Local AI for Apple Developers with Ollama, OpenCode and Coding Models

The Complete Guide

From First Steps to Expert Knowledge

Author: Christian Drapatz

Date: May 2026 · Ollama + OpenCode

Platform: macOS · Apple Silicon

Version 1.0

Disclaimer: *This guide was independently created based on publicly available sources and written in the author's own words. The primary sources were the official Ollama documentation (ollama.com, MIT License), the official OpenCode documentation (opencode.ai/docs, MIT License), as well as personal experience, community contributions, and testing. Technical facts such as API endpoints, configuration options, and commands were taken from these documentations and incorporated as such. All texts were independently structured and written — no verbatim reproduction of copyrighted content has occurred. Code examples are based on the official examples of the respective projects. The content provided is solely for educational purposes; no guarantee of completeness or accuracy is made. All mentioned brands, products, and technologies belong to their respective owners.*

Table of Contents

1 Introduction – What Is This About?

Many developers today work with Claude Code, ChatGPT, Gemini CLI, Cursor, or GitHub Copilot. These tools are powerful but mostly cloud-based. Source code, prompts, and project context frequently leave your own machine.

Ollama takes a different approach: it runs AI models locally on your Mac and makes them available via a local API. Ollama is the local model runner — but not a coding agent. It does not automatically know your project and does not modify files on its own.

OpenCode is the agent layer on top. It runs in the terminal within your project directory and can, depending on configuration and model, read files, execute commands, analyze code, and prepare changes.

In short:

```
Ollama = local AI engine
```

```
OpenCode = coding agent in the terminal
```

The combination looks like this:

```
Apple Silicon Mac
    ↓
Ollama (local model runner)
    ↓
Local Model (e.g. Qwen, Llama, or DeepSeek)
    ↓
OpenCode (terminal agent)
    ↓
Your Xcode / Swift / macOS project
```

1.1 What You Will Learn from This Guide

- Install Ollama and start local models
- Use OpenCode as a coding agent
- Get to know models for Apple development and test them yourself
- Work safely and in a controlled manner with local agents
- Meaningfully combine local AI with cloud tools

This guide is structured so that the beginning is aimed at beginners. The further you read, the deeper the knowledge. You can jump straight to a chapter — the table of contents shows where foundational knowledge ends and expert knowledge begins.

2 Quick Start in 10 Minutes

This section takes you from zero to a running local coding agent on your Mac in under ten minutes. Deeper explanations for each step follow in the subsequent chapters.

Step 1 – Install Ollama (1 minute)

```
brew install ollama
```

Check if it works:

```
ollama --version
```

Step 2 – Load a Model (5–8 minutes, depending on connection)

For the quick start, the 7B coder model — it downloads quickly and is sufficient for initial tests:

```
ollama pull qwen2.5-coder:7b
```

For serious work with OpenCode, a model with reliable Tool-Calling is recommended later, for example:

```
ollama pull qwen2.5:32b
```

More on this model choice in Chapters 8 and 14.

Step 3 – Install OpenCode

```
npm install -g opencode-ai

# or

brew install sst/tap/opencode

# or

curl -fsSL https://opencode.ai/install | bash
```

Step 4 – Create a Git Branch (10 seconds)

Before the agent touches any file:

```
git checkout -b ai/first-test
```

This is not an optional step. AI agents can modify many files simultaneously. A branch allows you to review everything with `git diff` and roll back completely with `git checkout main` if needed.

Step 5 – Start OpenCode and Initialize the Project

```
cd /path/to/your/project

opencode
```

In the OpenCode interface:

```
/init
```

This creates an `AGENTS.md` in your project folder with a summary of the project structure and a sensible starting configuration. You can customize it afterward.

Step 6 – Ask the First Question

Always start by reading, never by changing:

```
Explain the architecture of this project to me. Do not change any files.
```

Then review, and only then allow changes.

What Happens in the Background?

```
OpenCode reads your project files

    ↓

The local model analyzes the code

    ↓

You see in real time which files are being read

    ↓

The response comes locally — no code leaves your Mac
```

Tip: Detailed explanations for each step can be found starting from Chapter 3.

3 What Is Ollama?

Ollama is a local model runner for macOS, Linux, and Windows. You install it on your Mac, download a model, and can start it directly.

3.1 Typical Ollama Commands

The most important commands at a glance:

```
# Download a model

ollama pull qwen2.5:32b


# Start a model (interactive chat)

ollama run qwen2.5:32b


# Show all local models

ollama list
```

```
# Start the server (for API access by other tools)

ollama serve



# Delete a model

ollama rm qwen2.5-coder:7b
```

3.2 The Local API

Ollama provides a local HTTP server:

```
http://localhost:11434
```

This server offers two API styles:

- **Native Ollama API** at `/api/...` — e.g. `/api/generate`, `/api/chat`, `/api/tags`.
- **OpenAI-compatible API** at `/v1/...` — e.g. `/v1/chat/completions`, `/v1/models`.

The OpenAI-compatible variant is now the common denominator for many tools. Chapter 20 explains what's behind it.

Important: The model runs entirely on your machine. With purely local usage, code and prompts remain on your own computer.

3.3 Supported Model Families

Family	Examples	Strength
Qwen	qwen2.5, qwen2.5-coder, qwen3	Code, Reasoning
DeepSeek	deepseek-r1, deepseek-coder	Analysis, Code
Llama	llama3.3, llama3.2	General
Mistral	mistral, mixtral	General
Gemma	gemma3, codegemma	Lightweight, fast
Phi	phi4, phi3.5	Small, efficient

The complete model library is available at ollama.com/library.

4 What Is OpenCode?

OpenCode is a terminal agent for coding tasks. It does not open a graphical interface — it runs directly in the terminal and works within your project folder.

The difference from a normal chat tool:

Normal Chat	OpenCode Agent
gives answers	reads and modifies files
no file access	works directly in the project folder
no shell access	can execute commands (git, swift, xcode…)
session-based	knows project structure and git status

4.1 Provider Freedom

OpenCode supports a large number of providers (75+ according to the documentation). The same interface can be used with Ollama, OpenAI, Anthropic, Gemini, or other providers — simultaneously and switchable at runtime.

4.2 What OpenCode Is Well Suited For

OpenCode is particularly well suited for:

```
Analyzing projects

Explaining architecture

Refactoring code

Writing tests

Finding bugs

Creating documentation
```

For very large, multi-step agent tasks spanning many files, specialized tools like Claude Code are often more comfortable at present. More on this in Chapter 23.

5 Installation Step by Step

5.1 Installing Ollama

With Homebrew — the recommended approach on Mac:

```
brew install ollama
```

Then start the server manually (only necessary if Ollama is not installed as a desktop app):

```
ollama serve
```

Note: If you install Ollama as a macOS desktop app, the server starts automatically in the background. You don't need `ollama serve` — a second launch would fail with `address already in use`.

Check if the server is running:

```
curl http://localhost:11434

# Response: Ollama is running
```

5.2 What Does `ollama serve` Do Exactly?

`ollama serve` starts an HTTP server on port 11434. Tools like OpenCode, Continue, Cline, Open WebUI, or your own apps communicate with this server.

```
Tool (e.g. OpenCode)
```

```
    ↓  HTTP request to localhost:11434
Ollama Server
    ↓  loads model into Unified Memory
Model (e.g. qwen2.5:32b)
    ↓  generates response
Ollama Server
    ↓  HTTP response
Tool
```

When started, the following happens:

```
1. ollama serve runs as a background process
2. Port 11434 is open and ready for requests
3. Other processes on the Mac can now make requests
```

5.3 Loading Models

Where Do Ollama Models Come From?

Ollama models are provided via the Ollama Model Library. There you will find different model families such as Qwen, Llama, DeepSeek, Mistral, Gemma, or Phi.

Models are downloaded via the terminal and then run locally on the Mac. Which models are already installed can be checked at any time with:

```
ollama list
```

Important: A model that works well in Ollama is not automatically suitable for OpenCode. Some models deliver good chat, analysis, or translation results directly in Ollama, but do not support tool calls.

If OpenCode reports for a model:

```
does not support tools
```

that model cannot be meaningfully used in OpenCode for file operations, bash commands, or agentic workflows.

Can Other Models Be Connected?

Yes. Ollama can run many different models, provided they are available in the Ollama Model Library or can be integrated in a suitable format.

For OpenCode, however, it is not enough that a model basically starts in Ollama. What matters is whether the model reliably supports Tool-Calling. OpenCode does not only work with plain text responses — it uses tools like bash, read, grep, glob, edit, or write.

The model must be able to correctly generate and return such tool calls to OpenCode. A model can therefore work fine in Ollama but still be unsuitable for OpenCode.

Therefore, every model should be tested in your own setup, especially when OpenCode is expected to read files, execute commands, or change code.

For productive use with OpenCode, a model with the most reliable Tool-Calling possible is recommended. Regular chat or reasoning models can be helpful for explanations and analysis, but are not always suitable for agentic file and terminal operations.

```
ollama pull qwen2.5:32b
```

For simple code reading, quick tests, or less powerful hardware:

```
ollama pull qwen2.5-coder:14b   # recommended

ollama pull qwen2.5-coder:7b    # compact alternative
```

Models are downloaded once and stored locally — all subsequent requests start immediately.

Storage location on the Mac: Ollama stores all models in the user directory:

```
~/.ollama/models/

├── blobs/      # model data (large files)

└── manifests/  # mapping of names and versions
```

Do not delete models manually. Use the Ollama CLI instead:

```
ollama list       # show installed models

ollama rm <name>  # remove a model
```

5.4 Installing and Starting OpenCode

Installation:

```
npm install -g opencode-ai

# or

brew install sst/tap/opencode

# or

curl -fsSL https://opencode.ai/install | bash
```

Starting in the project directory:

```
cd /path/to/your/project

opencode
```

OpenCode does not automatically detect a local Ollama server as a provider. The configuration from Section 5.6 is required to make this work.

5.5 Which Other Tools Can Use Ollama?

The local server `localhost:11434` is compatible with the OpenAI API structure via `/v1`. The following tools work directly:

Continue (VS Code / JetBrains)

Code completion and chat directly in the editor. Configuration in `~/.continue/config.json`:

```
{
  "models": [
    {
      "title": "Qwen2.5-Coder",
      "provider": "ollama",
      "model": "qwen2.5-coder:14b"
    }
  ]
}
```

Cline (VS Code)

VS Code agent with file and shell access. In the Cline settings:

```
Base URL: http://localhost:11434/v1

API Key:  ollama  (any string)

Model:    qwen2.5-coder:14b
```

Open WebUI

Local web interface that looks like ChatGPT. Installation with Docker:

```
docker run -d \
  -p 3000:8080 \
  --add-host=host.docker.internal:host-gateway \
  -v open-webui:/app/backend/data \
  ghcr.io/open-webui/open-webui:main
```

Available at `http://localhost:3000` afterward. All Ollama models usable, no terminal required.

Your Own Swift App

Access Ollama directly from Swift, no SDK needed:

```
var request = URLRequest(url: URL(string:
"http://localhost:11434/api/generate")!)

request.httpMethod = "POST"

request.setValue("application/json", forHTTPHeaderField: "Content-Type")



let body: [String: Any] = [

    "model": "qwen2.5:32b",

    "prompt": "Explain Swift Actors in two sentences.",

    "stream": false

]

request.httpBody = try? JSONSerialization.data(withJSONObject: body)

let (data, _) = try await URLSession.shared.data(for: request)
```

Note: All integrations communicate with the same local server. Code never leaves your own machine.

5.6 Important OpenCode Configuration Files

OpenCode creates several files during operation that store state, authentication, and configuration. These files are important for diagnostics, customization, and understanding the system.

Overview

File	Location	Purpose	Required?
`opencode.json`	`~/.config/opencode/opencode.json`	Main configuration (provider, model, permissions)	No, but recommended
`model.json`	`~/.local/state/opencode/model.json`	Recently used and preferred models	No, automatic
`auth.json`	`~/.local/share/opencode/auth.json`	Authentication tokens for providers	No, automatic

`~/.config/opencode/opencode.json` – Main Configuration

The most important file. It defines which provider and model are used by default, what permissions OpenCode has, and how tool behavior is controlled.

Does it need to exist? No. Without the file, OpenCode starts with default settings and no local Ollama provider. For Ollama setups, it is mandatory.

Can it be changed? Yes, at any time. Changes take effect at the next OpenCode launch.

Format: JSON. Optional JSON schema for autocomplete available (see below).

When Does the File Need to Be Changed?

The opencode.json does not need to be adjusted on every start. It is only changed when the fundamental configuration of OpenCode should change.

Typical cases:

Situation: New Ollama model should be available → Add entry under provider.ollama.models

Situation: Default model should be changed → Adjust the model field

Situation: Model name in Ollama differs from config → Correct name in models

Situation: Ollama endpoint changes → Adjust baseURL under options

Situation: Adjust tool permissions → Edit the permission section

Situation: Change write access → Set edit / write in permission

Situation: Activate/deactivate subagents or web access → Set task / webfetch in permission

Situation: Use a different provider → Extend provider section accordingly

Example: After ollama pull qwen3:8b, the new model must be added to the config so it is selectable in OpenCode via /models.

After changes to opencode.json, restart OpenCode. Ollama itself usually does not need to be restarted.

Minimal Configuration for Ollama

```
{
  "providers": {
    "ollama": {
      "url": "http://localhost:11434"
    }
  },
  "model": "ollama/qwen2.5-coder:14b"
}
```

Full Example with Multiple Models and Permissions

```
{
  "$schema": "https://opencode.ai/config.json",
  "provider": {
    "ollama": {
      "npm": "@ai-sdk/openai-compatible",
      "options": {
        "baseURL": "http://localhost:11434/v1"
      },
      "models": {
        "qwen3:8b": {
          "name": "Qwen 3 8B"
        },
        "qwen2.5-coder:14b": {
          "name": "Qwen 2.5 Coder 14B"
        },
        "qwen2.5:32b": {
          "name": "Qwen 2.5 32B"
        }
      }
    }
  },
  "model": "ollama/qwen2.5:32b",
  "permission": {
    "bash": "allow",
    "read": "allow",
    "glob": "allow",
```

```
    "grep": "allow",
    "edit": "ask",
    "write": "ask",
    "task": "deny",
    "webfetch": "deny"
  }
}
```

The `permission` section controls what OpenCode may do without asking. Setting `bash`, `read`, `glob`, and `grep` to `allow` makes for smooth operation. Keeping `edit` and `write` at `ask` prevents unwanted file changes.

More information: opencode.ai/docs/de/config/

Configuration Order and Priority

OpenCode reads configurations in the following order (higher priority overrides lower):

```
1. Remote   → Organization standards via .well-known/opencode
2. Global   → ~/.config/opencode/opencode.json
3. Project  → opencode.json in the project folder
4. Environment → OPENCODE_CONFIG environment variable
```

Configurations are merged, not replaced. Project-specific settings only override the fields you specify.

The Most Important Configuration Fields

```
{
  "model": "ollama/qwen2.5:32b",
  "theme": "dark",
  "autoupdate": true,
  "permission": {
    "*": "ask"
  },
  "provider": {
    "ollama": {
      "npm": "@ai-sdk/openai-compatible",
```

```
      "options": {
        "baseURL": "http://localhost:11434/v1"
      }
    }
  }
}
```

Field	Meaning
`model`	Default model in `provider_id/model_id` format
`theme`	Color scheme of the TUI interface
`autoupdate`	Enable/disable automatic updates
`permission`	Permission rules (see Chapter 16)
`provider`	Provider URLs and options
`system`	Global system prompt for all sessions
`mcp`	MCP server configuration (see Chapter 21)

Variable Substitution

In the configuration, you can reference environment variables and files:

```
{
  "provider": {
    "openai": {
      "apiKey": "{env:OPENAI_API_KEY}"
    }
  },
  "system": "{file:./prompts/system-prompt.md}"
}
```

Useful when API keys should not be stored in plain text in the configuration.

Overriding the Model via Command Line

The model can also be passed directly at startup:

```
opencode --model ollama/qwen2.5:32b

# or short:

opencode -m ollama/qwen2.5:32b
```

Editor Support

The schema for autocomplete and validation is available at:

```
https://opencode.ai/config.json
```

Include it in VS Code in the configuration file:

```
{
  "$schema": "https://opencode.ai/config.json"
}
```

`~/.local/state/opencode/model.json` – Model History

Created and updated automatically by OpenCode. Stores which models were most recently used (`recent`) and which are marked as favorites (`favorite`).

Does it need to exist? No. OpenCode creates it automatically on the first model switch.

Can it be changed? Yes. You can e.g. shorten the `recent` list or adjust the order. The file is overwritten on the next model switch.

Format: JSON.

```
{
  "recent": [
    { "providerID": "ollama", "modelID": "qwen2.5:32b" },
    { "providerID": "ollama", "modelID": "qwen2.5-coder:14b" }
  ],
  "favorite": [],
  "variant": {
    "ollama/qwen2.5:32b": "default"
  }
}
```

`~/.local/share/opencode/auth.json` – Authentication

Stores API keys and authentication tokens for configured providers. For Ollama, it contains a symbolic key (`"ollama"`), since Ollama does not require real authentication locally.

Does it need to exist? No. OpenCode creates it automatically when a provider is connected.

Can it be changed? Yes, but with caution — incorrect entries can cause providers to not be recognized.

Format: JSON.

```
{
  "ollama": {
    "type": "api",
    "key": "ollama"
  }
}
```

For cloud providers like Anthropic or OpenAI, the actual API keys are stored here. This file should **not be included in the git repository**.

More information: opencode.ai/docs/de/providers/

6 First Launch – What to Expect

6.1 The TUI Interface

OpenCode shows a **TUI** (Terminal User Interface) — an interactive interface directly in the terminal. No window, no app, but a structured view with an input field, conversation history, and status bar.

On first launch you typically see:

```
┌─ OpenCode ──────────────────────────────────────────┐
│  Provider: ollama · Model: qwen2.5:32b              │
│  Working directory: /Users/.../MyProject            │
│                                                     │
│  > _                                                │
└─────────────────────────────────────────────────────┘
```

You can see which provider is active, which model is loaded, and which folder you are working in.

6.2 What OpenCode Does at Startup

OpenCode analyzes the structure of your project folder at startup. It therefore already knows some files before you ask the first question.

When OpenCode executes a task, you see the actions in real time:

```
Reading:  Sources/App/SearchService.swift
Reading:  Sources/App/NetworkLayer.swift
Running:  grep -r "SearchServiceImpl" .
```

```
Writing:  Sources/App/SearchService.swift
```

This is the important difference from normal chat: you see what the agent is doing, not just the finished answer.

6.3 Starting Safely: Git Is Mandatory

Before OpenCode is allowed to change files, always create a branch:

```
git checkout -b ai/opencode-test
```

Basic git knowledge is a prerequisite. You should understand what `git status`, `git diff`, `git checkout`, and `git reset` do, otherwise you will quickly lose track.

For those who prefer a graphical interface:

Client	Platform	Highlight
Fork	Mac and Windows	fast, clear, free to use
GitHub Desktop	Mac and Windows	simple, directly connected to GitHub
Tower	Mac and Windows	professional, many features, paid
SourceTree	Mac and Windows	free, good for Atlassian users

6.4 Recommended Workflow for the First Session

```
1. git checkout -b ai/test

2. opencode

3. /init  (let the project be analyzed)

4. "Explain this project to me. Do not change any files."

5. git diff  (check after every change)
```

7 Why Is This Interesting for Apple Developers?

For Swift and Xcode development there are clear use cases:

```
Explaining code

Preparing refactoring

Analyzing SwiftUI views

Writing unit tests

Reviewing architecture

Finding bugs

Improving CLI scripts

Writing documentation

Checking Localizable files
```

```
Explaining commit diffs
```

7.1 Particularly Interesting for Companies and Client Projects

For company projects, health apps, internal tools, or client code, local AI is especially interesting. You can have code analyzed and edited without sending it to a cloud service.

Note: This does not replace Claude Code or Codex in every scenario. For many daily tasks, however, it goes surprisingly far.

7.2 Apple Silicon Is Well Suited

Apple Silicon uses a unified memory architecture (Unified Memory). CPU, GPU, and Neural Engine share the same RAM. This makes local models on the Mac more efficient in practice than on comparable PC hardware with separate GPU VRAM.

8 Local Models – Overview and Practical Experience

> *The following models are meant as orientation. Which model works best in your own setup must be tested by each user individually. Actual quality depends strongly on hardware, model size, quantization, project scope, prompt style, and the model's Tool-Calling support. Since the reader's hardware is not known, only practical experience and hints can be given here.*

In the following sections I describe three models that have proven useful in my own setup. The statements are explicitly assessments from my own tests, not general guarantees. Results may differ on other hardware or with different versions of Ollama and OpenCode.

8.1 Qwen 2.5 32B – Strong at Tool-Calling in Personal Experience

`qwen2.5:32b` is the general Qwen 2.5 model in the 32B size. In my own tests with OpenCode and Ollama, it proved to be the most reliable model for **Tool-Calling** — i.e., when the model actually executes commands, reads files, and makes changes rather than just outputting JSON text.

```
ollama pull qwen2.5:32b
```

Possible tasks that worked well in personal tests:

```
Project analysis (model reads and understands files independently)

Bug hunting across multiple files

Understanding and explaining Swift code

Creating architecture overviews

Executing commands (git, find, grep)
```

Assessment:

Property	Value

Size	32B (~19 GB on disk)
RAM requirement	recommended from 32 GB (better 48 GB+)
Tool-Calling	stable and correct in personal tests
Speed	medium to slower (depending on hardware)
Strength	Coding, analysis, reasoning

> **Note on coder variants:** *In my tests, the coder variants (*`qwen2.5-coder:7b`*,* `qwen2.5-coder:14b`*) had problems with Tool-Calling via Ollama in OpenCode. OpenCode partially received the tool call output as plain text rather than as a structured* `tool_calls` *API call. The model described commands instead of executing them.* `qwen2.5:32b` *(without* `-coder`*) did not show this behavior in my tests. This observation does not necessarily apply universally — see also Section 8.2.*

8.2 Qwen2.5-Coder – Strong for Coding, but Not Reliable at Tool-Calling in Personal Tests

Qwen2.5-Coder is one of the strong open coding models for local use. It comes in six sizes: `0.5b`, `1.5b`, `3b`, `7b`, `14b`, and `32b`.

```
ollama pull qwen2.5-coder:14b
```

In multiple personal tests:

- **Code responses:** Qwen2.5-Coder can deliver very good code responses. For pure code reading, explaining, and suggestions without agent actions, it is a sensible choice.
- **Tool-Calling via Ollama in OpenCode:** does **not work reliably** in personal tests. Tool calls were sometimes only output as JSON or not executed correctly. This is a personal observation — on other hardware, with different modelfile variants or newer Ollama/OpenCode versions, results may differ.

Possible tasks for the coder variants — without agent actions:

```
Explaining Swift code

Analyzing SwiftUI views

Code suggestions for manual adoption

Designing DTOs and services

Describing code errors
```

Assessment by size:

Size	Speed	Note
7B	fast	good for small, clear tasks
14B	medium	usable starting point for code tasks
32B	slower	stronger, needs more RAM

Conclusion: Those who want to use OpenCode + Ollama with real Tool-Calling are currently — from personal experience — better served with `qwen2.5:32b` than with the coder variants. Those who want to use Qwen2.5-Coder as a pure code model without agent actions can do so in parallel.

9 Hardware Recommendations for Apple Silicon

RAM is the most important resource for local models on the Mac. The following values are rough guidelines — actual usability also depends on model size, quantization, and apps running in parallel.

Mac RAM	Suitable Models
8 GB	small models (3B/7B), only with limitations
16 GB	Qwen2.5-Coder 7B, smaller DeepSeek variants
24 GB	Qwen2.5-Coder 14B usably runnable
32 GB	Qwen2.5-Coder 14B, DeepSeek R1 14B well usable
48–64 GB	Qwen 2.5 32B, Qwen2.5-Coder 32B, DeepSeek R1 32B
96–128 GB	large 70B models possible, but slower

9.1 Recommendation for Most Apple Developers

With 32 GB RAM, a good starting point:

```
ollama pull qwen2.5-coder:14b
```

With Mac Studio or MacBook Pro (48 GB+):

```
ollama pull qwen2.5:32b
```

Tip: Anyone who wants to work seriously with local models should plan for at least 32 GB RAM when buying their next Mac.

10 Keyboard Shortcuts in OpenCode

OpenCode uses a **leader key system**: most actions are triggered with a key sequence, not a single key. This avoids conflicts with the terminal.

10.1 The Leader Key

`Default: ctrl+x`

Example: Create new session = `ctrl+x` → then `n`

10.2 The Most Important Keyboard Shortcuts

Key / Combination	What Happens
`Tab`	Activate / switch Plan Mode
`Shift+Tab`	Switch agent
`ctrl+x n`	Start new session
`ctrl+x l`	Show session list
`Page Up / Page Down`	Scroll through conversation history
`ctrl+a`	Go to beginning of input line
`ctrl+e`	Go to end of input line
`ctrl+c`	Cancel current action

10.3 Plan Mode vs. Build Mode

OpenCode has two modes:

```
Plan Mode  (Tab)   → agent only creates a plan, does not change files
```

```
Build Mode          → agent executes changes
```

Recommendation: Start in Plan Mode, read the plan, then switch to Build Mode. This gives you control over what happens next.

10.4 Customizing Shortcuts

Keyboard shortcuts can be customized in `~/.config/opencode/tui.json`. Individual shortcuts can be disabled:

```
{
  "keybinds": {
    "session:new": "none"
  }
}
```

11 Project Rules and Instruction Files

AI agents can be very powerful — without clear guidelines, however, they sometimes do things you don't want. Project rules are the solution: files that tell the agent at startup how to behave in your project.

11.1 What Is CLAUDE.md – and What Does It Have to Do with OpenCode?

`CLAUDE.md` is an instruction file from the ecosystem of **Claude Code** (Anthropic's AI coding agent). The idea: you place a text file in the project folder, and the agent reads it at startup as a rule set.

OpenCode has adopted and extended this concept. In OpenCode, the primary file is called **`AGENTS.md`**.

Tool variants:

Filename	Tool
`AGENTS.md`	OpenCode (primary file)
`CLAUDE.md`	Claude Code (Anthropic)
`RULES.md`	general convention

Note: OpenCode also reads `CLAUDE.md` and the global Claude fallback at `~/.claude/CLAUDE.md`. Those familiar with Claude Code can use the same file for both tools.

11.2 Where Are These Files – Global vs. Project-Specific Rules

There are two levels:

```
~/                                    ← Home directory
├── .claude/
```

```
│   └── CLAUDE.md                          ← Claude Code fallback (also read by
OpenCode)
├── .config/
│   └── opencode/
│       ├── opencode.json                  ← global OpenCode configuration
│       └── AGENTS.md                      ← global rules for all projects
└── GIT-Home/
    └── MyProject/
        ├── AGENTS.md                      ← project-specific rules
        ├── opencode.json                  ← project-specific configuration
        └── Sources/
```

11.3 In What Order Does OpenCode Search for Rule Files?

OpenCode searches at startup in the following order:

```
1. Current directory → parent directory → further up  (AGENTS.md, CLAUDE.md)
2. Global file: ~/.config/opencode/AGENTS.md
3. Claude Code fallback: ~/.claude/CLAUDE.md
```

OpenCode thus searches not only in the start directory, but also in parent folders. If you place `AGENTS.md` in a parent folder, it applies to all projects beneath it.

Important: The first file found per level wins — files are not automatically merged.

11.4 Automatic Generation with `/init`

You don't have to write `AGENTS.md` by hand. The `/init` command analyzes your project and generates a suitable file:

```
/init
```

OpenCode then scans:

- Project structure and folders
- Frameworks and libraries used
- Recognized architecture patterns
- Existing tests and CI configuration

The result is a ready-made `AGENTS.md` in your project folder that you can then customize.

11.5 How Do I Tell OpenCode Explicitly Which Rules to Apply?

Method 1 – Automatically via File in the Project Folder

The simplest method: place `AGENTS.md` in the project folder. OpenCode reads it at startup automatically.

Method 2 – Globally via AGENTS.md in the Config Folder

Rules that should apply to all projects:

```
~/.config/opencode/AGENTS.md
```

Method 3 – System Prompt in opencode.json

In `~/.config/opencode/opencode.json` or in the project's `opencode.json`:

```
{

  "system": "Always write comments in English. No force unwraps. Ask before file changes."

}
```

Method 4 – Directly in the Prompt at the Start of the Session

If no file is available, rules can be given directly:

```
For this session: No force unwraps, comments in English,

show me changes before execution.
```

This only applies to the current session. After `/clear` it is forgotten.

11.6 Example AGENTS.md for a Swift Project

```
# Project Rules for MyProject



## Architecture

- Maintain MVVM pattern

- Network logic belongs in services, not in ViewModels

- No logic directly in SwiftUI views



## Code Style

- No force unwraps (`!`)

- No implicit force casts (`as!`)

- Code identifiers in English

- Comments and DocC in English
```

```
## Safety Rules

- SwiftLint must remain green

- No new dependencies without asking

- No changes to CI scripts

- Never delete tests


## Working Style

- Show me all files you would change first

- Wait for my confirmation before changes

- When in doubt: ask rather than guess
```

11.7 Recommended Content

```
1. Architecture requirements (MVVM, Clean Architecture, …)

2. Code style (comments, language, naming)

3. Prohibitions (force unwrap, certain patterns)

4. Shell restrictions (what may be executed)

5. Confirmation requirements (when must the agent ask)

6. Lint and test requirements
```

Tip: Start with `/init`, read the generated file, customize the rules. A growing `AGENTS.md` is a good sign that the agent is being used productively.

12 Memory and Context

OpenCode has no real long-term memory. However, there are several levels that together act like memory.

12.1 Session Context

The current conversation is retained during the session. In personal tests, local 14B models lose context faster than cloud models.

12.2 Repository Context

OpenCode actively reads files, folders, and git history. This acts like memory but is actually ongoing project analysis.

12.3 Instruction Files as Persistent Memory

This is the most important form. Files like `AGENTS.md` specify rules, architecture, and style and are available on every startup. More on this in Chapter 11.

12.4 Experience with Local Models

From personal experience, local models forget faster. Cloud models like Claude Opus or GPT-4 retain architecture, rules, and relationships longer.

Memory Type	Local (14B)	Claude Opus
Session context	medium	very good
Architecture rules	good with file	very good
Long file chains	limited	significantly better
Agent steps	5–10 stable	20+ stable

These values are experience-based estimates, not benchmarks.

13 Built-in Commands and Slash Commands

OpenCode has built-in commands that you can type at any time in the TUI. They control the agent's behavior without asking a new question.

13.1 Slash Commands Overview

Command	What Happens
`/init`	Analyze project and automatically generate AGENTS.md
`/help`	Show all available commands
`/model`	Switch model (e.g. `/model qwen2.5:32b`)
`/models`	Show list of all configured models
`/clear`	Reset conversation history
`/undo`	Undo last file changes
`/redo`	Restore undone changes
`/share`	Generate link to the current conversation
`/connect`	Connect a new provider
`/summarize`	Generate summary of the current context

13.2 Switching Models

With `/model`, the model can be swapped mid-session, for example from a fast 7B to a stronger model when a task becomes more complex:

```
/model qwen2.5:32b
```

13.3 Resetting the Session

When the context becomes too large or the model forgets instructions:

```
/clear
```

This deletes the conversation history, but not the project files or `AGENTS.md`.

13.4 Undoing Changes

If OpenCode made changes you didn't want:

```
/undo
```

This rolls back the last file changes. Multiple `/undo` commands go further back.

13.5 Referencing Files and Shell Output in Prompts

You can include files directly in the prompt via `@` syntax — OpenCode then searches using fuzzy search:

```
Explain to me @SearchService and how it relates to @NetworkLayer.
```

Shell output can be embedded directly:

```
The command `!swift test` is failing. What is wrong?
```

OpenCode executes the command and provides the output as context.

13.6 Showing Help

`/help` shows all currently available commands for the installed version:

```
/help
```

Note: Commands evolve with each version. When in doubt, type `/help`.

14 Practical Test: Complete Setup

This section documents a complete setup and practical test with Ollama and OpenCode on macOS with Apple Silicon. The results complement the theoretical chapters with real observations — particularly regarding Tool-Calling behavior, configuration, and prompt strategy. This section intentionally contains practical repetitions from earlier chapters so that it remains readable as a standalone guide.

>

Tested with: Ollama 0.22.0, qwen2.5:32b, Apple M4 Max (28 GB RAM), OpenCode current – May 2026

Step 1 – Install Ollama

```
brew install ollama
```

```
ollama --version
```

Step 2 – Choose the Right Model

Model selection is the most important decision. Not every model supports Tool-Calling in OpenCode + Ollama equally reliably. Tool-Calling is the prerequisite for OpenCode to actually execute commands instead of just outputting JSON text.

Recommendation from personal tests: `qwen2.5:32b`

```
ollama pull qwen2.5:32b
```

```
ollama list
```

Personal Test Results Compared

This table summarizes personal observations. Behavior may differ on other hardware or with newer versions.

Model	Tool-Calling (personal tests)	Suitability	Note
`qwen2.5:32b`	correct	good for code	recommended, ~19 GB
`llama3.2:latest`	correct	limited	only 3B – too weak for many code tasks
`qwen2.5-coder:14b`	not reliable	not for agent mode	Tool calls partially appeared as text
`hhao/qwen2.5-coder-tools:14b`	not reliable	not for agent mode	Community variant, same issue

> **Possible explanation:** *The Ollama modelfile of some models appears not to contain a properly configured Tool-Calling template. Ollama then cannot translate the tool call output into the* `tool_calls` *API field and delivers it as text instead. This assessment is based on personal tests, not an official statement.*

Manually Testing a Model's Tool-Calling

```
curl http://localhost:11434/v1/chat/completions \

  -H "Content-Type: application/json" \

  -d '{

    "model": "qwen2.5:32b",

    "messages": [{"role":"user","content":"list files"}],

    "tools": [{"type":"function","function":{"name":"bash","description":"run
shell","parameters":{"type":"object","properties":{"command":{"type":"string"}},"
required":["command"]}}}]

  }'
```

- Response contains `"finish_reason": "tool_calls"` → model supports Tool-Calling in the test
- Response contains `"finish_reason": "stop"` with JSON in the `content` field → model delivers tool call as text instead of structured

Step 3 – Install OpenCode

```
npm install -g opencode-ai

# or

curl -fsSL https://opencode.ai/install | bash

# or

brew install sst/tap/opencode
```

Step 4 – Create the Configuration

OpenCode does not automatically recognize Ollama as a provider. The configuration must be created manually:

```
mkdir -p ~/.config/opencode

nano ~/.config/opencode/opencode.json
```

Content (configuration that works in my own setup):

```
{
  "$schema": "https://opencode.ai/config.json",
  "provider": {
    "ollama": {
      "npm": "@ai-sdk/openai-compatible",
      "options": {
        "baseURL": "http://localhost:11434/v1"
      },
      "models": {
        "qwen2.5:32b": {
          "name": "Qwen 2.5 32B"
        }
      }
    }
  },
  "model": "ollama/qwen2.5:32b",
  "permission": {
    "bash": "allow",
    "read": "allow",
    "glob": "allow",
    "grep": "allow",
    "edit": "ask",
    "write": "ask",
    "task": "deny",
```

```
    "webfetch": "deny"
  }
}
```

Set `bash`, `read`, `glob`, and `grep` to `allow`, otherwise OpenCode asks for confirmation on every file access. Keep `edit` and `write` at `ask` for control over file changes.

Step 5 – Start OpenCode

Ollama must be running in the background. The macOS app starts the daemon automatically (icon in the menu bar). If necessary:

```
# Check if Ollama is running (port 11434):
pgrep -fl ollama || open -a Ollama


# Go to the project directory and start OpenCode:
cd ~/MyProject
opencode
```

> **Note:** `ollama serve` *is* **not** *needed when the Ollama app is installed. A second start would fail with* `address already in use`*. Use* `ollama serve` *only without the desktop app (pure CLI installation).*

Step 6 – Formulating Prompts Correctly

With local models, from personal experience, you need to guide the agent more strongly than with cloud tools like Claude Code. The model should be explicitly instructed which steps to execute. General prompts like "Analyze this project" do not always work reliably.

Project analysis:

```
Respond in English.


Analyze this Swift project carefully and step by step.


Important:
- Do not change any files.
- Do not create any files.
- Do not use subagents.
- Do not use webfetch.
```

```
- Read the project structure first.

- Do not invent dependencies or architecture layers.


Check first:

- pwd

- ls -la

- find . -maxdepth 4 -type f \( -name "*.swift" -o -name "Package.swift" -o -name
"*.pbxproj" \) | sort


Then read the most important files and create an analysis:

1. Project type

2. Recognizable layers

3. Dependencies

4. Problematic areas

5. Recommended next steps
```

Bug search in a file:

```
Respond in English.

Do not change any files.


Search the project for a file containing "DependencyContainer" in its name:

  find . -name "*DependencyContainer*.swift"


Then read the found file.

Check for obvious errors, missing imports, incorrect initialization, or syntax
issues.

Analyze only the actually read content.
```

Syntax check for all Swift files:

```
Respond in English.

Do not change any files.
```

```
Execute:

  find . -name "*.swift" | sort


Read each file individually.

Check for syntax errors (wrong keywords, missing brackets, incorrect
declarations).

Report only real errors from the read content.
```

Troubleshooting from Practice

Problem: OpenCode Shows JSON Instead of Results

```
{

  "name": "bash",

  "arguments": { "command": "ls" }

}
```

Possible cause: The model does not support Tool-Calling reliably in the current Ollama variant.

Solution: Switch model (e.g. to `qwen2.5:32b`), update `opencode.json`, restart OpenCode.

Problem: OpenCode Explains Commands but Does Not Execute Them

Symptom: OpenCode writes "Execute `cat File.swift` and paste the content here" instead of reading the file itself.

Possible cause: The model behaves like a chatbot and does not use tools autonomously.

Solution: Formulate prompts more explicitly:

```
Use bash directly.

Execute yourself: find . -name "DependencyContainer.swift"

Then read the found file yourself.

Do not wait for my input.
```

Problem: Ollama Does Not Respond

```
lsof -i :11434                          # check if port is occupied

ollama serve                            # start Ollama manually (only without
desktop app)

curl http://localhost:11434/api/tags    # test connection
```

OpenCode + Ollama vs. Claude Code – Practical Comparison

The following statements are factual practical observations. Both tools have their place — they simply show their strengths in different situations.

Criterion	OpenCode + Ollama	Claude Code
Tool-Calling	Heavily dependent on model template	Native, very reliable in tests
Prompt guidance	Benefits from explicit step-by-step instructions	Usually captures context automatically
Autonomy	Often executes actions only on instruction	Searches and reads files itself
Privacy	Fully local possible	Cloud-based
Cost	Free after hardware investment	Usage-based API costs
Model quality	Limited by local computing power	Access to large cloud models
Consistency	Varies by model and prompt	Consistently high in tests

Conclusion: OpenCode with Ollama is well suited when privacy or offline operation are the priority. For complex, multi-step tasks — architecture analyses, refactorings across many files, debugging chains — Claude Code is currently often more comfortable, as it derives context itself and works with significantly more powerful models.

15 Built-in Tools Overview

OpenCode comes with a set of built-in tools that the model can use during a session. You don't need to call them manually — the model decides which tools are needed for a task.

Tool	What It Does
`bash`	Execute shell commands (git, swift, xcodebuild, …)
`edit`	Edit existing files (exact string replacement)
`write`	Create new files or completely overwrite
`read`	Read file contents
`grep`	Search files with regular expressions
`glob`	Find files by pattern matching (e.g. `**/*.swift`)
`lsp`	Code intelligence via LSP (definitions, references)
`patch`	Apply patch files
`todowrite`	Manage to-do lists during the session
`webfetch`	Retrieve web pages and use as context
`websearch`	Search the web
`question`	Ask you for clarification during a task

All tools can be controlled via the permissions system (Chapter 16). Individual tools can be allowed, set to ask, or completely blocked.

Important note for local models: For the model to call these tools autonomously, Tool-Calling support is necessary. In personal tests, this does not work equally reliably with all Ollama models (see Chapters 8 and 14).

16 The Permissions System

OpenCode has a fine-grained system that controls whether actions are executed automatically, require confirmation, or are completely blocked.

16.1 The Three Actions

Action	Meaning
`"allow"`	Action is executed without asking
`"ask"`	OpenCode asks you before execution
`"deny"`	Action is completely blocked

16.2 What Can Be Controlled?

Category	Meaning
`read`	Read files
`edit`	Edit files
`write`	Create/overwrite files
`bash`	Execute shell commands
`glob`	Find files by pattern
`grep`	Search files
`webfetch`	Retrieve web pages
`websearch`	Search the web
`task`	Start subagents
`external_directory`	Access to folders outside the project
`doom_loop`	Block identical repetition loops

16.3 Configuration in opencode.json

Set a global default rule and override individual tools:

```
{
  "permission": {
    "*": "ask",
    "read": "allow",
    "bash": "allow",
    "edit": "ask"
  }
}
```

Fine-grained control with wildcard patterns — especially important for shell commands:

```
{
  "permission": {
    "bash": {
      "*": "ask",
      "git status": "allow",
      "git diff": "allow",
      "swift test": "allow",
      "rm *": "deny",
```

```
      "git reset *": "ask"
    }
  }
}
```

16.4 Recommendation for Swift/Xcode Projects

```
{
  "permission": {
    "*": "ask",
    "read": "allow",
    "glob": "allow",
    "grep": "allow",
    "bash": {
      "*": "ask",
      "git status": "allow",
      "git diff": "allow",
      "swift build": "allow",
      "swift test": "allow",
      "xcodebuild test *": "allow",
      "rm *": "deny",
      "git reset --hard": "deny"
    }
  }
}
```

Tip: Start with `"*": "ask"` (ask for everything), observe what OpenCode does, and gradually expand the `allow` list. This way you get to know the behavior before building trust.

Default behavior: `.env` files are blocked by default. `external_directory` and `doom_loop` ask for confirmation by default.

17 OpenCode as an Agent – What It Can Really Do

Many developers install OpenCode and treat it like a normal chat. This is a misunderstanding.

OpenCode is:

- a **terminal agent**
- a **file agent**
- a **shell agent**
- a **workflow agent**

17.1 What Really Happens

When you write:

```
Refactor my project
```

OpenCode can internally:

- change 40 files
- start tests
- execute Bash scripts
- run git commands
- move files

Warning: Always run `git checkout -b ai/test` first. Then check `git diff` before accepting anything.

17.2 Behavior with Local Ollama Models

An important practical point: OpenCode with local Ollama models does not always behave like a cloud tool such as Claude Code. From personal experience:

- General prompts like "Analyze this project" do not work reliably in every setup.
- Local models don't always use tools autonomously. Some models behave like a chatbot and describe the step instead of executing it.
- Tool calls can appear in some Ollama models as JSON in the response text, rather than being correctly passed as a structured tool call to OpenCode.

In practice, it helps to explicitly instruct OpenCode:

```
First read the project structure (e.g. with ls and find).

Then search specifically for the relevant files.

Actually read these files.

Evaluate the real terminal output.

Only then make an assessment.

Execute Bash commands yourself, do not wait for my output.
```

This observation is a practical insight from personal tests, not necessarily universal. With future versions of Ollama, OpenCode, or improved modelfiles, this picture may change.

17.3 The Best Workflow

```
Bad: "Refactor my entire app."


Better:

1. "First analyze the architecture."

2. "Suggest refactoring steps."

3. "Only change the search layer."
```

Small, clear tasks deliver better results than large, vague instructions. This is especially true for local models.

18 File System Access

OpenCode works directly in the current project folder. Depending on the configuration, it can read, modify, create, delete files, and use git.

18.1 Important for macOS

OpenCode has the same rights as your user. If your user has access to SSH keys, certificates, company repositories, or keychain files, OpenCode can theoretically work with them.

This does not automatically mean danger, but you should be aware of it and control it with the permissions system (Chapter 16).

18.2 Recommendation for Company Projects

```
separate macOS user for AI work

sandbox projects without production access

test repositories

no unnecessary keys in the AI working folder
```

19 Shell and Bash Access

OpenCode can execute commands directly in the terminal:

```
git · find · grep · swift · xcodebuild · npm · brew · rm · mv
```

This makes it powerful. When you write "Start the tests and fix the errors", the following might happen internally:

```
xcodebuild test -scheme MyScheme ...

grep -r "ErrorMessageX" .
```

```
swift test
```

19.1 What Should Always Be Checked

Always review these commands before confirming:

```
rm -rf

mv

chmod

git reset --hard

git clean -fd
```

Warning: Never blindly confirm destructive commands. The permissions system in Chapter 16 helps to specifically block these.

20 The OpenAI API Structure Explained

This chapter is for developers who want to build their own apps or scripts against Ollama.

20.1 What Is the OpenAI API Structure?

The OpenAI API structure is an HTTP-based protocol that OpenAI defined for its models and that has since become the de facto industry standard. Many providers and local tools implement the same endpoints.

20.2 Native Ollama API vs. OpenAI-Compatible API

Ollama offers both variants in parallel:

```
Native Ollama API:

  POST /api/generate

  POST /api/chat

  GET  /api/tags


OpenAI-compatible API:

  POST /v1/chat/completions

  POST /v1/completions

  GET  /v1/models
```

The OpenAI-compatible variant is usually the right choice when a tool or library was already built for OpenAI. The native Ollama API offers additional, Ollama-specific options.

With Ollama locally:

```
http://localhost:11434/v1/chat/completions

http://localhost:11434/v1/models

http://localhost:11434/api/tags
```

20.3 The Request Format (OpenAI-Compatible)

A request is a JSON object with `model` and `messages`:

```
{
  "model": "qwen2.5:32b",
  "messages": [
    { "role": "system", "content": "You are a Swift expert." },
    { "role": "user",   "content": "Explain Swift Optionals." }
  ],
  "temperature": 0.7
}
```

Role	Meaning
`system`	System instruction – character and rules of the model
`user`	Your input
`assistant`	Model's response (for conversation history)

20.4 The Response Format

```
{
  "choices": [
    {
      "message": {
        "role": "assistant",
        "content": "An Optional in Swift is..."
      }
    }
  ],
  "usage": {
    "prompt_tokens": 42,
    "completion_tokens": 120
  }
}
```

20.5 Why Has This Format Become the Standard?

OpenAI was early to market, and many tools were developed against this API. When Anthropic, Google, Mistral, and others arrived, the OpenAI structure became the common basis. It is an industry standard through market dominance, not an open standard through committee.

20.6 The Practical Advantage

Every library written for OpenAI often works with Ollama — with a single change:

```
# OpenAI (Cloud)

client = OpenAI(base_url="https://api.openai.com/v1", api_key="sk-...")


# Ollama (local) - same library, different URL

client = OpenAI(base_url="http://localhost:11434/v1", api_key="ollama")
```

This is also why Continue, Cline, and other tools can use Ollama without their own SDK support.

21 MCP Servers – Integrating External Tools

MCP (**Model Context Protocol**) is a standard that allows AI agents to use external tools. OpenCode supports MCP servers — this allows you to give the agent tools that go beyond the built-in tools.

21.1 What Are MCP Servers?

MCP servers are external processes that OpenCode makes available as tools. After configuration, the model uses them automatically alongside the built-in tools.

Practical examples:

- **Sentry** – errors and issues directly in the agent context
- **Context7** – retrieve current documentation from the web
- **GitHub** – read and create issues and PRs

21.2 Configuration in opencode.json

Local MCP server (runs on your Mac):

```
{
  "mcp": {
    "my-tool": {
      "type": "local",
      "command": ["npx", "-y", "my-mcp-package"]
```

```
    }
  }
}
```

Remote server:

```
{
  "mcp": {
    "my-remote-tool": {
      "type": "remote",
      "url": "https://mcp-server.example.com"
    }
  }
}
```

21.3 Authentication

For servers with OAuth authentication:

```
opencode mcp auth my-tool
```

OpenCode then opens the OAuth flow automatically.

21.4 Important Note for Ollama Users

If tool calls via MCP do not work reliably, it can help to increase the context window:

```
ollama run qwen2.5:32b --num_ctx 8192
```

Local models sometimes need more context for complex Tool-Calling.

22 Custom Commands – Creating Your Own Commands

You can define your own slash commands for recurring tasks. This is particularly useful for Swift project routines like starting tests, checking lint, or generating documentation.

22.1 Where Are Custom Commands Located?

```
MyProject/
└── .opencode/
    └── commands/
        ├── test.md
        ├── lint.md
```

```
        └── docc.md
```

22.2 Format of a Command File

```
---

description: Start tests and explain errors

---


Start the tests with `swift test`. Show all errors and explain

what is wrong in each individual case. Do not change any files yet.
```

After startup, `/test` is available as a command.

22.3 With Arguments

```
---

description: Generate DocC for a class

---


Generate English DocC comments for the class $1. Code identifiers

remain in English. Check the current state first with @$1.
```

Usage: `/docc SearchService`

22.4 Embedding Shell Output

```
---

description: Check and fix SwiftLint

---


SwiftLint reports the following errors:

!swiftlint lint --reporter json


Fix all auto-fixable issues. Show me the remaining errors.
```

23 OpenCode vs. Claude Code – Comparison

Claude Code is model + agent + workflow + tuning in one closed system. OpenCode is an open agent framework — quality depends strongly on the chosen model. Both have their place; the following comparison is meant to be factual.

Area	Claude Code	OpenCode
Setup	very convenient	more technical
Model quality	very high	depends on the chosen model
Local models	not the main path	very well suited
Privacy	mostly cloud	locally possible
Provider freedom	limited	very high (75+ providers)
Cost	subscription/API	locally free (except hardware)
Agent stability	usually more stable	varies by model
Tool-Calling	very mature	model-dependent
Permissions	integrated	configurable (Chapter 16)
MCP support	yes	yes

23.1 Assessment

```
Claude Code             → comfortable and automatic for large, multi-step tasks

OpenCode + Ollama       → strong for local control, cost, and privacy

OpenCode + Cloud API    → combines provider freedom with strong models
```

Tip: OpenCode and Claude Code are not mutually exclusive. Many developers use OpenCode with Ollama for everyday tasks and turn to Claude Code for large refactorings.

24 Comparison with Codex, Gemini CLI, and Grok

A brief overview of other well-known terminal agents and cloud tools.

24.1 OpenAI Codex

Cloud-based software engineering agent. Can write features, fix bugs, and suggest pull requests. Tasks run in isolated cloud sandboxes with repository context.

24.2 Gemini CLI

Open-source agent in the terminal. Makes Gemini directly usable in the terminal and works via a reason-and-act loop with tools and MCP servers.

24.3 Grok

xAI Grok 4 is a model for agentic reasoning, knowledge work, and tool use. Primarily available via API and web interface.

24.4 Overall Overview

Tool	Type	Strength
Claude Code	Premium coding agent	very strong for large projects
OpenAI Codex	Cloud coding agent	tasks, PRs, bug fixes
Gemini CLI	Terminal agent	strong with Google ecosystem and MCP
OpenCode	Open terminal agent	provider freedom, Ollama, local

		models
Cline/Roo Code	IDE agent	good VS Code integration
Continue	IDE extension	chat, completion, local models

25 Performance Overview of Known Models

The following values are rough practical assessments from personal experience, **not official benchmarks***. They may vary significantly depending on hardware, task, prompt, and model version.*

Model	Type	Practical Coding Assessment
Claude Opus + Claude Code	Premium Agent	95–100%
GPT / Codex	Premium Agent	93–98%
Gemini Pro / Gemini CLI	Premium Agent	90–96%
Grok 4.x	Premium Model/API	88–94%
Claude Sonnet	Premium Everyday	86–93%
DeepSeek R1 32B/70B	Open Reasoning	80–90%
Qwen 2.5 32B	Open All-round	80–88%
Qwen2.5-Coder 32B	Open Coding	80–88%
Qwen2.5-Coder 14B	Open Coding	74–82%
DeepSeek-Coder	Open Coding	72–82%
Qwen2.5-Coder 7B	Open Coding	68–76%
Gemma / Phi	Small models	62–74%
Code Llama	Older, solid	58–70%

Note: An 80% model is not bad. For simple and medium tasks it is often sufficient. For large refactorings and long agent chains, the gap to cloud models becomes noticeable.

26 Which Models Should an Apple Developer Use?

The answer depends on hardware, task, and working style. The following recommendations are based on practical experience — always test in your own setup.

26.1 Local Starting Combination

For many developers with 32 GB RAM — one model for code, one for analysis:

```
ollama pull qwen2.5-coder:14b
```

```
ollama pull deepseek-r1:14b
```

Those who want to use real Tool-Calling with OpenCode should additionally or alternatively load a model with reliable Tool-Calling:

```
ollama pull qwen2.5:32b
```

26.2 For More Powerful Macs (48 GB+ RAM)

Those with a Mac Studio or MacBook Pro with 48 GB+ RAM can use the 32B variants — stronger, results closer to cloud models:

```
ollama pull qwen2.5:32b
```

```
ollama pull qwen2.5-coder:32b
```

26.3 For Smaller Macs (16 GB RAM)

The 7B model — fast, small, often sufficient for many everyday tasks:

```
ollama pull qwen2.5-coder:7b
```

26.4 Decision Guide

Situation	Possible Choice
Daily coding tasks (local)	Qwen2.5-Coder 14B or 32B
Coding with agent actions / tool calls	Qwen 2.5 32B (personal tests)
Architecture and analysis	DeepSeek R1 14B / 32B
Quick questions, less RAM	Qwen2.5-Coder 7B
Large refactorings	Claude Code or Codex
Sensitive company projects	Ollama local

27 Strengths and Weaknesses of Local Models

Local models have clear advantages and disadvantages compared to cloud services. Knowing both helps with the right task selection.

27.1 Strengths

```
local and private

no ongoing API costs

no rate limits

no server outages from providers

good for everyday code

good for explanations and tests

good for offline work
```

27.2 Weaknesses

```
weaker with very large codebases

slower with large models

more configuration needed

less reliable with long agent chains

Tool-Calling heavily model-dependent

small models hallucinate faster

limited context window

more own hardware needed
```

27.3 Most Important Point

Local models are not automatically worse. They benefit from:

- better task formulation
- smaller work packages
- clear rule files (`AGENTS.md`)
- configured permissions for control

28 Hybrid Workflow: Combining Local and Cloud

Many Apple developers today use a combination — local models for everyday use, cloud for heavy tasks:

Task	Possible Tool
Local daily coding	OpenCode + Qwen
Architecture analysis	DeepSeek R1
Large refactorings	Claude Code
Quick code completion	local models
Sensitive projects	Ollama local
Massive agent tasks	Claude / GPT / Codex
Offline / airplane	Ollama local
Test generation	Qwen + OpenCode
Documentation	local models

28.1 The Idea Behind It

```
Local AI = standard tool for many everyday tasks

Cloud AI  = reserve for heavy and complex tasks
```

This saves costs and gives control without sacrificing quality on difficult tasks.

29 Troubleshooting

This section helps with the most common problems when getting started.

29.1 Ollama Does Not Respond

Check if the server is running:

```
curl http://localhost:11434

# Expected response: Ollama is running
```

If not:

```
ollama serve
```

(Only needed without the desktop app.)

29.2 OpenCode Does Not Start

Finding log files:

```
macOS/Linux: ~/.local/share/opencode/log/
```

Restart OpenCode and watch for error messages:

```
opencode --log-level debug
```

29.3 Tool Calls Don't Work (Especially with Ollama)

Local models sometimes need a larger context window for Tool-Calling. In the Ollama configuration or at startup:

```
ollama run qwen2.5:32b --num_ctx 8192
```

If that doesn't help, switch the model (see Chapters 8 and 14).

29.4 Model Runs Very Slowly

The model may not fit in RAM — Ollama then swaps parts to SSD. Solution: use a smaller model.

```
ollama list  # check memory requirements
```

As a rule of thumb: the model needs about 1 GB per billion parameters.

29.5 OpenCode Made Unwanted Changes

Undo changes:

```
/undo
```

Or completely reset via git:

```
git checkout .
```

29.6 Where to Get Help?

```
GitHub Issues:  github.com/sst/opencode

Discord:        Invitation link on opencode.ai

Ollama Discord: discord.gg/ollama
```

Include logs from `~/.local/share/opencode/log/` in bug reports.

30 Conclusion

Ollama is more than a toy for developers by now. In combination with OpenCode, you get a local coding agent that is sufficient for many everyday tasks. Qwen2.5-Coder is a strong choice for pure code tasks. For agent mode with Tool-Calling, Qwen 2.5 32B proved more reliable in personal tests. DeepSeek R1 is a useful complement for analysis and technical decisions.

Claude Code, Codex, and Gemini CLI remain often more comfortable for large, difficult tasks. For local work, privacy, and recurring developer tasks, OpenCode with Ollama is an excellent addition.

30.1 Practical Recommendation

```
Qwen 2.5 32B          → standard model for agent mode with Tool-Calling

Qwen2.5-Coder 14B/32B → code model for explanations and suggestions

DeepSeek R1 14B       → analysis and architecture

OpenCode              → local agent

AGENTS.md             → define project rules

Permissions           → configure security

Claude Code           → for particularly heavy tasks
```

30.2 Reality 2026

Local AI is to be taken seriously. The agent systems are still young — sometimes impressive, sometimes cumbersome. The best results come from:

```
small, clear tasks

clean git state

good AGENTS.md

configured permissions

controlled changes
```

31 Further Resources

31.1 Official Documentation (Sources for This Guide)

The following resources served as the main sources for this guide. Content was adapted from them in paraphrase, independently structured, and translated.

Resource	URL	License	Content
OpenCode Docs (DE)	https://opencode.ai/docs/de	MIT	complete documentation in German
OpenCode GitHub	https://github.com/sst/opencode	MIT	source code, issues, changelog
Ollama Docs	https://docs.ollama.com	MIT	Ollama API, commands, configuration
Ollama GitHub	https://github.com/ollama/ollama	MIT	source code, model formats
Ollama Library	https://ollama.com/library	—	all available models with description

31.2 Important Subpages of the OpenCode Documentation

Page	When Relevant
opencode.ai/docs/de/rules/	AGENTS.md and rule sets
opencode.ai/docs/de/permissions/	permissions configuration
opencode.ai/docs/de/config/	full opencode.json

opencode.ai/docs/de/mcp-servers/	integrating external tools
opencode.ai/docs/de/providers/	configuring all providers
opencode.ai/docs/de/commands/	creating custom commands
opencode.ai/docs/de/troubleshooting/	troubleshooting
opencode.ai/docs/de/keybinds/	customizing keyboard shortcuts

31.3 Community

```
GitHub:   github.com/sst/opencode

Discord:  Link on opencode.ai
```

32 Glossary

Agent: An AI system that not only answers but can read and modify files and execute commands.

AGENTS.md: Instruction file for OpenCode. Contains project rules, architecture requirements, and behavioral rules for the agent. Equivalent to `CLAUDE.md` in the Claude Code ecosystem.

Apple Silicon: ARM-based chip family from Apple (M1–M4). Unified Memory makes local AI models more efficient in practice than on conventional PC hardware.

auth.json: File in `~/.local/share/opencode/` that stores API keys and authentication tokens for providers.

Context Window: The maximum amount of text a model can process at once. Often smaller for local models than for cloud models.

CLAUDE.md: Instruction file from the Claude Code ecosystem. Also read by OpenCode.

Custom Commands: Custom slash commands for recurring tasks, defined as markdown files in `.opencode/commands/`.

DeepSeek R1: Open reasoning model, in personal tests strong for analysis, architecture, and technical decisions.

Hallucination: When an AI model generates plausible-sounding but incorrect information — especially a risk with small models.

Homebrew: Package manager for macOS. Ollama is installed with `brew install ollama`.

Leader Key: Key in OpenCode (default: `ctrl+x`), after which further keyboard shortcuts follow. Avoids conflicts with terminal shortcuts.

LLM: Large Language Model. Large language models like GPT-4, Claude Opus, or Qwen 2.5.

MCP: Model Context Protocol. Standard for external tools in AI agents.

Model Size: Measured in billions of parameters (7B, 14B, 32B). More parameters = usually stronger, but more RAM needed.

num_ctx: Context window size in Ollama. Increasing it can help with complex Tool-Calling.

Ollama: Local model runner. Starts and manages LLMs on your own hardware.

OpenCode: Open terminal agent for coding tasks. Supports 75+ providers.

OpenAI API Structure: HTTP protocol defined by OpenAI that has become the de facto industry standard. Ollama implements the endpoints at `/v1/...`.

Permissions: Configuration in `opencode.json` that controls which actions are allowed, asked about, or blocked.

Plan Mode: Mode in OpenCode (Tab key), in which the agent only plans and does not change files.

Provider: Provider of AI models (Anthropic, OpenAI, Google, Ollama, …).

Qwen 2.5 / Qwen2.5-Coder: Models from the Qwen family by Alibaba Cloud. Qwen 2.5 is an all-round model, Qwen2.5-Coder is specialized for code.

Quantization: Technique for reducing model size. Enables running large models on less RAM.

Reasoning Model: Model that performs internal thinking steps before answering. DeepSeek R1 is an example.

Session Context: Conversation history of the current session. Deleted after `/clear`.

Tool-Calling: Mechanism by which a model passes structured tool calls to the agent (`tool_calls` field in the response) — prerequisite for OpenCode to actually execute commands.

TUI: Terminal User Interface. Interactive interface directly in the terminal — no graphical window.

Unified Memory: Apple architecture in which CPU, GPU, and Neural Engine share the same RAM. No separate GPU VRAM needed — ideal for local AI.

xcodebuild: Apple CLI for building and testing Xcode projects from the terminal.

Disclaimer: *This guide was independently created based on publicly available sources and written in the author's own words. The primary sources were the official Ollama documentation (ollama.com, MIT License), the official OpenCode documentation (opencode.ai/docs, MIT License), as well as personal experience, community contributions, and testing. Technical facts such as API endpoints, configuration options, and commands were taken from these documentations and incorporated as such. All texts were independently structured and written — no verbatim reproduction of copyrighted content has occurred. Code examples are based on the official examples of the respective projects. The content provided is solely for educational purposes; no guarantee of completeness or accuracy is made. All mentioned brands, products, and technologies belong to their respective owners.*

Date: May 2026 — Sources: opencode.ai/docs/de (MIT) · ollama.com (MIT) · github.com/sst/opencode · github.com/ollama/ollama