

The Complete Guide to Autonomous Coding with CORA



CORA

Contents

- 1. Introduction 4**
- 2. Fundamentals 6**
- 3. Getting Started 8**
- 4. Features and Workflows 11**
 - 4.1 Auto Mode 11
 - 4.2 Plan Mode 13
 - 4.3 Ask Mode 16
 - 4.4 Code Mode 18
 - 4.5 Review Mode 20
 - 4.6 Marshal Mode 22
 - 4.7 Skills 25
 - 4.8 Rules 27
 - 4.9 Workflows 29
 - 4.10 MCP Servers 31
- 5. Use Cases 35**
- 6. Tips, Patterns and Troubleshooting 37**
- 7. Resources and Reference 40**

CHAPTER 1

Introduction



1. Introduction

CORA is CodeMate's autonomous agentic coding engine. It runs inside your IDE and has access to your full codebase, project-specific configuration, and any external tools you connect to it. CORA handles everything that happens before typing: understanding real requirements, making architectural decisions, mapping dependencies, and executing the build in the correct order.

Most AI coding tools complete lines or generate snippets based on what is in the current file, with no understanding of the broader system. CORA approaches a prompt the way a senior engineer would. It maps the problem, identifies requirements not yet mentioned, selects a stack with explicit reasoning, produces a dependency-ordered task graph, and builds layer by layer. Every change requires your approval before it is applied.

Who this guide is for

- Teams who want an agent that plans architecture before writing code
- Developers building features that span multiple files and require correct sequencing
- Teams that need disciplined, auditable, quality-controlled implementation
- Anyone encoding team standards, conventions, or recurring processes into an AI agent

What you will learn

- What makes CORA different from code completion and generation tools
- How to install CORA and choose the right mode for a task
- How to use all six modes: Auto, Plan, Ask, Code, Review, and Marshal
- How to configure Skills, Rules, Workflows, and MCP Servers
- Common issues and how to resolve them

CHAPTER 2

Fundamentals



2. Fundamentals

What CORA is

CORA is not a code completion tool. It is an autonomous coding agent that understands what needs to be built before writing a single line, maps dependencies, makes architectural decisions explicitly, and executes in the correct order with your approval at every step.

What CORA does that other tools cannot

CORA is an agentic architect. It takes a high-level prompt, maps requirements nobody mentioned but everyone will eventually need, picks a stack with explicit reasoning, produces a dependency-ordered task graph, and then builds layer by layer, validating each layer before the next. The output is a complete, architecturally sound codebase, not disconnected snippets.

Core concepts

- **Six modes.** Auto, Plan, Ask, Code, Review, and Marshal. Auto orchestrates the others and can switch mid-task; the rest are used deliberately for specific situations.
- **Approval before execution.** Every change CORA makes requires your approval before it is applied, unless Auto-accept has been explicitly enabled.
- **Configuration as plain text.** Skills, Rules, Workflows, MCP server definitions, and system instructions all live on your local filesystem as Markdown or JSON, transparent and version-controllable.
- **MCP and browser control.** CORA can connect to MCP servers such as GitHub, Jira, and Slack, and includes a browser agent capable of navigating pages, interacting with a running app, and testing UI flows from a natural language prompt.

CHAPTER 3

Getting Started



3. Getting Started

Installing CORA

CORA is available across multiple IDEs, including VS Code, JetBrains, Cursor, and Antigravity. Install it from your IDE's extension marketplace by searching for CORA or CodeMate Agent, or through CodeMate Toolbox by navigating to CORA and clicking Install Extension.

Selecting a mode

- **Auto.** Default mode. CORA selects and sequences appropriate modes based on what the prompt requires. Use for most tasks.
- **Plan.** Produces architecture documents and task breakdowns without writing any code. Use when starting something large and wanting to review the approach first.
- **Ask.** Answers questions about the codebase without writing, modifying, or deleting any files. Use for safe, read-only exploration.
- **Code.** Writes, modifies, and refactors code. Use when you have a clear scoped task and want direct execution.
- **Review.** Audits code for correctness, security vulnerabilities, code understanding, code quality, and architectural alignment. Use after any change to a security-sensitive area.

Getting started is fast

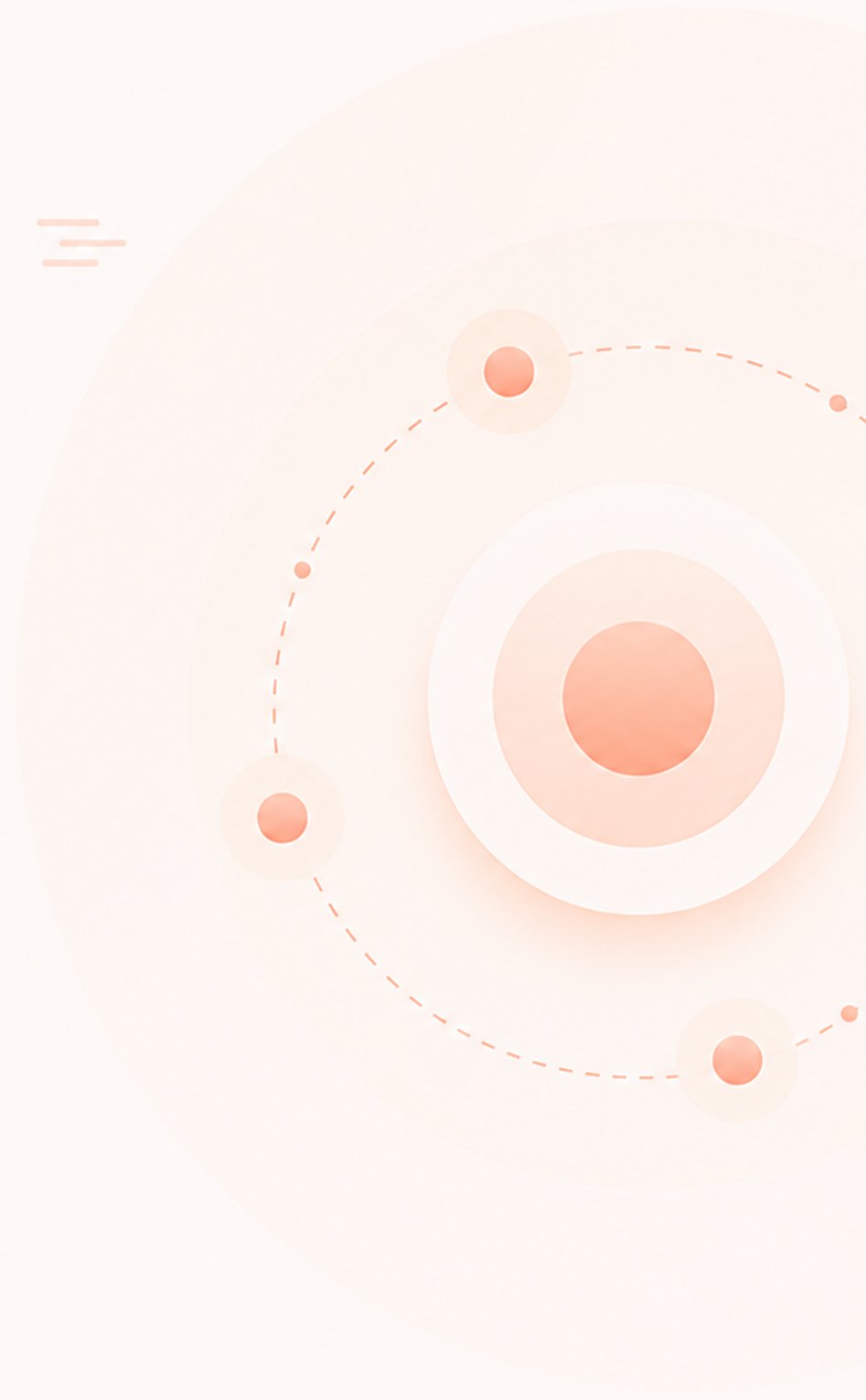
Getting started with CORA requires installing the extension, opening the panel, and selecting the right mode before sending a prompt. CORA makes its reasoning transparent at every step and requires your approval before any changes are written to disk.



The CORA extension in the IDE Marketplace showing its description as an autonomous AI coding agent that plans, architects, and builds production-ready software inside the IDE, with the mode selector dropdown showing Ask, Plan, Code, Review, and Auto

CHAPTER 4

Features & Workflows



4. Features and Workflows

4.1 Auto Mode

Auto mode is CORA's default orchestrator. It analyzes each prompt and determines which combination of modes is required, potentially starting in Plan mode, switching to Ask mode for context, then executing in Code mode, all within a single prompt.

When to use it: Use for most tasks, especially complex ones spanning multiple stages, or when you want CORA to make the mode decisions itself based on the prompt.

How to use it: Select Auto from the mode selector. Type your prompt and press Enter. CORA shows its progress including which modes it is using and what files it is reading.

Example 1

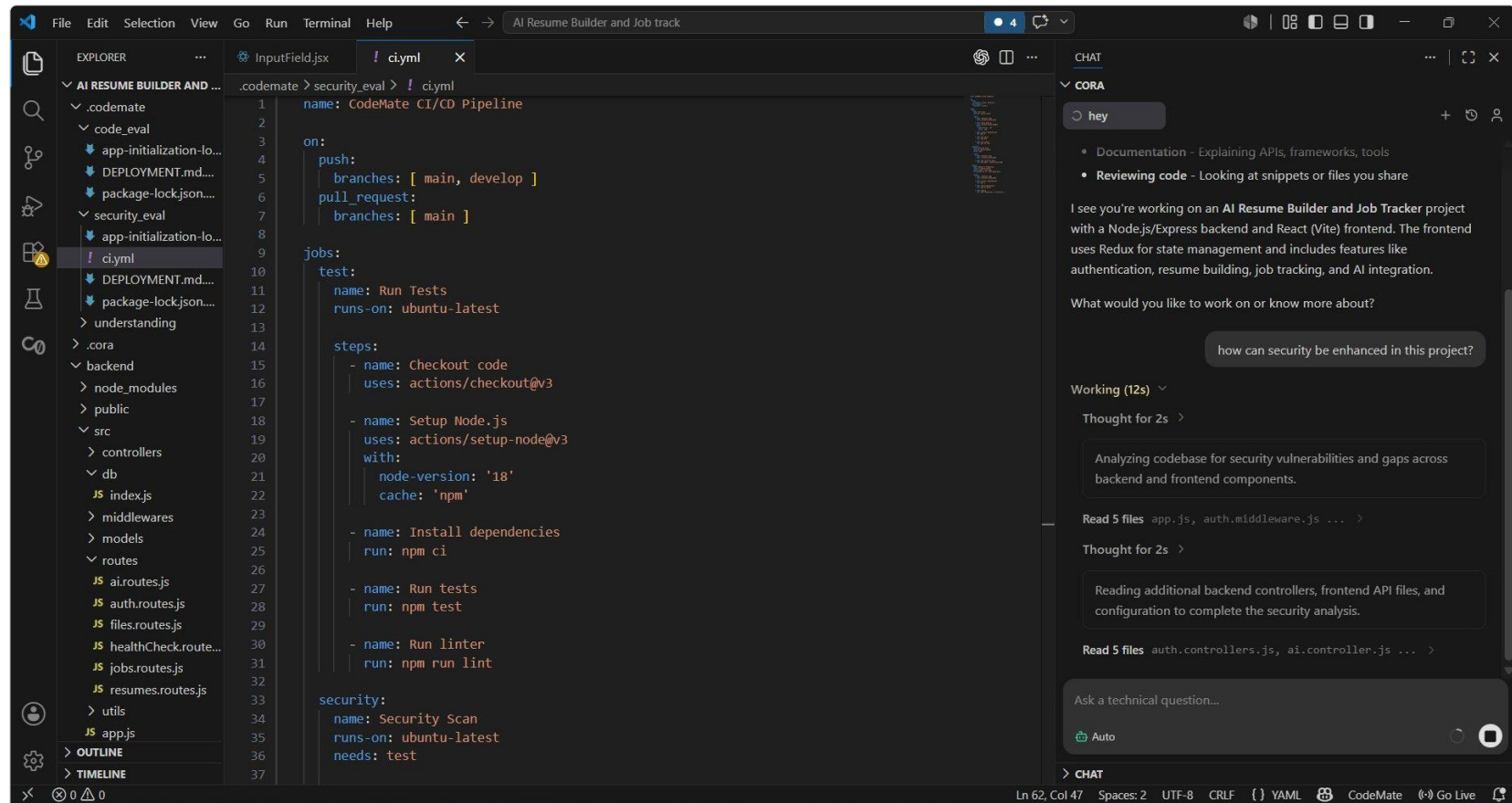
A developer types "How can security be enhanced in this project?" in Auto mode. CORA reads files across the project and produces a comprehensive Critical Vulnerabilities report identifying specific security issues with exact file locations, vulnerable code highlighted, XSS risk explanations, and concrete code fixes for each issue.

Example 2

A developer asks CORA to "add a dark mode toggle to the settings page." CORA starts in Ask mode to locate the theme configuration, switches to Code mode to implement the toggle and persist the preference, then confirms the change compiles and matches existing component patterns.

Example 3

A team asks CORA to "fix the failing checkout tests." CORA reads the test files and the checkout flow, identifies a mismatched API response shape, applies the fix in Code mode, and reruns the tests to confirm they pass.



CORA actively working on a security analysis task showing its transparent reasoning process, including Read 5 files including app.js and auth.middleware.js

4.2 Plan Mode

Plan mode produces architecture documents, stack decisions, and dependency-ordered task breakdowns without writing any code. It asks clarifying questions before producing the plan and saves the output as a Markdown file in the project.

When to use it: Use at the start of any significant feature or project where getting the architecture right before writing code will prevent expensive rework.

How to use it: Select Plan. Describe what you want to build. CORA asks clarifying questions. Answer them and CORA produces the full architecture document. Review and approve the plan before any code is written.

Example 1

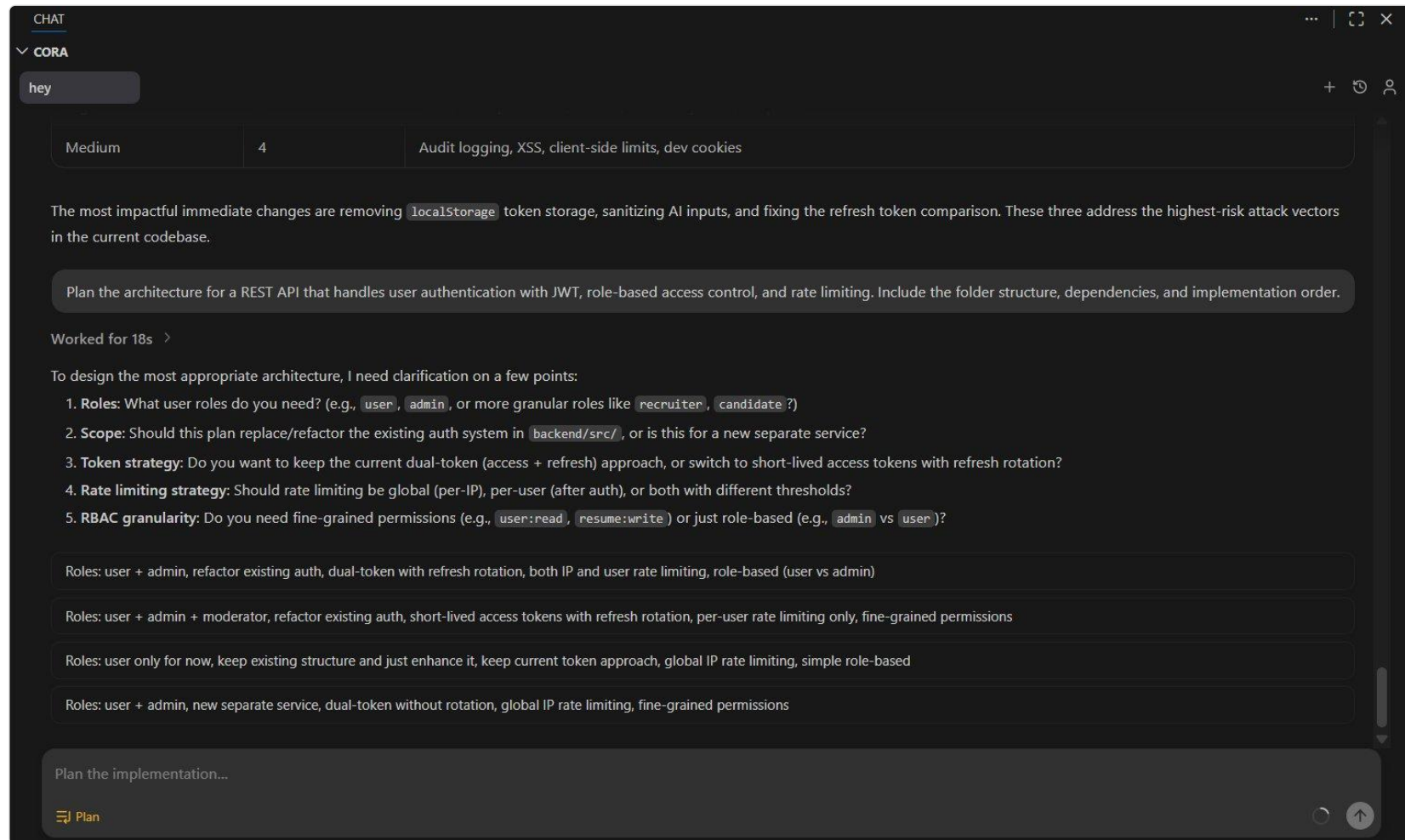
A team building a new authentication system with JWT, RBAC, and rate limiting switches to Plan mode and describes the requirements. CORA asks five clarifying questions and after receiving answers produces a complete architecture document covering every file that needs to be created or modified, saved directly to `plans/rest-api-auth-architecture.md`.

Example 2

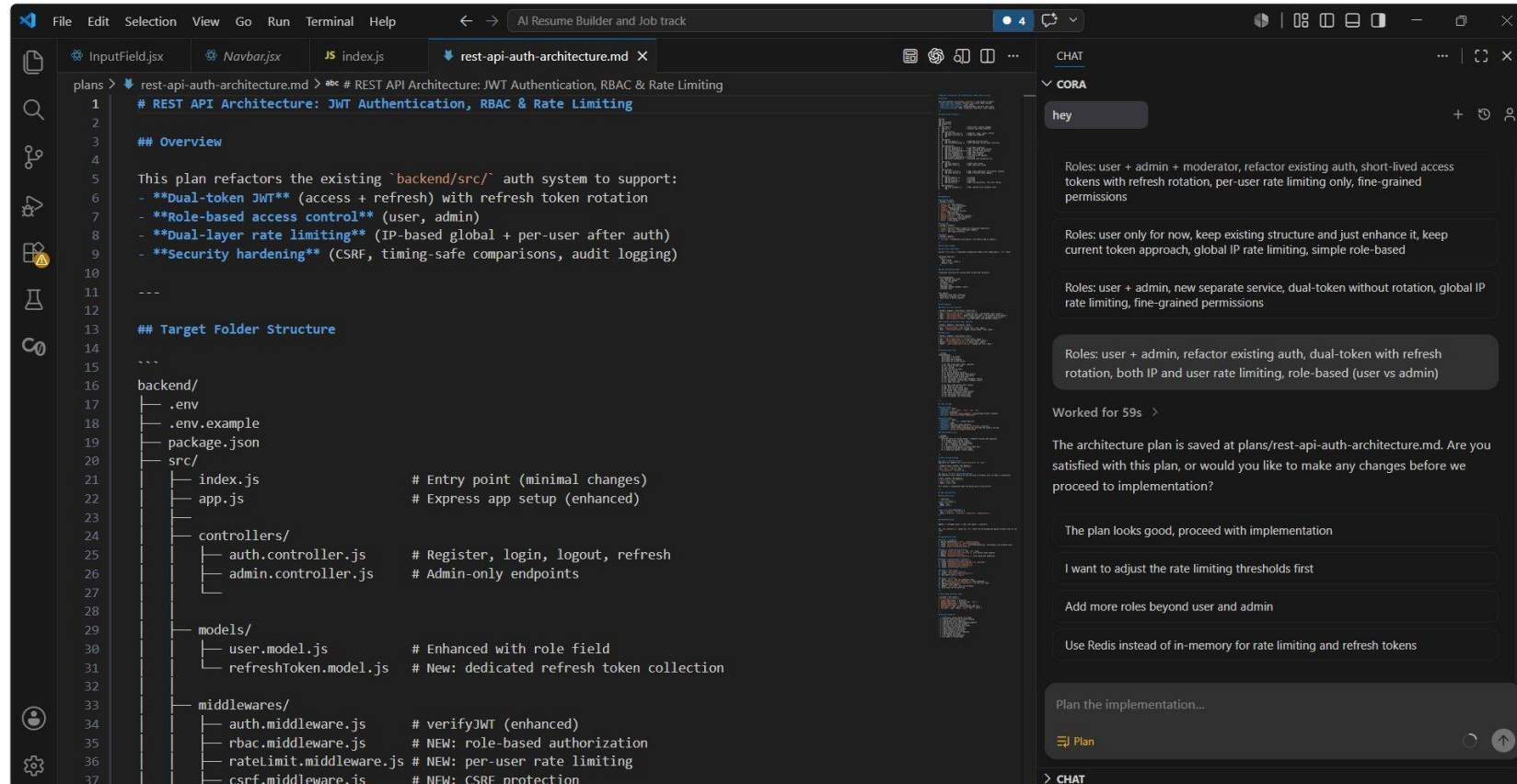
A developer planning a notifications system spanning email, in-app, and push channels switches to Plan mode and describes the requirements. CORA proposes a provider-agnostic notification service, a queue-based delivery pipeline, and a preferences table, saving the plan to `plans/notifications-architecture.md`.

Example 3

A team migrating from REST to GraphQL switches to Plan mode to map the transition. CORA produces a phased migration plan covering schema design, resolver structure, and a compatibility layer for existing clients, saved to `plans/graphql-migration.md`.



CORA Plan mode asking structured clarifying questions before producing the architecture plan, covering roles needed, scope, token strategy, rate limiting approach, and RBAC granularity, with suggested answer options



CORA Plan mode output showing the completed REST API Architecture document with the full annotated folder structure covering dual-token JWT, role-based access control, and dual-layer rate limiting, with CORA asking for approval before proceeding to implementation

4.3 Ask Mode

Ask mode answers questions about the codebase and explains how things work. It reads files and follows call chains to produce accurate answers grounded in actual code. Ask mode never writes, modifies, or deletes any files under any circumstances.

When to use it: Use when you want to understand something about the codebase before making a decision, when onboarding to an unfamiliar project, or when investigating how a specific feature works.

How to use it: Select Ask. Ask any question in plain language. CORA reads relevant files and responds with a specific, grounded answer. Ask mode is completely safe to use freely at any time.

Example 1

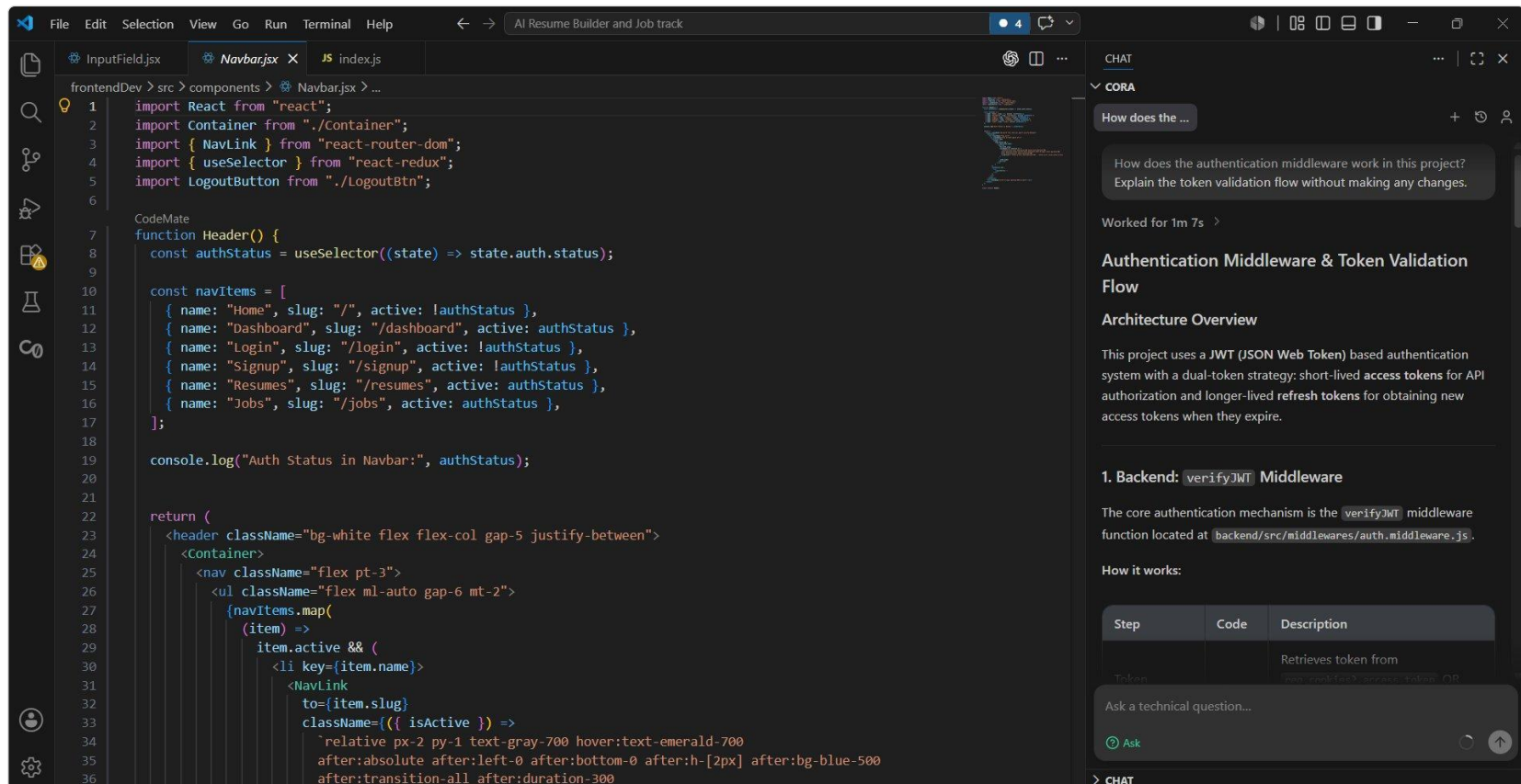
A developer who has just joined a project needs to understand the authentication flow before modifying the middleware. They switch to Ask mode and ask about the token validation flow. CORA reads all relevant files and produces a complete document with an Architecture Overview and a step-by-step table covering every stage with exact file location and line number.

Example 2

A developer notices an API endpoint returning stale data and asks CORA in Ask mode to trace the caching logic. CORA reads the relevant service and cache layer files and explains exactly where the cache invalidation is missing.

Example 3

Before removing a legacy utility function, a developer asks CORA in Ask mode which files depend on it. CORA searches the codebase and lists every file and line that imports the function, so nothing breaks when it is removed.



The screenshot displays a VS Code editor with a file named `Navbar.jsx` open. The code defines a `Header` function that uses `useSelector` to get the authentication status and maps over a list of navigation items to render a `NavLink` component. The chat window on the right shows a query about authentication middleware and a comprehensive response.

```
1 import React from "react";
2 import Container from "../Container";
3 import { NavLink } from "react-router-dom";
4 import { useSelector } from "react-redux";
5 import LogoutButton from "../LogoutBtn";
6
7 CodeMate
8 function Header() {
9   const authStatus = useSelector((state) => state.auth.status);
10
11   const navItems = [
12     { name: "Home", slug: "/", active: authStatus },
13     { name: "Dashboard", slug: "/dashboard", active: authStatus },
14     { name: "Login", slug: "/login", active: authStatus },
15     { name: "Signup", slug: "/signup", active: authStatus },
16     { name: "Resumes", slug: "/resumes", active: authStatus },
17     { name: "Jobs", slug: "/jobs", active: authStatus },
18   ];
19
20   console.log("Auth Status in Navbar:", authStatus);
21
22   return (
23     <header className="bg-white flex flex-col gap-5 justify-between">
24       <Container>
25         <nav className="flex pt-3">
26           <ul className="flex ml-auto gap-6 mt-2">
27             {navItems.map(
28               (item) =>
29                 item.active && (
30                   <li key={item.name}>
31                     <NavLink
32                       to={item.slug}
33                       className={({ isActive }) =>
34                         `relative px-2 py-1 text-gray-700 hover:text-emerald-700
35                           after:absolute after:left-0 after:bottom-0 after:h-[2px] after:bg-blue-500
36                           after:transition-all after:duration-300`
37                     )
38                   )
39             )}
40           </ul>
41         </nav>
42       </Container>
43     </header>
44   );
45 }
```

CHAT

CORA

How does the ...

How does the authentication middleware work in this project? Explain the token validation flow without making any changes.

Worked for 1m 7s >

Authentication Middleware & Token Validation Flow

Architecture Overview

This project uses a **JWT (JSON Web Token)** based authentication system with a dual-token strategy: short-lived **access tokens** for API authorization and longer-lived **refresh tokens** for obtaining new access tokens when they expire.

1. Backend: `verifyJWT` Middleware

The core authentication mechanism is the `verifyJWT` middleware function located at `backend/src/middlewares/auth.middleware.js`.

How it works:

Step	Code	Description
1	<code>Token</code>	Retrieves token from request header
2	<code>Token</code>	Verifies token using <code>verifyJWT</code> middleware

Ask a technical question...

Ask

> CHAT

CORA Ask mode full response showing Authentication Middleware and Token Validation Flow with Architecture Overview explaining the dual-token JWT strategy and a structured How it works table

4.4 Code Mode

Code mode writes, modifies, and refactors code. It reads the existing codebase to understand conventions and patterns, then generates code that fits naturally into the existing architecture. Every change is presented for review before being applied.

When to use it: Use when you have a clear scoped task and want direct execution, after approving a Plan mode output, when implementing a specific component, or when fixing a bug that has been identified.

How to use it: Select Code. Describe what you want to build or change. CORA reads relevant files, generates the code, and presents the changes for review before applying.

Example 1

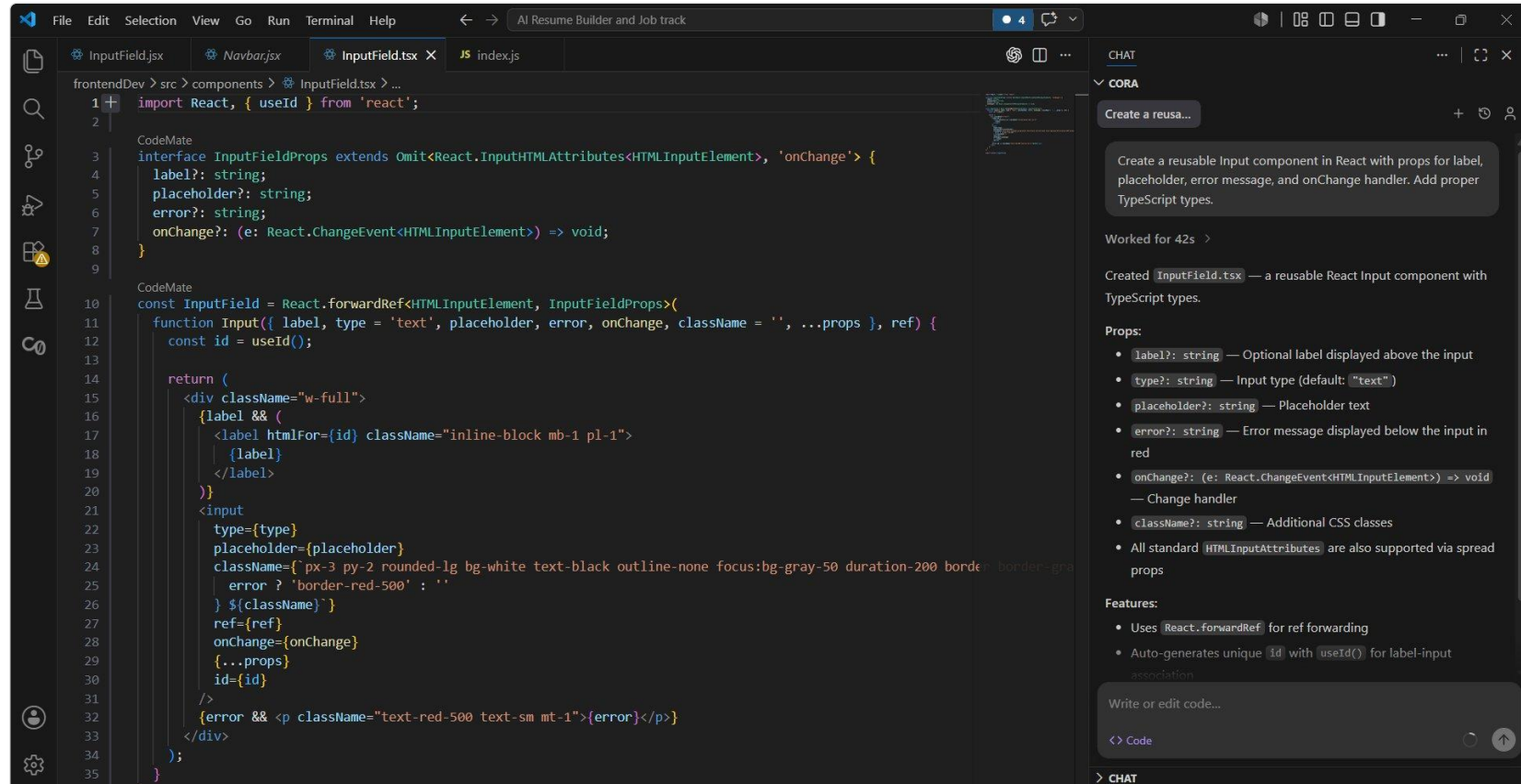
A developer asks CORA in Code mode to create a reusable Input component in React with TypeScript types. CORA reads existing component patterns and creates `InputField.tsx` with a complete TypeScript interface, `forwardRef` support, auto-generated unique id using `useId`, conditional label rendering, error state styling, and spread props support for all standard HTML input attributes.

Example 2

A developer asks CORA in Code mode to add pagination to an existing list endpoint. CORA reads the current route and query patterns, then implements page and limit parameters, adds response metadata, and updates the corresponding tests.

Example 3

A developer asks CORA to refactor a large class component into a functional component with hooks. CORA rewrites the file, preserves existing behavior and prop types, and confirms the component still renders correctly.



CORA Code mode output showing a completed `InputField.tsx` React component created with the full TypeScript implementation visible in the editor, alongside the CORA panel showing Props and Features documentation

4.5 Review Mode

Review mode audits code changes for correctness, consistency, security vulnerabilities, and architectural alignment. It produces a structured review report listing every issue with severity, exact file location, description, and a specific recommendation. Review mode never writes new code.

When to use it: Use before merging any change to a security-sensitive area, as part of a pre-merge checklist, when taking ownership of code written by someone else, or when preparing code for a security audit.

How to use it: Select Review. Describe what you want reviewed. CORA reads the relevant files, analyzes them against best practices and the project's own conventions, and produces a structured review report.

Example 1

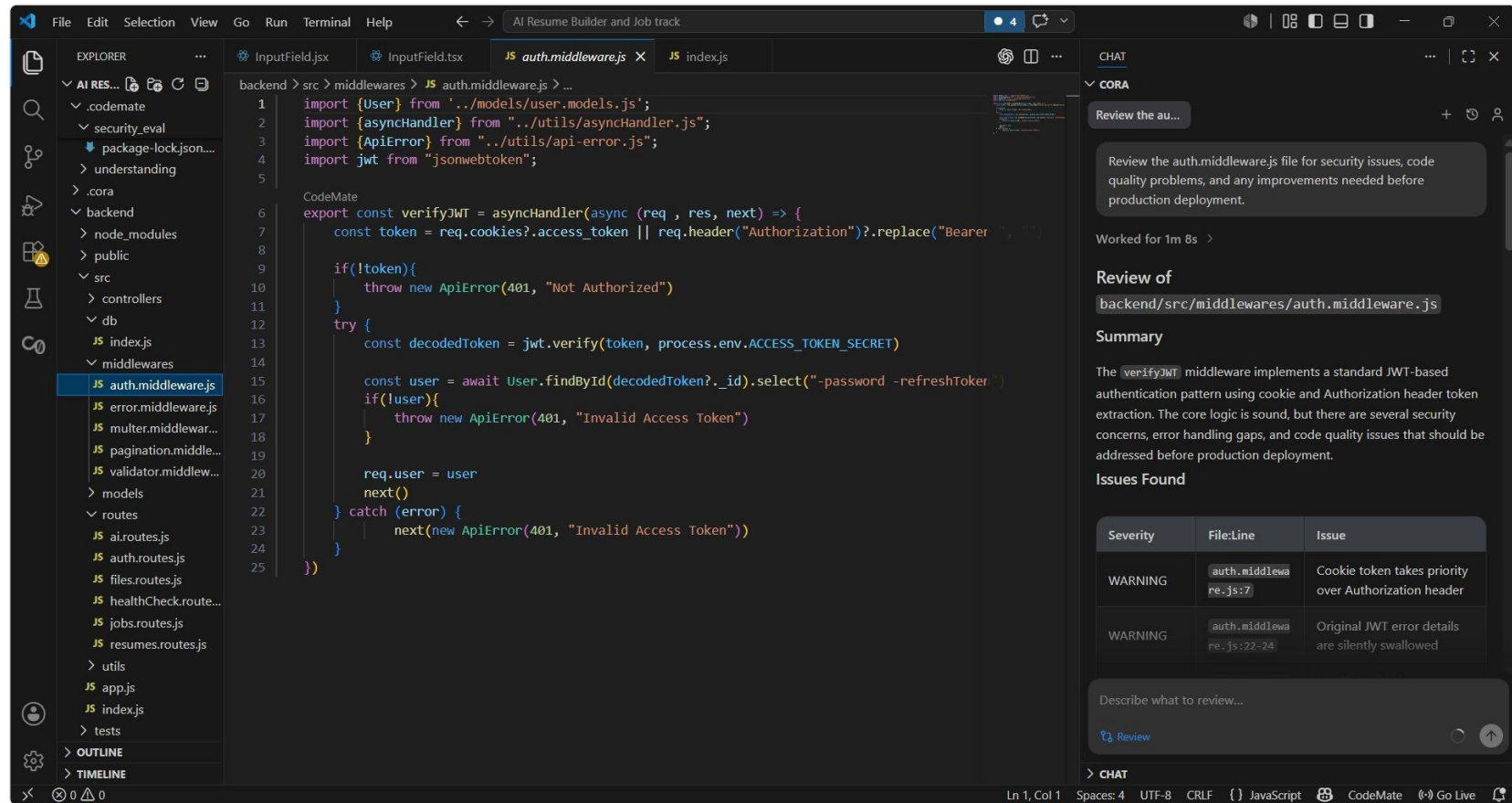
A team runs CORA in Review mode on `auth.middleware.js` before deploying a new authentication flow. CORA produces a review flagging two security concerns: the cookie token taking priority over the Authorization header creating a potential bypass, and original JWT error details being silently swallowed making debugging significantly harder.

Example 2

Before merging a new payments integration, a team runs CORA in Review mode on the checkout controller. CORA flags a missing idempotency key on the payment creation call and an unhandled webhook signature verification failure.

Example 3

A developer runs CORA in Review mode on a database migration script. CORA identifies a missing index on a frequently queried column and a column type mismatch that would silently truncate data.



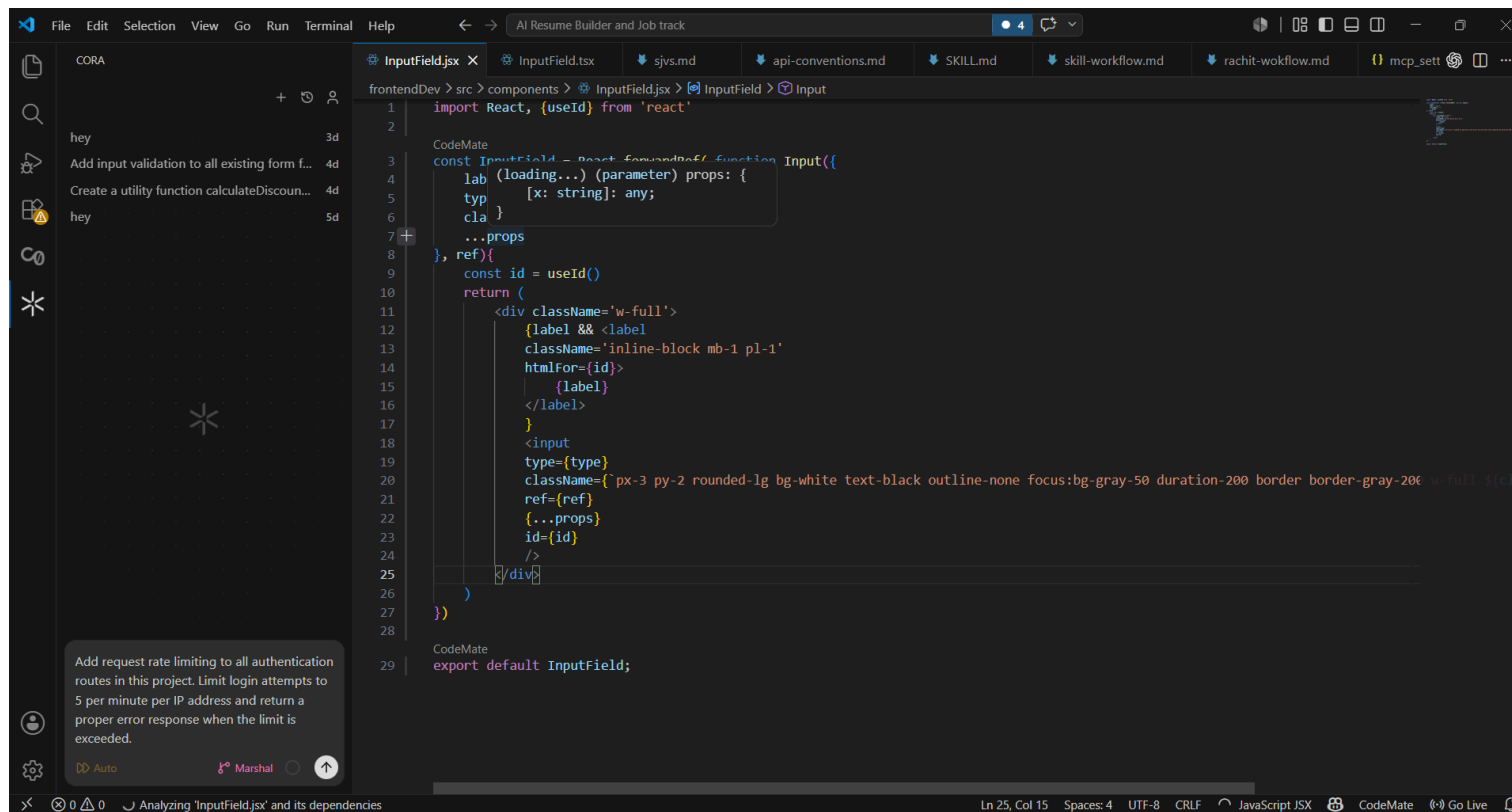
CORA Review mode showing a completed review of `auth.middleware.js` with a Summary noting the core logic is sound but there are security concerns, followed by an Issues Found table listing WARNING severity issues

4.6 Marshal Mode

Marshal mode is CORA's most structured operating mode. While other modes let you jump straight into code edits or conversation, Marshal mode enforces a strict five-stage pipeline before any implementation happens.

The key idea is coordination over execution. Marshal mode does not write source code directly. Instead it researches, plans, reflects, delegates, reviews, and verifies, ensuring every task is approached with discipline and leaves a clear trail of artifacts behind it.

When to use it: Use when the task is complex, when correctness matters more than speed, or when you need the work to be traceable and reviewable after the fact.

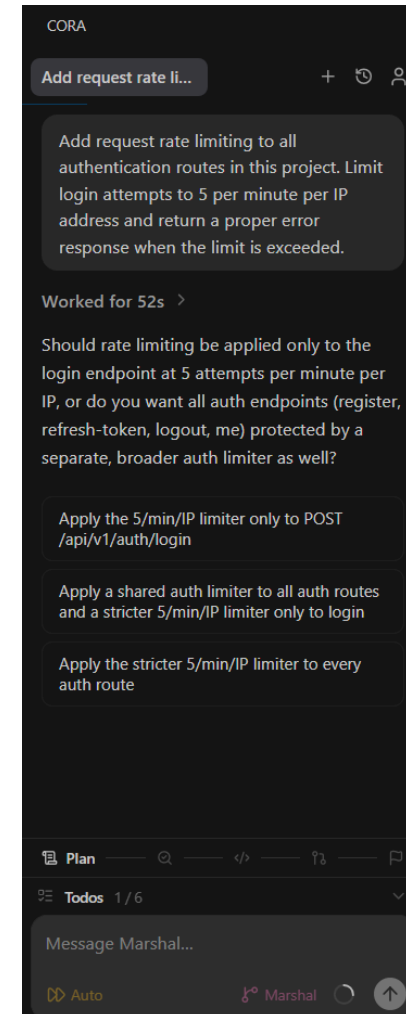


The CORA panel with Marshal mode active, Plan stage running, codebase analysis underway

The five-stage pipeline

Every task run in Marshal mode passes through the same five stages in order, with no skipping and no combining. This is what makes Marshal mode predictable and auditable.

- **1. Plan.** CORA researches the codebase, identifies all the work involved, and writes a concrete implementation plan, saved as a Markdown file inside `.cora/plans/` so it remains visible and reviewable.
- **2. Reflect.** CORA inspects its own plan and checks for risks, edge cases, and gaps. If issues are found the plan is refined before anything proceeds. This stage often completes quickly in the background.
- **3. Implement.** CORA delegates the actual code changes to a code-mode subtask. Marshal mode itself does not edit source files; it hands off to the subtask and monitors its output.
- **4. Review.** A separate review subtask is delegated to validate the implementation against the original plan. This is an independent check, not a self-review by Marshal mode.
- **5. Assessment.** Marshal mode performs a final review of the entire task end to end before marking it complete. This is the last quality gate.



Plan stage: Marshal researched the codebase, then surfaced a clarifying question before writing the plan. Pipeline bar visible at bottom with Plan active, Todos 1/6

Core responsibilities

- **Orchestrate the workflow.** Marshal mode owns the full lifecycle of a task from the first line of the plan through to the final assessment, coordinating all stages and ensuring each completes before the next begins.
- **Enforce stage discipline.** No stage can be skipped. No two stages can be combined. No shortcuts are taken; this rigidity is what makes the output trustworthy and predictable.
- **Coordinate subtasks.** For the Implement and Review stages, Marshal mode spawns dedicated subtasks rather than doing the work itself, ensuring implementation and review are always independent of each other.
- **Keep work traceable.** Every Marshal mode run produces artifacts: the plan file, the review output, and the assessment summary, all of which remain in the project after the task is complete.

When to use Marshal mode

- The task is large or touches multiple files across the codebase
- You need the work to be auditable or traceable after completion
- Correctness is more important than speed
- You want an independent review of the implementation before accepting it
- The task involves architectural decisions, not just small edits

Marshal mode versus other CORA modes

Aspect	Marshal Mode	Other CORA Modes
Planning	Mandatory: written plan before any implementation	Optional or inline
Code editing	Delegated to a code-mode subtask	Done directly
Review	Independent review subtask	Varies by mode

Aspect	Marshal Mode	Other CORA Modes
Speed	Slower: thoroughness is the priority	Faster for simple tasks
Traceability	Full artifact trail left behind	Limited
Best for	Complex, high-stakes, multi-file tasks	Quick edits and exploration

4.7 Skills

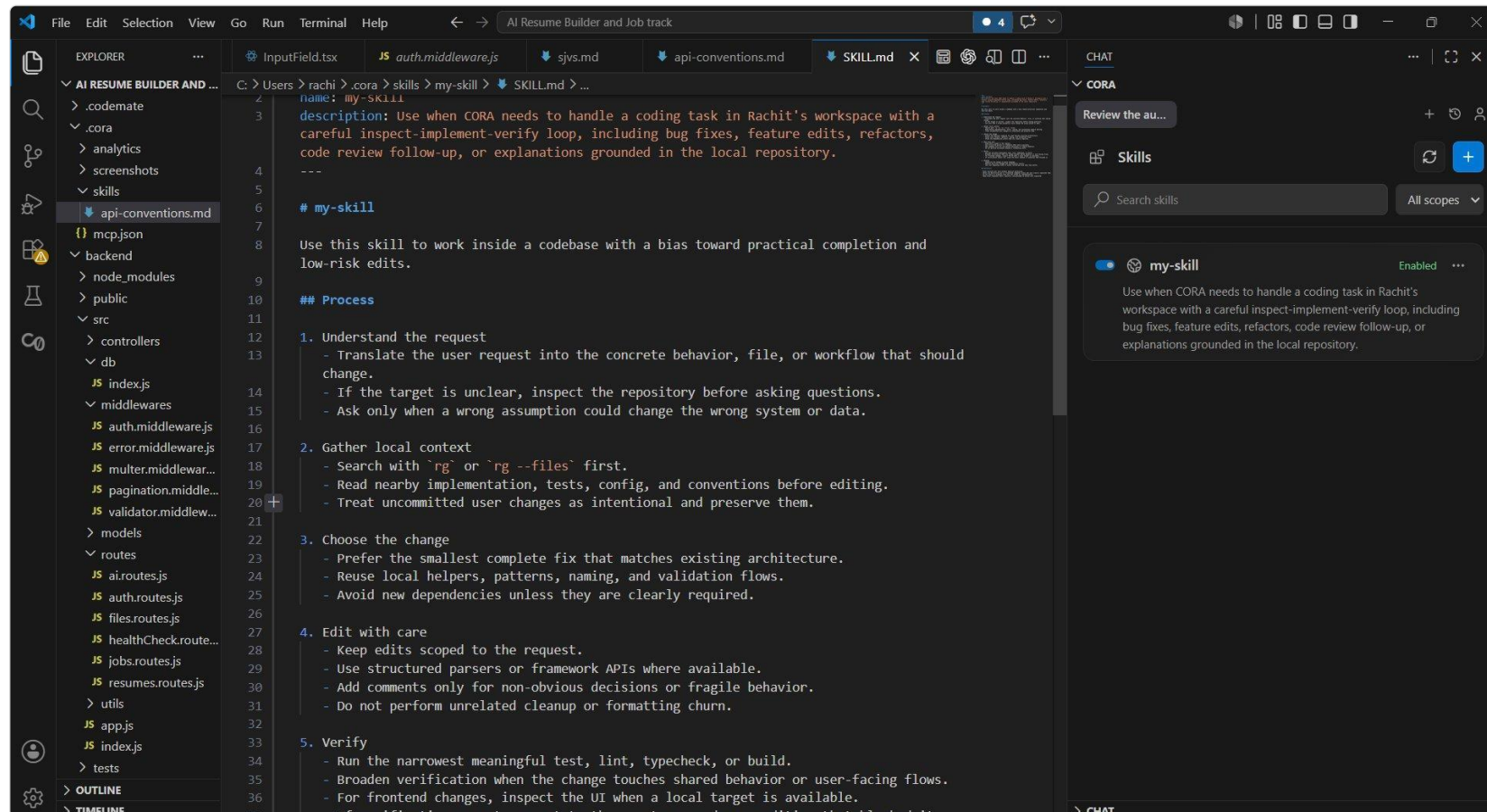
Skills are Markdown files that teach CORA domain knowledge specific to your project, stack, or conventions. A Workspace Skill stored in `.cora/skills/` applies only to the current project. A Global Skill stored in the global settings directory applies to every project you open.

When to use it: Create a skill whenever you find yourself repeating the same context to CORA across multiple sessions. If you always have to explain your API response structure or testing framework, that information belongs in a skill.

How to use it: Open the Skills panel from the CORA utilities menu. Click the plus button and choose between Workspace Skill and Global Skill. Write the skill as a Markdown file with a clear opening description of what it covers. Enable it using the toggle in the Skills panel.

Example

A team creates a Workspace Skill called `api-conventions.md` describing their API response format, authentication header requirements, pagination approach, and file naming conventions. From that point on, every time CORA creates a new route handler it automatically follows the documented conventions without being asked.



The my-skill Workspace Skill file open in the editor showing the full skill content including a 5-step Process covering Understand the request, Gather local context, Choose the change, Edit with care, and Verify, with the Skills panel showing my-skill listed as Enabled

4.8 Rules

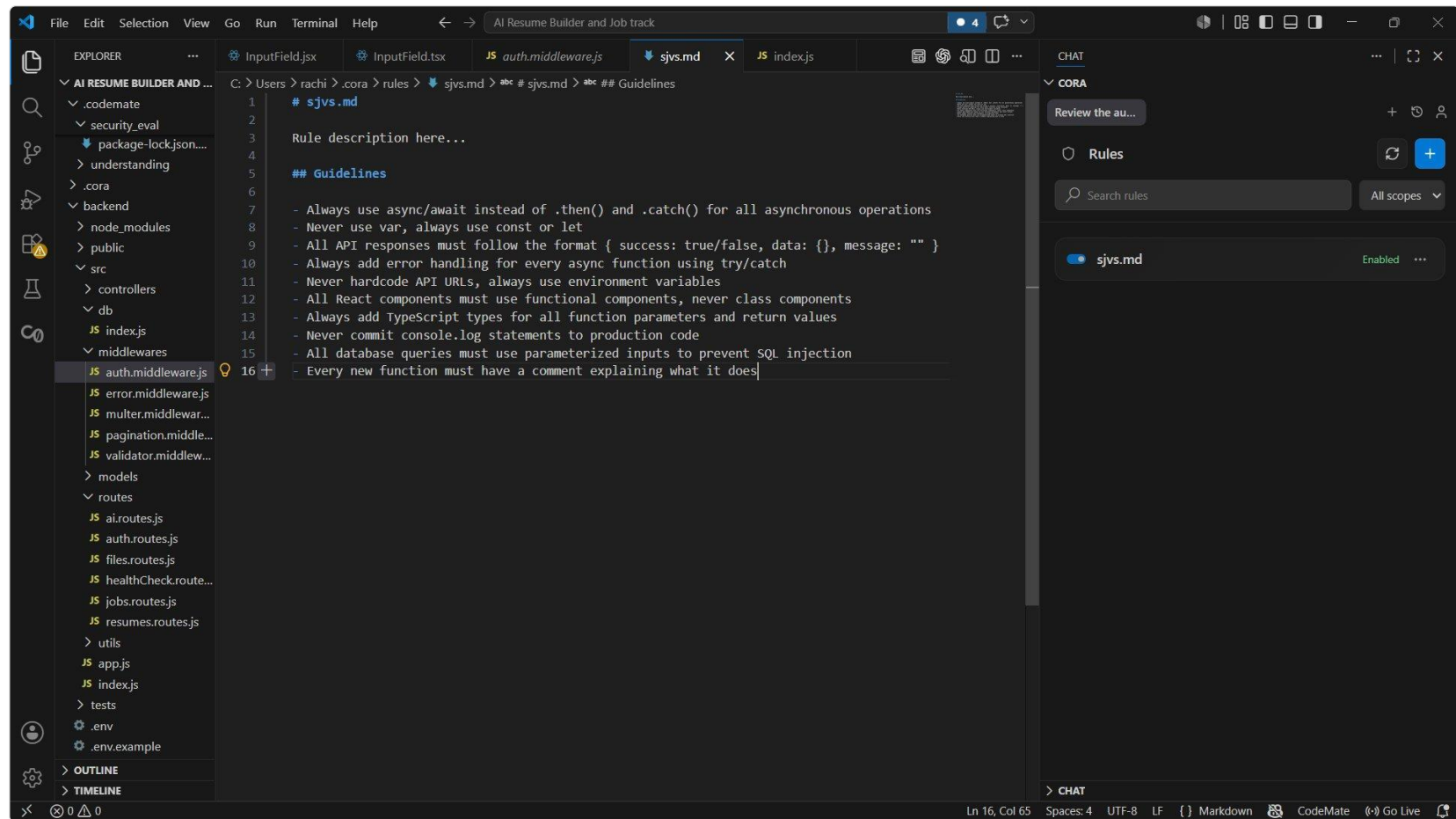
Rules are Markdown files that define hard constraints CORA applies to every single interaction without exception. Unlike Skills, which are loaded selectively, Rules are always active and cannot be overridden by prompt instructions.

When to use it: Create Rules for coding standards and constraints that must be followed universally without exception, such as security requirements, code quality standards, consistency requirements, and safety constraints.

How to use it: Open the Rules panel from the CORA utilities menu. Click the plus button to create a new rule file. Write rules as a Markdown file with clear, unambiguous statements of what CORA must always or never do. Enable the rule using the toggle.

Example

A team creates a Rules file with 10 coding standards. From that point on, every piece of code CORA generates in any mode automatically follows all 10 standards. A developer who generates a new route handler does not need to say "use async/await" or "follow the response format" in their prompt; CORA applies all rules automatically to every line of every file it touches.



A Rules file showing 10 active guidelines including always use async/await, never use var, follow the standard API response format, always add error handling, never hardcode API URLs, use functional React components, always add TypeScript types, never commit console.log statements, use parameterized database queries, and add function comments, with the rules file shown as Enabled

4.9 Workflows

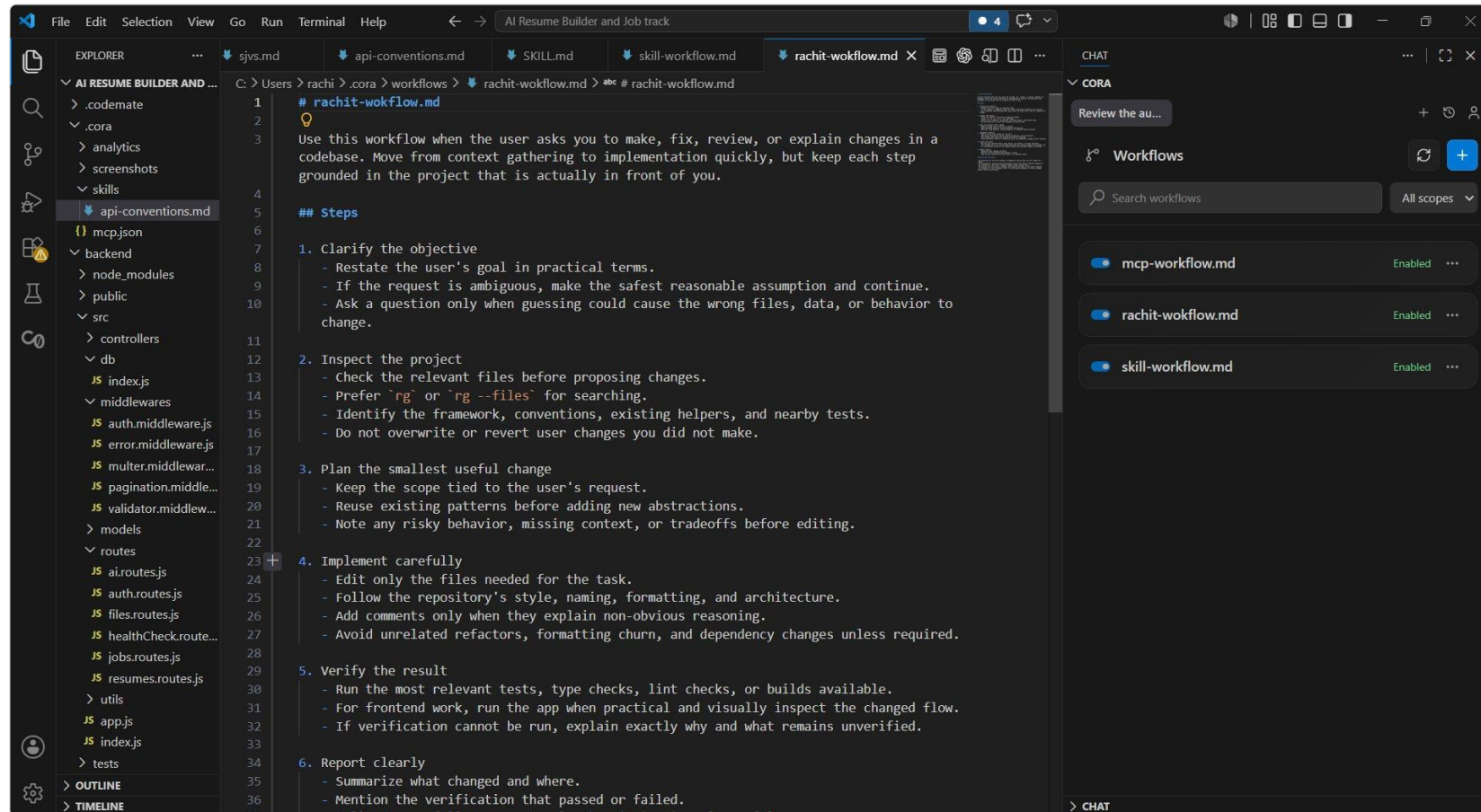
Workflows are Markdown files that define step-by-step processes for recurring tasks. A workflow is a reusable procedure CORA follows exactly when you invoke it by name. Instead of re-explaining a multi-step process every time, you define it once and trigger it with a short prompt.

When to use it: Create workflows for any multi-step process the team performs repeatedly and that needs to be done consistently every time: scaffolding a new feature, preparing a pull request, adding a new database model, or creating a new API endpoint.

How to use it: Open the Workflows panel from the CORA utilities menu. Click the plus button to create a new workflow file. Write the workflow as a Markdown file with numbered steps specific enough for CORA to execute. Invoke a workflow by referencing it by name in a CORA prompt.

Example

A team creates a workflow called `new-feature.md` with 8 ordered steps. When any developer needs to add a new feature, they type "Run the new-feature workflow for a webhook subscription system." CORA reads the workflow and executes all 8 steps in the correct dependency order, asking for approval at each stage before proceeding.



A Workflow file showing a complete 6-step workflow covering Clarify the objective, Inspect the project, Plan the smallest useful change, Implement carefully, Verify the result, and Report clearly, with three workflows all Enabled in the Workflows panel

4.10 MCP Servers

MCP Servers extend CORA's capabilities beyond the local codebase by connecting it to external tools and services through the Model Context Protocol. Once configured, CORA can interact with connected servers in any mode through plain natural language prompts. All processing happens locally, so credentials and data never leave the developer's environment.

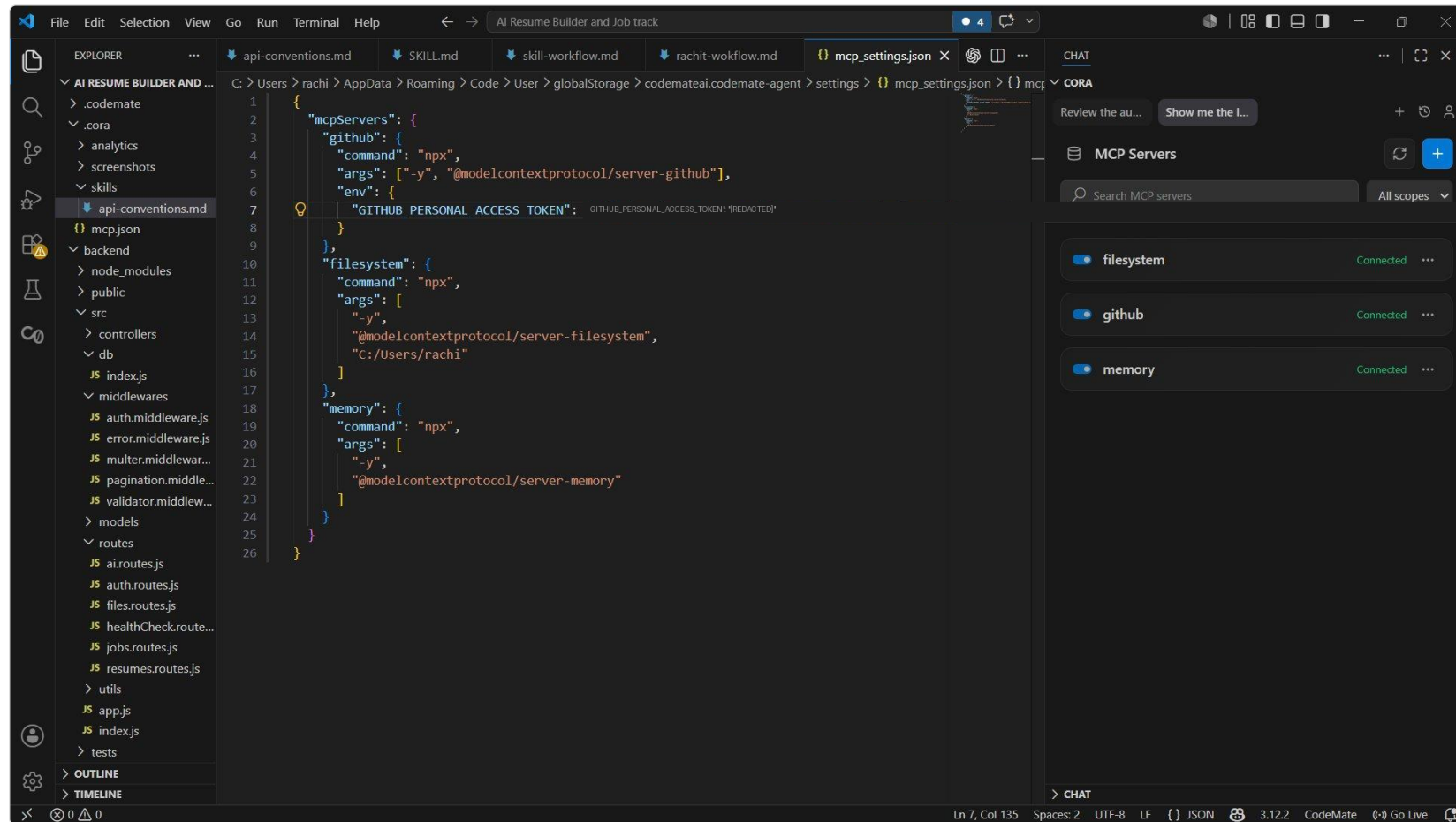
When to use it: Configure MCP Servers when you want CORA to work with systems beyond the local codebase. Enterprise teams commonly connect GitHub for issues and PR history, Jira for ticket context, Slack for notifications, databases for live data querying, and internal APIs for custom integrations.

How to use it: Open the MCP Servers panel from the CORA utilities menu. MCP servers are configured in `mcp_settings.json`. Each server entry specifies the command to launch it, its arguments, and any environment variables such as API keys.

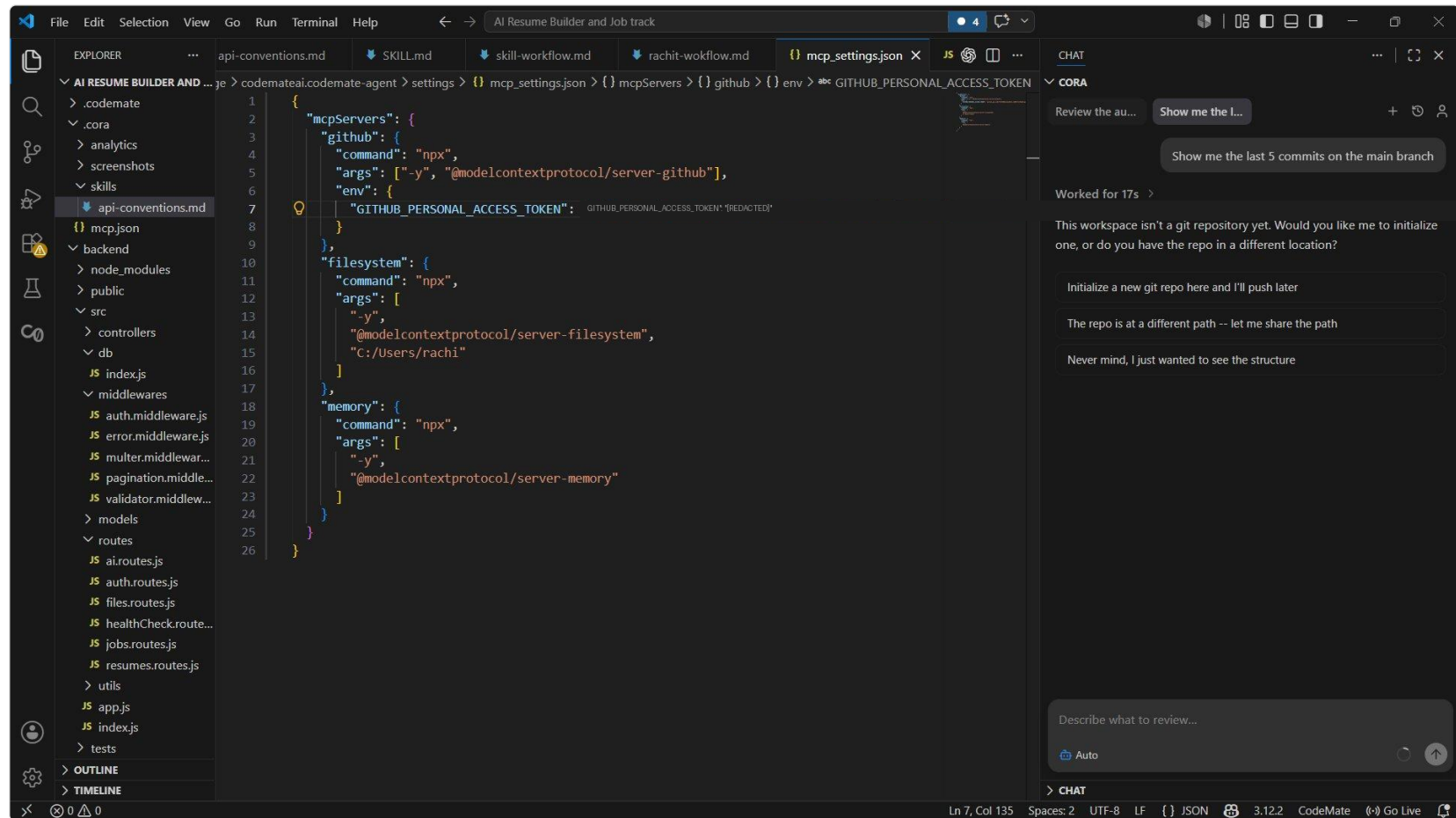
- **GitHub.** Connects CORA to GitHub repositories for reading issues, pull requests, and commit history. Requires a GitHub Personal Access Token with repo scope.
- **Jira.** Connects CORA to Jira Software for pulling tickets into context, creating new tickets, and updating existing ones. Requires a Jira URL, email, and API token.
- **Slack.** Connects CORA to Slack workspaces for posting summaries and sharing review findings. Requires a Slack Bot Token.
- **PostgreSQL.** Connects CORA to a PostgreSQL database for querying live data during debugging and validating query results. Requires a database connection string.
- **Filesystem.** Gives CORA read and write access to files across the entire machine beyond the current project directory.
- **Memory.** Gives CORA a persistent memory store that survives across sessions for storing facts, decisions, and context.

Example

A senior developer asks CORA "Check GitHub for open issues tagged auth and summarize what is blocking the multi-tenant login bug." CORA calls the GitHub MCP server, fetches all open issues with the auth label, reads their descriptions and comments, identifies the blocking dependencies between issues, and produces a structured summary showing which issues need to be resolved first and why.



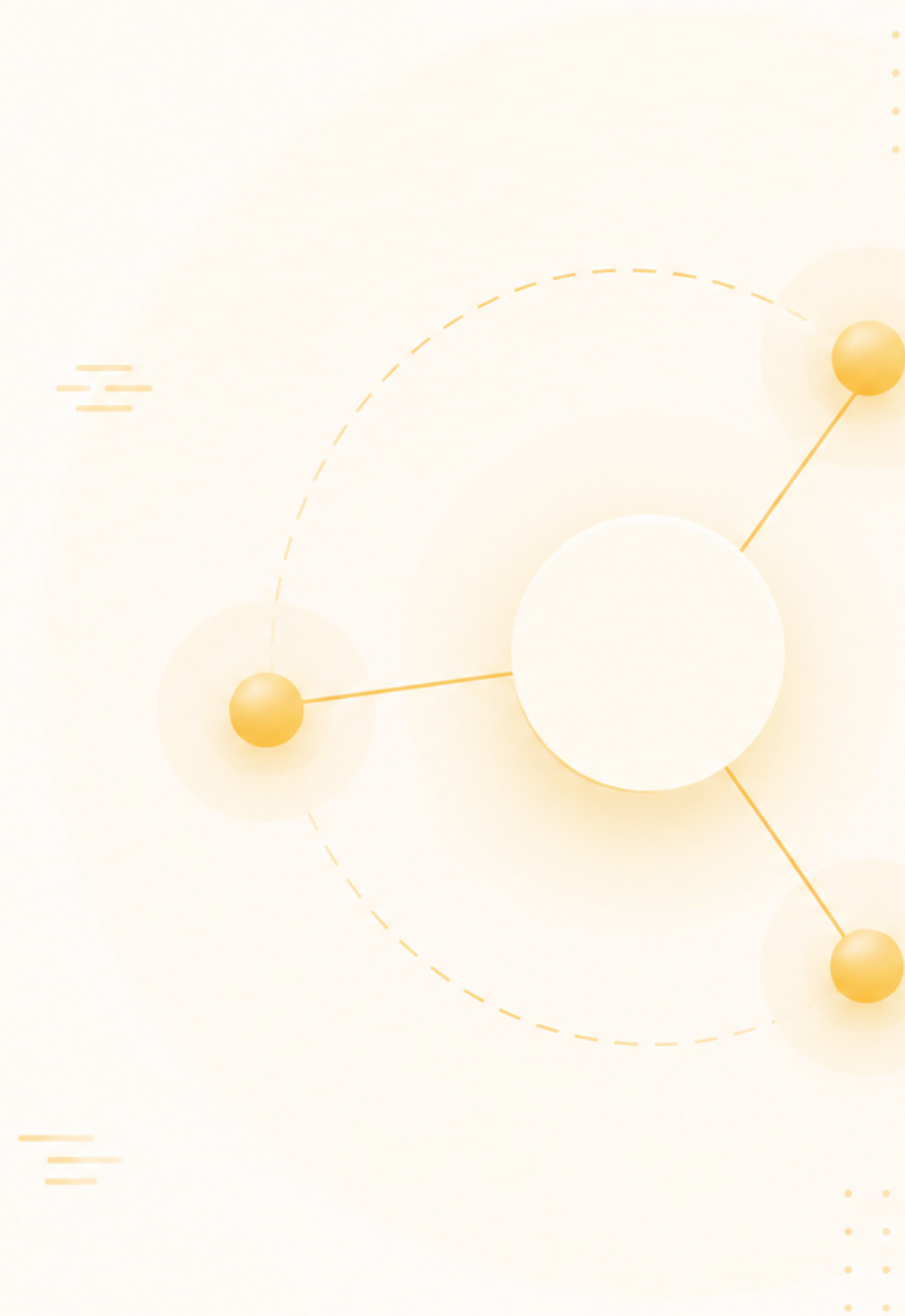
The MCP Servers panel showing three servers all with Connected status: filesystem, github connected using a GitHub Personal Access Token, and memory for persistent cross-session storage, alongside the mcp_settings.json configuration file



CORA actively using the GitHub MCP server connection, responding to a repository query, demonstrating CORA's ability to interact with external systems through MCP and reason about repository context

CHAPTER 5

Use Cases



5. Use Cases

Building new features from a plain language description

Trigger: A developer needs a complete implementation across all files that need to change.

- Describe what to build; CORA asks clarifying questions when the prompt is ambiguous
- Approve the plan, then CORA executes layer by layer

Result: A feature that would take two days to implement manually produced in minutes.

Planning architecture before writing any code

Trigger: A team wants to use CORA as an architectural thinking partner before implementation begins.

- Switch to Plan mode and describe the requirements
- Review the stack decisions, folder structure, and dependency-ordered task graph

Result: A fully annotated architecture plan saved as a Markdown file for review before any code is written.

Understanding an unfamiliar codebase

Trigger: A developer needs to ask questions about a codebase with zero risk of files being changed.

- Switch to Ask mode and ask how a system works, where a feature is implemented, or what a change would break

Result: Grounded answers with no risk to the codebase, since Ask mode never modifies files.

Enforcing coding standards automatically

Trigger: A team wants their coding standards, architectural conventions, and domain knowledge applied consistently without repeating instructions.

- Define the standards once as Rules or Skills
- CORA applies them to every interaction automatically from that point on

Result: Consistent output that matches team standards without re-explaining them in every prompt.

CHAPTER 6

Tips, Patterns, & Troubleshooting



6. Tips, Patterns and Troubleshooting

Tips and patterns

- Auto mode is the right default for most tasks since CORA will pick and sequence the appropriate modes based on your prompt. Reach for a specific mode when you need its particular guarantee, such as Ask mode's read-only safety.
- Use Marshal mode for high-stakes, security-sensitive, or multi-file work where correctness and traceability matter more than speed.
- All CORA configuration lives on your local filesystem as plain text files, making it transparent, version-controllable, and easy to share across a team through your repository.
- Open the relevant files in your editor before sending a prompt, or reference specific files explicitly, to help CORA navigate large codebases more accurately.

Troubleshooting

Issue	Cause	Solution
CORA is not producing the output you expected	Using the wrong mode for the task.	Check you are in the correct mode before sending your prompt. Plan mode produces architecture documents only. Ask mode never modifies files. Code mode is for direct implementation. Review mode audits without writing new code. Marshal mode enforces a strict staged pipeline.
CORA is applying file changes without asking for approval	Auto-accept is enabled.	Go to the CORA configuration panel and verify the Auto-accept setting. Disable it to require explicit confirmation before any file is written to disk.
Skills, Rules, or Workflows are not taking effect	The file is saved in the wrong location, or is not enabled.	Workspace Skills and Rules go in the .cora/ folder of your project; Global files go in your user configuration directory. Check the toggle is enabled in the CORA panel, ensure the file uses valid Markdown with a clear

Issue	Cause	Solution
		opening description, and restart the IDE to reload the configuration fresh.
An MCP Server is not connecting	An incorrect launch configuration or an expired credential.	Verify the server entry in your MCP settings file has the correct launch command, arguments, and environment variables including any required API keys. Check that the key or token is valid and has not expired. Restart the Toolbox subsystem to reinitialize MCP connections, then check the CORA MCP Servers panel confirms the status shows Connected.
CORA is not reading the right files in your project	CORA lacks the context needed to locate the relevant files.	Open the relevant files in your editor before sending the prompt, or reference specific files explicitly in your message. For complex projects, create a Skill file describing your project structure, key conventions, and important file locations.

CHAPTER 7

Resources and Reference

7. Resources and Reference

Where to go next

- Product documentation: docs.codemate.ai
- CodeMate website: codemate.ai
- For issues beyond this guide, see the CodeMate CORA FAQ and Troubleshooting documents.

Quick checklist

Use this before considering your CORA setup complete.

- Installed CORA through Toolbox or the IDE Marketplace
- Comfortable choosing between Auto, Plan, Ask, Code, Review, and Marshal for a given task
- Confirmed Auto-accept is configured the way your team wants it
- Defined at least the Skills and Rules your team repeats most often
- Connected any MCP Servers your workflow depends on, such as GitHub or Jira