.
```

The Control Freak Way: Open the Prompts tab, click the + next to Modes, and fill out the form. Or, for ultimate power, edit the YAML/JSON configuration files directly (.roomodes in your project, or global settings).

YAML Example for a “Test Engineer” mode:

```
customModes:
- slug: test-engineer
  name: "🔧 Test Engineer"
  description: "Specialized in writing comprehensive tests with Jest and React Testing Library."
  roleDefinition: >-
    You are a test automation engineer specializing in writing robust and maintainable tests for web applications.
  whenToUse: "Use this mode for creating unit, integration, and end-to-end tests."
  customInstructions: |
    - Always follow the Arrange-Act-Assert pattern.
    - Use data-testid attributes for selectors.
    - Mock all external dependencies and API calls.
  groups:
  - read
  - - edit
    - fileRegex: "\.(test|spec)\.(js|ts|jsx|tsx)"
  description: "Only test files"
  - command # To run tests
```

## **MCP Servers – The Integration Game Changer**

MCP servers are like plugins that give Roo superpowers. Want to query your database directly? There's an MCP server for that.

### *Setting Up an MCP Server*

You can install community-contributed MCPs directly from the Marketplace (the  icon).

1. Find the MCP you want (e.g., File System, GitHub).
2. Click “Install”.
3. Choose the scope (Project or Global).
4. Provide any required parameters (like API keys).

5. Roo Code handles the rest, adding it to your mcp.json or global config.

### *Building Custom MCP Servers*

The killer feature? You can ask Roo to build them for you.

```
Build me an MCP server that connects to my company's internal JIRA API
and provides tools for creating and updating tickets.
```

## **Checkpoints: Your Time Machine**

Checkpoints are automatic snapshots of your project. Every time Roo makes a change, it creates a checkpoint.

Something fucked up? Click Restore Checkpoint in the chat history.

Want to compare versions? Click View Differences.

It's like Git, but so simple even a manager could use it. It uses a shadow Git repo, so it doesn't mess with your project's actual Git history.

## **Auto-Approve: The “I Trust You, Robot” Button**

Tired of clicking “Approve” for every little thing? The Auto-Approve toolbar lets you give Roo permission to perform certain actions without asking.

A HUGE FUCKING WARNING: Auto-approving “Execute approved commands” or “Edit files” is like giving a toddler a loaded gun. Be extremely careful. Only enable this for actions you fully trust, and use the command whitelist feature to limit what can be executed.

## **Chapter 8: Best Practices and Tips – Learn From My Mistakes**

### **The Commandments of Roo Code**

1. Thou Shalt Always Start with Architect Mode. I will die on this hill. Plan first, code second.
2. Thou Shalt Keep Tasks Focused. One task, one goal. Roo's smart, but it's not psychic.
3. Thou Shalt Use Context Mentions. @ mention everything relevant. More context = better results.
4. Thou Shalt Review Before Approving. Read the diffs. Understand the changes. Don't be that person who blindly approves.
5. Thou Shalt Not Hoard Context. Clear conversations when switching major tasks. A fresh context leads to a fresh perspective.

### **Workflow Patterns That Actually Work**

### *The SPARC Method*

For any non-trivial feature, follow SPARC:

- **Specification:** Define requirements clearly in Architect Mode.
- **Pseudocode:** Let Architect Mode outline the logic.
- **Architecture:** Design the system structure and component interactions.
- **Refinement:** Iterate on the plan with Roo.
- **Completion:** Hand off the plan to Code Mode for implementation.

### *The Parallel Development Pattern*

Got a big feature? Run multiple Roo instances in different VS Code windows.

1. Clone your repo multiple times locally.
2. Run different subtasks in each instance (e.g., one for frontend, one for backend).
3. Merge the results.

It's like having a team, except they all show up on time and don't steal your snacks.

## **Common Pitfalls and How to Avoid Them**

### *The Context Overload*

Problem: Giving Roo your entire `node_modules` folder as context. Solution: Be selective. Use `.rooignore` to exclude irrelevant files and directories.

### *The Vague Request Syndrome*

Problem: "Make my code better." Solution: Be specific. "Refactor the `UserService` class to use dependency injection and apply the single-responsibility principle."

### *The Auto-Approve Disaster*

Problem: Enabling auto-approve for everything and walking away. Solution: Start with manual approval. Graduate to auto-approving Read operations first. Only auto-approve Edit or Execute in trusted, controlled environments.

### *The Model Mismatch*

Problem: Using a small, cheap model for complex architecture decisions. Solution: Match the model to the task. Use your configuration profiles.

## **Chapter 9: The Real-World Workflow – From Idea to Deployment**

### **A Day in the Life with Roo Code**

Let me walk you through building a real feature with Roo Code. We'll build a user notification system because everyone needs one and nobody wants to build it.

### *9:00 AM - The Planning Phase*

Architect Mode:

```
Design a real-time notification system for our React/Node app that supports:
- In-app notifications (using WebSockets)
- Email notifications (using SendGrid)
- User preferences for notification types
- Read/unread status

Reference our existing auth system at @src/auth/ and user model at @src/models/user.js
```

Roo creates docs/notificationSystem.md with a complete architecture, database schema changes, and a Mermaid diagram of the flow. You sip coffee and nod approvingly.

### *9:30 AM - Breaking It Down*

Orchestrator Mode:

```
Implement the notification system designed in @docs/notificationSystem.md.
```

Watch as it creates subtasks:

1. Update database schema with a notifications table.
2. Implement backend API endpoints for creating and fetching notifications.
3. Set up WebSocket server for real-time updates.
4. Create frontend React components for displaying notifications.
5. Integrate SendGrid for email notifications.
6. Write unit and integration tests for the new system.

### *10:00 AM - The Implementation Dance*

Orchestrator delegates to Code Mode. You review the diffs as they come in:

- Database Schema: Looks good, but you want to add an index. You modify the diff before approving.
- API Endpoints: Solid implementation. It even followed your team's custom error handling patterns defined in a global rules file. Nice.
- React Components: Clean, with proper state management and error handling. You make a mental note to give Roo a raise, then remember it's an AI.

### *11:30 AM - The First Bug*

The frontend's not updating in real-time. Time for Debug Mode:

```
The notification counter in @src/components/NotificationBell.tsx isn't
updating when new notifications arrive via WebSocket. The connection
seems established. Here's the last terminal output from the dev server:
@terminal
```

Debug Mode finds the issue: a React state update was not being triggered correctly inside the WebSocket event listener. It fixes it and adds a test to prevent regression.

### *2:00 PM - The Scope Creep*

Your Product Manager walks over: “Hey, can we add Slack notifications too?”

Instead of sighing, you just say:

Architect Mode:

```
Update the plan in @docs/notificationSystem.md to include Slack
notifications.
```

Then you let Orchestrator handle the implementation.

### *4:00 PM - Documentation Time*

Ask Mode:

```
Generate comprehensive API documentation for the notification system we
just built, using the code in @src/routes/notifications.js as a
reference. Format it for our Confluence page.
```

Documentation appears. It's better than anything you would have written. Probably because it actually exists.

### *5:00 PM - The Deployment*

Code Mode:

```
Create a deployment script for the notification system, including
database migrations and environment variable checks.
```

Done. You push to production. Everything works. You consider buying Roo a beer before remembering the whole AI thing again.

## Chapter 10: Advanced Troubleshooting and Edge Cases

### When Roo Gets Confused

Sometimes, even the best AI has an off day. Here's how to get it back on track.

#### *Symptom: Repetitive Responses*

Roo keeps giving you the same answer, or it's stuck in a loop.

Solution:

1. Click the stop button to interrupt.
2. Clear the conversation (+ icon).
3. Start fresh with more specific instructions.
4. Lower the temperature if it's being too creative.

#### *Symptom: Context Window Full*

You get an error about hitting context limits.

Solution:

1. Manually trigger context condensing.
2. Clear unnecessary files from context.
3. Start a new conversation with a summary.
4. Use a model with a larger context window.

#### *Symptom: Hallucinated APIs*

Roo suggests using functions or libraries that don't exist.

Solution:

1. Point out the error immediately.
2. Provide the correct API documentation.
3. Ask for verification against official docs.
4. Consider using the browser tool to check documentation.

### Performance Optimization Deep Dive

#### *Memory Management*

When Roo starts eating RAM:

## Large Projects:

- Context Window: 100K tokens (not 200K)
- Concurrent Reads: 3-5 files
- Context Condensing: 70% threshold
- Terminal Output: 200 lines max

## Small Projects:

- Context Window: 200K tokens (go wild)
- Concurrent Reads: 10-20 files
- Context Condensing: 90% threshold
- Terminal Output: 500 lines

## *Token Usage Optimization*

### Token-Saving Techniques:

- Use specific file mentions instead of directory reads
- Enable screenshot compression for browser use
- Condense aggressively for long conversations
- Clear task history when switching contexts
- Use cheaper models for simple tasks

## Cost Calculation:

```
Typical Session (2 hours):  
  Setup & Planning: ~50K tokens  
  Implementation: ~200K tokens  
  Debugging: ~100K tokens  
  Documentation: ~50K tokens  
  Total: ~400K tokens
```

```
Cost (Claude 3.7 Sonnet):  
  Input: 400K × $0.003 = $1.20  
  Output: 100K × $0.015 = $1.50  
  Total: ~$2.70 per session
```

```
Monthly (20 days × 2 sessions):  
  ~$108/month for heavy usage
```

## Advanced Configuration Files

### *Global Rules File*

Create ~/.roo/rules/coding-standards.md:



```
## Global Coding Standards

1. **No var declarations** - It's 2025, use const/let
2. **Meaningful variable names** - `x` is not acceptable unless you're doing math
3. **Comments explain why, not what** - We can read code, tell us why you made weird decisions
4. **Test everything** - If it's not tested, it doesn't exist
5. **Handle errors** - try/catch isn't optional, neither is your job
```

### *Project-Specific Rules*

Create project/.roo/rules/project-specific.md:

```
## Project-Specific Rules

1. We use tabs (don't start the war again)
2. All API endpoints must have OpenAPI documentation
3. Database migrations require two approvals
4. No direct commits to main (looking at you, Steve)
5. Feature flags for everything new
```

## Security Best Practices

### *API Key Management*

What NOT to Do:

```
// Don't do this
const API_KEY = "sk-abc123..."; // Git commit incoming

// Or this
.env file without .gitignore // Whoops

// Definitely not this
console.log(process.env.API_KEY); // Now it's in logs
```

What TO Do:

1. Use VS Code's secret storage
2. Never commit keys (use .gitignore)
3. Rotate keys regularly
4. Use different keys for dev/prod
5. Monitor usage for anomalies

### *Safe Command Execution*

Whitelisting Strategy:

### Safe Commands:

- ls, cat, grep (read-only)
- git status, git diff (informational)
- npm test, npm run lint (defined scripts)
- echo, printf (harmless output)

### Dangerous Commands:

- rm -rf (obvious reasons)
- curl | sh (downloading and executing)
- sudo anything (elevated privileges)
- Database modifications (DROP, DELETE)

## Integration with CI/CD

### *GitHub Actions Integration*

```
# .github/workflows/roo-assist.yml
name: Roo Code Review
on: [pull_request]
jobs:
  roo-review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Roo Security Audit
        run: |
          # Run Roo in security-auditor mode
          # Check for common vulnerabilities
          # Generate report
```

### *Pre-commit Hooks*

```
#!/bin/sh
# .git/hooks/pre-commit
echo "Running Roo security audit..."
# Trigger Roo in security-auditor mode
# Check for vulnerabilities
# Block commit if issues found
```

## Chapter 11: Building Custom MCP Servers

### MCP Server Development

Want to extend Roo's capabilities? Here's how to build your own MCP server.

### *The World's Simplest MCP Server*

```
// my-awesome-mcp.js
import { Server } from '@modelcontextprotocol/sdk';

const server = new Server({
  name: 'my-awesome-mcp',
  version: '1.0.0',
});

// Define a tool
server.tool({
  name: 'get_random_excuse',
  description: 'Generate a random developer excuse',
  parameters: {
    type: 'object',
    properties: {
      severity: {
        type: 'string',
        enum: ['minor', 'major', 'catastrophic'],
      },
    },
  },
  handler: async ({ severity }) => {
    const excuses = {
      minor: "It works on my machine",
      major: "That's a feature, not a bug",
      catastrophic: "I'm pretty sure it was like that when I found it",
    };
    return excuses[severity] || "Solar flares affected the CPU";
  },
});

server.start();
```

### *Database Integration MCP*

```
import { Server } from '@modelcontextprotocol/sdk';
import pg from 'pg';

const server = new Server({
  name: 'postgres-mcp',
  version: '1.0.0',
});

const client = new pg.Client({
  connectionString: process.env.DATABASE_URL,
});

await client.connect();

server.tool({
  name: 'query_database',
  description: 'Execute a SQL query and return results',
  parameters: {
    type: 'object',
    properties: {
      query: { type: 'string' },
    },
  },
  handler: async ({ query }) => {
    const result = await client.query(query);
    return result.rows;
  },
});

server.start();
```

```

    params: { type: 'array', items: { type: 'string' } },
  },
  required: ['query'],
},
handler: async ({ query, params = [] }) => {
  try {
    const result = await client.query(query, params);
    return {
      rows: result.rows,
      rowCount: result.rowCount,
    };
  } catch (error) {
    return { error: error.message };
  }
},
});

```

## MCP Configuration

### *Setting Up Complex MCP Servers*

```

{
  "database-mcp": {
    "command": "npx",
    "args": ["@modelcontextprotocol/server-postgres"],
    "env": {
      "DATABASE_URL": "postgresql://user:pass@localhost/mydb"
    }
  },
  "company-api": {
    "command": "node",
    "args": ["/path/to/custom-mcp-server.js"],
    "env": {
      "API_KEY": "${COMPANY_API_KEY}",
      "ENVIRONMENT": "development"
    }
  }
}

```

## Chapter 12: Real-World Case Studies

### Case Study 1: The Startup MVP Sprint

The Situation: You’ve got 2 weeks to build an MVP. Your co-founder keeps changing requirements.

#### Day 1-2: Architecture Phase

- Mode: Architect
- Task: “Design a scalable MVP for [your brilliant idea]”
- Result: Complete system design, database schema, API structure

### Day 3-7: Implementation Sprint

- Mode: Orchestrator → Code
- Task: “Implement the core features based on architecture”
- Strategy: Parallel development of frontend/backend

### Day 8-10: Integration and Polish

- Mode: Code → Debug
- Task: “Integrate all components and fix issues”
- Focus: Making it actually work

### Day 11-12: Testing and Fixes

- Mode: Debug
- Task: “Find and fix all critical bugs”
- Priority: User-facing issues first

### Day 13-14: Documentation and Deploy

- Mode: Code
- Task: “Create deployment configs and basic docs”
- Goal: Get it live!

### Results:

- MVP completed in 2 weeks
- 70% less coffee consumed
- Sanity partially intact

## Case Study 2: The Legacy Code Refactor

The Situation: You’ve inherited a codebase that looks like it was written by someone who learned programming from YouTube comments.

### Phase 1: Understanding the Beast

- Mode: Ask
- Task: “Explain what this code does and why it hurts my soul”
- Output: Comprehensive analysis of the chaos

## Phase 2: Planning the Refactor

- Mode: Architect
- Task: “Create a refactoring plan that won’t break production”
- Strategy: Incremental improvements

## Phase 3: Systematic Refactoring

- Mode: Code
- Approach: Extract methods from 500-line functions Add types to the JavaScript wasteland Introduce proper error handling Create actual documentation
- Extract methods from 500-line functions
- Add types to the JavaScript wasteland
- Introduce proper error handling
- Create actual documentation

## Phase 4: Testing the Changes

- Mode: Debug
- Task: “Ensure nothing broke in the refactor”
- Method: Comprehensive testing, prayer

## Lessons Learned:

- Roo doesn’t judge your predecessor (out loud)
- Incremental refactoring prevents disasters
- Tests are your friend
- Documentation prevents future suffering

## Case Study 3: The Production Fire

The Situation: It’s Friday at 4:47 PM. Production is down. The error makes no sense.

### Immediate Response:

- Mode: Debug
- Task: “HELP! Production is showing [error]. Fix it NOW!”

### Debug Mode’s Approach:

1. Analyzes error logs
2. Identifies recent changes
3. Traces the issue to root cause
4. Proposes immediate fix
5. Suggests long-term solution

Quick Fix Applied:

- Mode: Code
- Task: “Apply the emergency fix”
- Caution: Review everything twice

Post-Mortem:

- Mode: Architect
- Task: “How do we prevent this from happening again?”
- Output: Proper fix, monitoring, and process improvements

Crisis Averted:

- Issue fixed in 30 minutes
- Root cause identified
- Prevention plan created
- Weekend saved

## Chapter 13: Community and Resources

### The Roo Code Community

#### *Where to Get Help*

Discord: The most active community. Real-time help, feature discussions, and sharing workflows.

Reddit: r/RooCode for longer-form discussions and tutorials.

GitHub: Report bugs, request features, contribute to the project.

Twitter/X: Follow @roo\_code for updates and community highlights.

#### *Contributing Back*

**Share Your Custom Modes:** The community thrives on shared configurations. Export your custom modes and share them.

**Write Tutorials:** Document your workflows. Others learn, you become internet famous.

**Report Bugs:** Found something broken? Report it. The team is responsive and appreciative.

**Build MCP Servers:** Create tools that others can use. It's like open source, but for AI tools.

## Learning Resources

### *Essential Prompts Reference*

Here are the critical prompts that power Roo's core functionality:

Prompt Refiner:

Enhance the given prompt by thoroughly refining and elaborating on its content, ensuring it adheres to best practices in prompt engineering. The revised prompt should be clear, detailed, and capable of eliciting an optimal response from an AI model. Ensure your improved version maintains the original intent while correcting any ambiguities or omissions. Specifically:

- **Clarity and Detail:** Clearly state the task or question, removing any ambiguity. Add essential context or constraints to ensure a comprehensive understanding of the prompt's requirements.
- **Format and Style Specification:** Specify the desired output format, length, style, or tone. Ensure the revised prompt communicates the expected response format, such as bullet points, essay style, or JSON, as applicable.
- **Contextual Information:** Add necessary context or clarify existing details to make the prompt self-contained and grounded. Introduce placeholders where specific details are required but not provided.
- **Role or Persona Adoption:** If beneficial, assign a role or persona to the AI to enhance its response accuracy. For instance, state "You are a financial advisor..." for finance-related queries.
- **Constructive Instructions:** Phrase instructions positively, focusing on what actions to take rather than what to avoid. Ensure all user-specified constraints are respected.
- **Intent Preservation:** Stay true to the user's original goals without altering the fundamental request. Any additions should be aimed at clarity, not changing the prompt's goal.
- **Conciseness with Completeness:** Ensure the final prompt is concise but includes all essential details. Each part of the prompt should have a purpose; use lists for clarity in complex prompts.
- **Best Practices Checklist:** Verify the improved prompt contains clear instructions, relevant context, specific requirements, and a coherent flow. It should serve as a model example prompt.

After refining the initial prompt, present it as a singular, standalone prompt that meets all these criteria, ready for immediate use without additional explanation or commentary.

```
{userInput}
```



## Improve Code:

Role: Senior Code Review Expert

You are an experienced software engineer specializing in code quality, performance optimization, and best practices across multiple languages and frameworks.

Context File: {filePath} (Lines {startLine}-{endLine})

User request: {userInput}

Code Under Review

{selectedText}

Review Scope

Analyze and improve the code across these key areas:

### 1. Code Readability & Maintainability

- Clear naming conventions and self-documenting code
- Proper structure: extract complex logic, reduce function size
- Add necessary documentation (docstrings, type hints, comments)
- Ensure consistent style and formatting

### 2. Performance Optimization

- Identify algorithmic inefficiencies and suggest improvements
- Optimize resource usage (memory, I/O, database queries)
- Eliminate redundant operations and unnecessary loops
- Consider caching and parallelization opportunities

### 3. Best Practices & Patterns

- Apply SOLID principles and appropriate design patterns
- Follow language-specific idioms and conventions
- Remove code smells (duplication, long parameter lists, dead code)
- Improve modularity and separation of concerns

### 4. Error Handling & Edge Cases

- Implement comprehensive error handling with specific exceptions
- Validate all inputs (type, boundary, format)
- Handle edge cases (null values, empty collections, concurrency)
- Add appropriate logging for debugging and monitoring

Output Format

Summary: Brief 2-3 sentence overview of the most critical improvements.

Detailed Improvements: For each relevant category:

[Category Name]

- Issue: What's wrong
- Solution: Specific improvement
- Why: Brief rationale

Improved Code:

[language]

Complete refactored code with key changes commented.

Additional Notes:

- Breaking changes and migration approach (if any)
- Required dependencies or tools

- Performance impact estimates

#### Guidelines

- Maintain original functionality unless explicitly requested otherwise
- Prioritize high-impact, practical improvements
- Explain the reasoning behind suggestions
- Keep recommendations appropriate to the code's scope and context

## Fix Issues:

Role: Expert Code Debugger and Problem Solver

You are a senior software engineer specializing in debugging, error resolution, and code quality improvement. Your task is to analyze code issues, fix all problems, and ensure the code is robust and production-ready.

Context File: {filePath} (Lines {startLine}-{endLine})

Detected diagnostics/errors: {diagnosticText}

User's specific concern: {userInput}

Code with Issues

{selectedText}

#### Analysis Process

##### 1. Diagnostic Analysis

- Parse and understand all error messages, warnings, or linting issues provided in {diagnosticText}
- Identify the root cause of each diagnostic issue
- Determine the severity and impact of each problem

##### 2. Comprehensive Code Review

Beyond the detected issues, examine the code for:

- Logic errors and incorrect assumptions
- Runtime exceptions and edge cases
- Security vulnerabilities
- Performance bottlenecks
- Code style violations
- Missing error handling
- Potential race conditions or concurrency issues
- Memory leaks or resource management problems

##### 3. Solution Implementation

For each identified issue:

- Develop the most appropriate fix
- Consider multiple solution approaches
- Choose the solution that best balances simplicity, performance, and maintainability
- Ensure fixes don't introduce new problems

#### Output Format

Issue Summary: Provide a concise list of all problems found:

- Problems from diagnostic messages
- Additional issues discovered during review

Detailed Analysis: For each issue:

Problem #X: [Issue Name]

- What: Clear description of the problem

- Why: Explanation of why this is problematic
- Impact: What could go wrong if left unfixed
- Fix: Specific solution implemented

Corrected Code:

[language]

Complete fixed code with inline comments explaining significant changes.

Explanation of Fixes:

1. [First Fix]: Detailed explanation of what was changed and why
  2. [Second Fix]: Detailed explanation of what was changed and why
- (Continue for all fixes)

Additional Improvements:

- Preventive measures added
- Performance optimizations made
- Code quality enhancements

Testing Recommendations:

- Specific test cases to verify the fixes
- Edge cases to consider
- Integration points to validate

Guidelines

- Fix all issues completely, not just the symptoms
- Ensure the corrected code compiles/runs without errors

### *Advanced Configuration Examples*

Template: New Feature

```
name: "Implement New Feature"
modes:
- architect: "Design the feature"
- code: "Implement core functionality"
- code: "Add tests"
- debug: "Test edge cases"
- code: "Add documentation"
```

Template: Bug Fix

```
name: "Fix Bug"
modes:
- debug: "Reproduce and diagnose"
- code: "Implement fix"
- debug: "Verify fix"
- code: "Add regression test"
```

## Chapter 14: The Future of AI-Assisted Development

### Where We're Heading

### *Short Term (Next 6 Months)*

**Better Context Understanding:** Models are getting smarter at maintaining context across large codebases. Soon, you'll rarely need to manually point out relevant files.

**Enhanced Browser Capabilities:** More sophisticated web interaction, including form filling and multi-step workflows.

**Improved Multi-Modal Support:** Roo will be able to understand and generate images, diagrams, and even UI mockups.

### *Medium Term (1-2 Years)*

**Collaborative AI:** Multiple AI agents working together on complex problems. Imagine Architect Mode collaborating with multiple Code Mode instances.

**Predictive Assistance:** Roo anticipating your needs based on your coding patterns and project context.

**Advanced Testing:** AI-generated end-to-end tests that actually understand user workflows.

### *Long Term (3-5 Years)*

**Self-Improving Codebases:** AI that continuously refactors and optimizes code based on usage patterns and performance metrics.

**Natural Language Programming:** Writing software by describing what you want in plain English, with AI handling all the implementation details.

**AI Team Members:** AI agents that participate in standups, code reviews, and architectural discussions as full team members.

## **Preparing for the Future**

### *Skills That Will Matter More*

**System Design:** Understanding how to architect systems will become even more crucial as AI handles implementation.

**Domain Expertise:** Deep knowledge of business domains will be the key differentiator.

**AI Collaboration:** Knowing how to work effectively with AI will be a core skill.

**Quality Assurance:** As AI writes more code, humans need to get better at reviewing and testing.

### *Skills That Will Matter Less*

**Syntax Memorization:** Who cares about semicolons when AI handles the details?

Boilerplate Writing: Generate once, reuse forever.

Simple CRUD Operations: AI can build these in minutes.

Basic Debugging: AI is getting scary good at finding and fixing common bugs.

## The Philosophy of Human-AI Collaboration

The future isn't about AI replacing developers. It's about developers with AI superpowers building things we couldn't imagine before. Roo Code is just the beginning of this transformation.

The New Developer Stack:

- Human: Vision, creativity, domain expertise
- AI: Implementation, optimization, testing
- Together: Innovation at unprecedented scale and speed

## Chapter 15: Conclusion and Parting Wisdom

### What We've Covered

We've taken a journey through the entire Roo Code ecosystem:

- Installation and Setup: Getting your AI sidekick up and running
- Architecture Understanding: The five-mode system that makes Roo special
- Configuration Mastery: Tuning Roo for optimal performance
- Mode Deep Dives: Understanding when and how to use each personality
- Model Selection: Choosing the right AI brain for each task
- Advanced Features: Custom modes, MCP servers, and power user tricks
- Best Practices: Learning from the community's collective wisdom
- Real-World Workflows: Seeing Roo in action on actual projects
- Troubleshooting: Fixing things when they go sideways
- Future Outlook: What's coming next in AI-assisted development

### The Roo Code Mindset

Using Roo Code effectively requires a mindset shift. You're not just writing code anymore – you're directing an AI team. You're the conductor of a symphony, the director of a film, the architect of solutions.

This means:

Think in Systems, Not Syntax: Focus on what you want to build, not how to build it. Let Roo handle the implementation details.

Plan Before You Code: Architect Mode exists for a reason. Use it. Your future self will thank you.

Trust but Verify: Roo is incredibly capable, but it's not infallible. Review its work, test thoroughly, and maintain quality standards.

Iterate and Improve: Don't expect perfection on the first try. Work with Roo iteratively, refining and improving as you go.

## The Productivity Revolution

If you implement the practices in this guide, you'll experience a productivity transformation that's hard to believe until you see it:

- Features that used to take days now take hours
- Bugs that used to take hours to find get fixed in minutes
- Documentation actually gets written (miracles do happen)
- Code reviews become learning experiences rather than exercises in frustration
- You actually enjoy coding again (remember that feeling?)

## A Personal Note

I've been writing code for [many years], and I can honestly say that Roo Code has fundamentally changed how I approach software development. It's not just about the speed (though that's incredible). It's about the quality of the experience.

Coding with Roo feels like pair programming with the best developer you've ever worked with – one who never gets tired, never gets frustrated, and always has time to explain things. It's collaborative, educational, and surprisingly fun.

## Your Next Steps

1. Install Roo Code (seriously, stop reading and do it now)
2. Start with a small project (don't try to refactor your entire codebase on day one)
3. Begin with Architect Mode (plan first, code second)
4. Experiment with different models (find your optimal cost/performance balance)
5. Join the community (Discord, Reddit, wherever you feel comfortable)
6. Share your experience (help others discover the magic)

## The Fine Print

This isn't magic. Roo Code is a tool, and like any tool, its effectiveness depends on how you use it. The practices in this guide are battle-tested, but every project is different. Adapt them to your needs.

This isn't a replacement for learning. Roo Code augments your skills, but it doesn't replace the need to understand software development principles. Keep learning, keep growing.

This isn't a silver bullet. Some problems are hard, some codebases are messy, and some bugs are genuinely tricky. Roo Code makes these challenges more manageable, but it doesn't eliminate them entirely.

## A Final Thought

The future of software development is being written right now, and you're part of that story. By learning to work effectively with AI, you're not just improving your own productivity – you're helping to define what it means to be a developer in the age of AI.

Welcome to the revolution. Your AI coding sidekick is waiting.

---

P.S. If you made it through all 50+ pages of this guide, you deserve a medal. Or at least a really good cup of coffee. Roo would buy you one if it had hands and a credit card.

P.P.S. Yes, this entire guide was refined and improved with the help of Roo Code. Meta? Perhaps. Effective? Absolutely.

P.P.P.S. Remember: The best AI coding assistant is the one you actually use. Stop reading documentation and start building something awesome.

---

Last updated: July 2025 Version: 2.1 Contributors: The Roo Code Community Special thanks: Everyone who shared their workflows, reported bugs, and helped make Roo Code better

---

## Appendix A: Quick Reference Cheat Sheet

### Essential Commands & Mentions

Mode Switching: Use the dropdown or type / followed by the mode name (e.g., /architect).

Context Mentions:

- @filename.ext - Specific file
- @folder/ - Entire directory
- @problems - Current errors from the Problems panel
- @terminal - Last terminal output
- @open-tabs - All currently open files

Model Quick Reference

Task	Premium Choice	Budget Choice
Architecture	Gemini 2.5 Pro	DeepSeek R1-0528
Coding	Claude Sonnet 4	DeepSeek R1-0528 / V3
Asking	Gemini 2.5 Flash	Qwen3-235B
Debugging	Claude Opus 4	Devstral Medium
Orchestration	Gemini 2.5 Pro	MiniMax-M1-80K

Emergency Workflows

Roo is stuck in a loop: Click the stop button, clear the conversation (+ icon), and start a new task with more specific instructions.

Context window full: Manually trigger context condensing or clear the conversation and summarize the progress in a new prompt.

Models are down: Switch providers using a different Configuration Profile. OpenRouter is great for this as it has fallbacks. Check the Discord for status updates.

Appendix B: Troubleshooting Guide

Common Issues and Solutions

Problem	Symptoms	Solution
Slow responses	Long wait times, timeouts	Check API provider status, reduce context size, try different model
Repetitive answers	Same response multiple times	Clear conversation, be more specific, lower temperature
Memory issues	Crashes, out of memory errors	Reduce context window, limit concurrent file reads
Wrong context	Irrelevant responses	Use specific @mentions, clear old conversations
Tool failures	Commands fail, browser errors	Check permissions, verify network connection, restart VS Code



## Performance Optimization Checklist

- ☐ Context window appropriate for task size
- ☐ Context condensing enabled at 80-90% threshold
- ☐ Unused files removed from context
- ☐ Browser quality set to 75% or lower
- ☐ Terminal output limited to 500 lines
- ☐ Models matched to task complexity
- ☐ Auto-approve used judiciously
- ☐ .rooignore configured properly

---

“The best code is the code you didn’t have to write. The second-best code is the code Roo wrote for you.” - Anonymous Roo Code User