SPEC-DOC-001: User Onboarding & Getting Started Guide

Status: Pending **Priority**: P0 (High) **Estimated Effort**: 8-12 hours **Target Audience**: New users **Created**: 2025-11-17

Objectives

Create comprehensive onboarding documentation that enables new users to: 1. Install the codex CLI on their system (npm, Homebrew, or from source) 2. Complete first-time setup (authentication, configuration) 3. Execute their first AI-assisted coding task within 15 minutes 4. Understand common workflows (spec-kit automation vs manual coding) 5. Troubleshoot common issues independently 6. Find answers to frequently asked questions

---

Scope

In Scope

Installation Guide: - npm installation (recommended) - Homebrew installation (macOS/Linux) - Building from source (all platforms) - System requirements and dependencies - Verification steps

First-Time Setup: - API key configuration (OpenAI, Anthropic, Google) - auth.json setup - Basic config.toml configuration - MCP server setup (local-memory, git-status) - Workspace initialization

Quick Start Tutorial (5-minute walkthrough): - Interactive chat mode - Running first spec-kit command (/speckit.new) - Understanding agent responses - Basic file operations

Common Workflows: - Spec-kit automation (full pipeline /speckit.auto) - Manual coding assistance (chat mode) - Code review and refactoring - Running tests and validation

Troubleshooting Guide: - Installation errors - Authentication issues - MCP connection problems - Agent execution failures - Performance issues - Common configuration mistakes

FAQ: - Model selection and switching - Cost management - Offline usage - Privacy and data handling - Customization options - Comparison with other tools (Cursor, Copilot)

Out of Scope

- Advanced configuration (see SPEC-DOC-006)
- Internal architecture details (see SPEC-DOC-002)
- Contributing to the project (see SPEC-DOC-005)
- Security deep-dive (see SPEC-DOC-007)

Deliverables

Primary Documentation

1. **installation.md** - Comprehensive installation guide
2. **first-time-setup.md** - Setup walkthrough
3. **quick-start.md** - 5-minute tutorial
4. **workflows.md** - Common usage patterns
5. **troubleshooting.md** - Error resolution guide
6. **faq.md** - Frequently asked questions

Supporting Materials

- **evidence/screenshots/** - Installation screenshots, UI examples
- **evidence/terminal-examples/** - Command outputs, typical

workflows

Success Criteria

- [ ] New user can install within 5 minutes
- [ ] New user can complete setup within 10 minutes
- [ ] New user can run first command within 15 minutes total
- [ ] Troubleshooting guide addresses 90%+ common errors
- [ ] FAQ answers 30+ common questions
- [ ] All installation paths tested (npm, Homebrew, source)
- [ ] All screenshots current and clear

Related SPECs

- SPEC-DOC-000 (Master)
- SPEC-DOC-002 (Architecture - for advanced users)
- SPEC-DOC-003 (Spec-Kit Framework - detailed command reference)
- SPEC-DOC-006 (Configuration - advanced customization)

Status: Structure defined, content pending

Frequently Asked Questions (FAQ)

Comprehensive answers to common questions about Code CLI.

Table of Contents

General Questions

What is Code CLI?

Code (also `@just-every/code`) is a fast, local coding agent for your terminal. It's a community-driven fork of `openai/codex` focused on real developer ergonomics:

- 🌐 Browser integration (CDP support, headless browsing)
- 📝 Diff viewer (side-by-side diffs with syntax highlighting)

- 🤖 Multi-agent commands (/plan, /solve, /code)
- 🎨 Theme system with live preview
- 🧠 Reasoning control (dynamic effort adjustment)
- 🔌 MCP support (Model Context Protocol)
- 🔒 Safety modes (read-only, approvals, sandboxing)

Fork Enhancements (theturtlecsz/code): - **Spec-Kit Framework**: Multi-agent PRD automation pipeline - **Native MCP integration**: 5.3× faster than subprocess - **Quality gates**: Configurable consensus checkpoints - **Evidence repository**: Automated telemetry collection

---

How is this different from the original OpenAI Codex?

OpenAI Codex (2021): - AI model for code generation (deprecated March 2023) - Not related to this CLI tool

Code CLI (this project): - Community fork of `openai/codex` terminal interface - Adds browser integration, multi-agent workflows, themes - Maintains full compatibility with upstream - Uses modern models (GPT-5, Claude, Gemini)

Is this affiliated with OpenAI or Anthropic?

No. Code is a community-driven open source project: - **Not** affiliated with, sponsored by, or endorsed by OpenAI - **Not** affiliated with Anthropic (though supports Claude via CLI) - **Not** related to "Anthropic's Claude Code" (different product) - Apache 2.0 license, community maintained

---

Can I use my existing Codex configuration?

Yes. Code maintains full backwards compatibility:

- Reads from both `~/.code/` (primary) and `~/.codex/` (legacy)
- Writes only to `~/.code/`
- Automatically migrates settings on first run
- Codex will keep running if you switch back

To migrate manually:

```
# Copy config
cp ~/.codex/config.toml ~/.code/config.toml

# Copy auth (if using ChatGPT login)
cp ~/.codex/auth.json ~/.code/auth.json
```

What operating systems are supported?

Officially Supported: - ✅ macOS 12+ (Monterey and later) - ✅ Ubuntu 20.04+ / Debian 10+ - ✅ Windows 11 **via WSL2** (Windows Subsystem for Linux)

Experimental: - ⚠ Other Linux distributions (Alpine, Fedora, Arch) - usually work - ⚠ Direct Windows install - may work but unsupported

Not Supported: - ❌ macOS 11 and earlier - ❌ Ubuntu 18.04 and earlier - ❌ Windows without WSL2

Model and Authentication

Which models are supported?

OpenAI Models (primary): - **GPT-5** (recommended, default: `gpt-5-codex`) - **GPT-4o** (faster, cheaper) - **GPT-4o-mini** (cheapest, good for simple tasks) - **o3**, **o4-mini** (reasoning models)

Anthropic Claude (via multi-agent setup): - **Claude Sonnet 4.5** (balanced) - **Claude Haiku 3.5** (cheap and fast) - **Claude Opus 3.5** (premium reasoning)

Google Gemini (via multi-agent setup): - **Gemini Pro 1.5** (balanced) - **Gemini Flash 1.5** (cheapest: $0.075/1M tokens)

Local Models (experimental): - Ollama support via custom provider configuration - Any OpenAI API-compatible endpoint

---

Do I need ChatGPT Plus or an API key?

You need ONE of the following:

Option 1: ChatGPT Subscription (no per-token billing) - ChatGPT Plus ($20/month) - ChatGPT Pro ($200/month) - ChatGPT Team (varies) - Uses models included in your plan - ✅ Best for: Regular interactive use

Option 2: OpenAI API Key (pay-as-you-go) - Usage-based billing (see pricing) - ✅ Best for: Automation, CI/CD, precise cost control

ChatGPT Free Tier does NOT work with Code CLI.

Can I switch between ChatGPT auth and API key?

Yes, anytime:

Switch to API key (from ChatGPT):

```
# Set API key environment variable
export OPENAI_API_KEY="sk-proj-YOUR_KEY"

# Or add to config.toml
echo 'preferred_auth_method = "apikey"' >> ~/.code/config.toml
```

Switch to ChatGPT (from API key):

```
# Remove API key
unset OPENAI_API_KEY

# Remove from .env if present
rm ~/.code/.env
```

```
# Re-authenticate
code login
```

Force specific method in config:

```
# ~/.code/config.toml
preferred_auth_method = "chatgpt"  # or "apikey"
```

Why does `o3` or `o4-mini` not work for me?

Possible causes:

1. **Account not verified**: Free tier API accounts need verification to access reasoning models
2. **Model not available to your account**: Some models require paid tier
3. **Wrong model name**: Use exact name (e.g., `o3` not `o-3`)

Solution:

```
# Check account verification at platform.openai.com
# Upgrade to paid tier if needed
# Or use GPT-5 instead:

code --model gpt-5 "your task"
```

Cost and Pricing

How much does it cost?

With ChatGPT Plus/Pro/Team: - **$0 per use** (covered by subscription) - Already paying for ChatGPT subscription

With API Key (pay-as-you-go):

Model Pricing (January 2025):

Provider	Model	Input (1M tokens)	Output (1M tokens)
OpenAI	GPT-5	$5.00	$15.00
	GPT-4o	$2.50	$10.00
	GPT-4o-mini	$0.15	$0.60
Anthropic	Claude Sonnet 4.5	$3.00	$15.00
	Claude Haiku 3.5	$0.80	$4.00
	Claude Opus 3.5	$15.00	$75.00
Google	Gemini Pro 1.5	$1.25	$5.00
	Gemini Flash 1.5	**$0.075**	**$0.30**

Typical Usage Costs:

Task	Model	Estimated Cost
Simple code explanation	GPT-4o-mini	~$0.01
Refactor function	GPT-4o	~$0.05
Generate module with tests	GPT-5	~$0.20
Full Spec-Kit pipeline	Multi-agent balanced	~$2.70
Complex architectural decision	o3 (high reasoning)	~$1.50

Cost-Saving Strategies:

1. **Use cheaper models for simple tasks**:

```
code --model gpt-4o-mini "format this file"
```

2. **Use Gemini Flash** (12× cheaper than GPT-4o):

```
[quality_gates]
tasks = ["gemini"]  # $0.075/1M vs $2.50/1M for GPT-4o
```

3. **Optimize Spec-Kit quality gates**:

```
# Cheap for simple stages, premium for critical
tasks = ["gemini"]           # ~$0.10
plan = ["gemini", "claude"]  # ~$0.35 (multi-agent)
audit = ["gpt-5"]            # ~$0.80 (premium for critical)
```

4. **Use native tools** (FREE):

```
/speckit.new        # $0 (native)
/speckit.clarify    # $0 (native heuristics)
/speckit.analyze    # $0 (native structural diff)
/speckit.checklist  # $0 (native rubric scoring)
```

Is there a free tier?

For API usage: - OpenAI free tier: 3 requests/min, 200 requests/day (very limited) - Anthropic free tier: 5 requests/min - Google Gemini free tier: 15 requests/min, 1,500 requests/day

Best free option: Use **Gemini Flash** (12.5× cheaper) or get ChatGPT Plus subscription (unlimited use within plan limits).

How can I monitor my costs?

With API Key:

1. **OpenAI Dashboard**: https://platform.openai.com/usage
   - Shows daily/monthly usage
   - Breakdown by model
   - Set spending limits
2. **Enable debug mode** to see token counts:

```
code --debug
```

3. **Use `--read-only` mode** for cost-free exploration:

   ```
   code --read-only "analyze this codebase"
   ```

With ChatGPT subscription: - No per-token billing - Covered by flat monthly fee

---

Privacy and Security

Is my data secure?

Yes. Code CLI follows these security practices:

1. **Authentication stays local**:
   - Credentials stored at `~/.code/auth.json` (0600 permissions)
   - No proxying through third-party servers
   - Direct communication with OpenAI/Anthropic/Google
2. **No telemetry by default**:
   - Code doesn't send usage data to project maintainers
   - Only communication is with AI providers you configure
3. **Conversation history**:
   - Stored locally at `~/.code/history.jsonl`
   - File permissions: 0600 (owner read/write only)
   - Can disable: `[history] persistence = "none"`
4. **Sandbox modes**:
   - `read-only`: No file writes, no network
   - `workspace-write`: Limited writes to workspace only
   - `danger-full-access`: Full access (use in Docker/isolated env)

Where does my data go?

Inputs/outputs you send through Code are handled under AI provider terms:

- **OpenAI**: See OpenAI Privacy Policy
- **Anthropic**: See Anthropic Privacy Policy
- **Google**: See Google AI Privacy Policy

Key points: - Code doesn't store or proxy your conversations - AI providers may use data per their terms (check policy) - For zero data retention: Use OpenAI ZDR (Zero Data Retention) orgs `toml disable_response_storage = true`

Can I use Code in an enterprise environment?

Yes, with considerations:

1. **API Key method** (recommended for enterprise):

   ```
   export OPENAI_API_KEY="sk-proj-ENTERPRISE_KEY"
   ```

2. **Zero Data Retention** (for sensitive code):

   ```
   # ~/.code/config.toml
   disable_response_storage = true
   ```

3. **Network restrictions**:

    - Requires outbound HTTPS to `api.openai.com`, `api.anthropic.com`, etc.

    - Configure proxy if needed:

      ```
      export HTTPS_PROXY="http://proxy.company.com:8080"
      ```

4. **Disable history** (for compliance):

    ```
    [history]
    persistence = "none"
    ```

5. **Read-only mode** for analysis:

    ```
    code --read-only "analyze codebase for vulnerabilities"
    ```

How do I prevent Code from editing my files?

Use read-only mode:

```
# CLI flag
code --read-only

# Or in config.toml
sandbox_mode = "read-only"

# Or mid-conversation
/approvals
# Select "Read Only" preset
```

Read-only mode: - ✅ Can read files - ✅ Can run commands (sandboxed) - ✅ Can answer questions - ❌ Cannot write files - ❌ Cannot modify code

---

Features and Capabilities

What can Code CLI do?

Core Capabilities: - ✅ Code generation (functions, modules, full features) - ✅ Code refactoring and optimization - ✅ Bug fixing and debugging - ✅ Test generation (unit, integration, E2E) - ✅ Documentation generation (README, API docs, comments) - ✅ Code review and analysis - ✅ Codebase Q&A and exploration - ✅ File operations (read, write, modify with approval) - ✅ Command execution (sandboxed)

Advanced Features: - ✅ Browser control (Chrome DevTools Protocol) - ✅ Multi-agent workflows (consensus, racing, collaboration) - ✅ MCP server integration (filesystem, databases, APIs) - ✅ Spec-Kit automation (fork feature: PRD → implementation pipeline) - ✅ Quality gates (multi-agent validation checkpoints) - ✅ Theming and customization - ✅ Reasoning control (adjust effort dynamically)

Does it work offline?

No, Code requires internet for AI models: - OpenAI API requires network - Claude and Gemini also require network - MCP servers may require network (depends on server)

Partial offline (experimental): - Use local models via Ollama (requires setup) - Configure local OpenAI-compatible endpoint - Quality depends on local model capabilities

---

Can Code CLI commit and push to Git?

Yes, with proper sandbox configuration:

```
# Allow git operations
sandbox_mode = "workspace_write"

[sandbox_workspace_write]
allow_git_writes = true  # Default: true
```

Example:

```
code "Create a commit for the changes we made with message 'Add user authentication'"

# Code will:
# 1. Stage changes (git add)
# 2. Create commit (git commit)
# 3. Show commit hash and message
```

Safety: - Code doesn’t push automatically (unless explicitly requested) - Always review commits before pushing - Use `/status` to check git state

---

Can Code generate entire applications from scratch?

Yes, but with considerations:

Best for: - ✅ Small to medium applications (todo apps, APIs, dashboards) - ✅ Well-defined requirements (clear specifications) - ✅ Standard tech stacks (React, Express, Flask, etc.)

Challenges: - ⚠ Large applications may hit context limits - ⚠ Requires iterative refinement - ⚠ Generated code needs review and testing

Recommended approach:

1. **Use Spec-Kit automation** for structured development:

   ```
   /speckit.new Build a REST API for a blog with user auth, posts, and comments
   /speckit.auto SPEC-ID
   ```

2. **Break into modules**:

   ```
   # Generate one module at a time
   code "Create the user authentication module with JWT"
   code "Create the blog post CRUD operations"
   code "Create the comments module with nested replies"
   ```

3. **Iterate and refine**:

```
code "Add input validation to the auth module"
code "Add rate limiting to the API endpoints"
code "Generate comprehensive tests for all modules"
```

Comparison with Other Tools

How is Code different from GitHub Copilot?

Feature	Code CLI	GitHub Copilot
Interface	Terminal (TUI)	IDE extension
Scope	Full file context, multi-file	Line/function suggestions
Autonomy	Can execute commands, make changes	Suggestions only
Conversation	Interactive chat	No conversation
Testing	Can run tests, fix failures	No test execution
Reasoning	Adjustable reasoning levels	Fixed
Multi-agent	Yes (consensus/racing)	No
Browser control	Yes (CDP)	No
Cost	Pay-per-use or ChatGPT sub	$10/month subscription

Use Code CLI for: - Large refactorings - Debugging complex issues - Test generation and execution - Documentation generation - Multi-file code generation

Use Copilot for: - Quick inline suggestions while coding - Auto-completion - IDE-integrated workflow

---

How is Code different from Cursor?

Feature	Code CLI	Cursor
Interface	Terminal	Full IDE (VS Code fork)
Autonomy	Full automation pipelines	Assisted coding
Spec-Kit	Yes (fork feature)	No
Multi-agent	Yes	Limited
Browser control	Yes	No
Setup	Lightweight (CLI only)	Full IDE installation
Terminal workflow	Native	Via IDE terminal
Cost	Flexible (API or	$20/month

subscription)

Use Code CLI for: - Terminal-first workflows - CI/CD automation - Spec-Kit PRD automation - Multi-agent consensus - Server/remote environments

Use Cursor for: - IDE-integrated development - GUI-first workflows - Inline editing with AI assistance

---

How does Spec-Kit compare to other AI dev tools?

Spec-Kit (theturtlecsz/code fork feature): - ✅ Full PRD → Plan → Tasks → Implementation → Validation → Audit pipeline - ✅ Multi-agent consensus at each stage - ✅ Configurable quality gates - ✅ Native cost optimization ($2.70 vs $11 for full pipeline) - ✅ Evidence collection and telemetry - ✅ Automated or manual step-through

Other tools: - Devin, Replit Agent: Cloud-based, less customizable - Aider: Terminal-based but single-agent, no pipeline - Smol Developer: Script-based, no consensus

Spec-Kit advantages: - Multi-agent validation (3-5 agents per critical stage) - Quality gates prevent bad implementations early - Evidence trail for debugging and auditing - Cost-optimized (cheap agents for simple stages, premium for critical)

Customization and Configuration

Can I customize the model for different tasks?

Yes, multiple ways:

Per-command (CLI flag):

```
code --model gpt-4o-mini "simple formatting task"
code --model o3 --config model_reasoning_effort=high "complex refactoring"
```

Profiles (named configurations):

```
# ~/.code/config.toml

[profiles.fast]
model = "gpt-4o-mini"
model_reasoning_effort = "low"
approval_policy = "never"

[profiles.premium]
model = "o3"
model_reasoning_effort = "high"
approval_policy = "on-request"

[profiles.automation]
model = "gpt-4o"
approval_policy = "never"
sandbox_mode = "read-only"
```

Use profiles:

```
code --profile fast "quick task"
code --profile premium "complex architecture decision"
code --profile automation "generate report"
```

Dynamic switching in TUI:

```
# In Code:
/model          # Interactive model selector
/reasoning high # Adjust reasoning mid-conversation
```

Can I extend Code with custom tools?

Yes, via MCP (Model Context Protocol) servers:

Example custom MCP server (Node.js):

```
// custom-mcp-server.js
import { Server } from "@modelcontextprotocol/sdk/server";

const server = new Server({
  name: "custom-tools",
  version: "1.0.0"
});

// Define custom tool
server.tool("deploy_to_production", async (params) => {
  // Your custom deployment logic
  return { success: true, message: "Deployed successfully" };
});

server.listen();
```

Configure in config.toml:

```
[mcp_servers.custom-tools]
command = "node"
args = ["/path/to/custom-mcp-server.js"]
```

Use in Code:

```
code "Deploy the application to production"
# Code will invoke your custom MCP tool
```

Can I use Code with custom OpenAI-compatible endpoints?

Yes, configure custom provider:

```
# ~/.code/config.toml

[model_providers.custom]
name = "Custom Provider"
base_url = "https://custom-api.example.com/v1"
env_key = "CUSTOM_API_KEY"
wire_api = "chat"  # or "responses"
```

```
# Use custom provider
model = "custom-model"
model_provider = "custom"
```

Examples:

Ollama (local models):

```
[model_providers.ollama]
name = "Ollama"
base_url = "http://localhost:11434/v1"

model = "mistral"
model_provider = "ollama"
```

Azure OpenAI:

```
[model_providers.azure]
name = "Azure OpenAI"
base_url = "https://YOUR_PROJECT.openai.azure.com/openai"
env_key = "AZURE_OPENAI_API_KEY"
query_params = { api-version = "2025-04-01-preview" }
wire_api = "responses"
```

Troubleshooting

Why is Code slow to start?

Common causes:

1. **Large history file**: `~/.code/history.jsonl` >100MB

   ```
   ls -lh ~/.code/history.jsonl
   # Truncate if large
   mv ~/.code/history.jsonl ~/.code/history.jsonl.backup
   ```

2. **Slow MCP servers**: Servers take time to initialize

   ```
   # Temporarily disable to test
   # Comment out in config.toml
   ```

3. **Network issues**: Slow connection to API providers

   ```
   # Test connectivity
   ping api.openai.com
   ```

Solutions → See troubleshooting.md

Why do I keep getting rate limit errors?

Causes: - Free tier API limits (3 req/min, 200 req/day for OpenAI) - Too many Spec-Kit multi-agent requests - Shared IP address (VPN, corporate network)

Solutions:

1. **Upgrade to paid tier**: Higher rate limits

2. **Use cheaper providers**: Gemini has higher free tier limits
3. **Wait and retry**: Rate limits reset after time period
4. **Reduce agent count**: Use single agent for simple tasks

   ```
   [quality_gates]
   tasks = ["code"]  # Single agent instead of multi-agent
   ```

Code generated incorrect/broken code. What do I do?

Immediate steps:

1. **Review the diff before approving**:
   - Always check changes carefully
   - Understand why changes were made
   - Look for unintended side effects
2. **Reject and provide feedback**:

   ```
   # In approval prompt, reject and respond:
   "The function should use async/await, not callbacks. Please refactor."
   ```

3. **Use higher reasoning for complex tasks**:

   ```
   code --model o3 --config model_reasoning_effort=high "complex task"
   ```

4. **Use multi-agent consensus** for critical changes:

   ```
   /code "refactor authentication system"
   # Multiple agents review and validate
   ```

Prevention:

- ✅ Write specific, clear prompts
- ✅ Provide examples of desired output
- ✅ Review diffs before approving
- ✅ Run tests after changes
- ✅ Use read-only mode first to see proposed changes

Where can I get more help?

Documentation: - Troubleshooting Guide - Comprehensive error solutions - Installation Guide - Setup issues - Configuration Docs - Advanced configuration

Community: - GitHub Issues: https://github.com/just-every/code/issues - GitHub Discussions: https://github.com/just-every/code/discussions

Fork-Specific (theturtlecsz/code): - Fork Issues: https://github.com/theturtlecsz/code/issues - Spec-Kit Docs: ../../spec-kit/README.md

---

Additional Questions?

Didn’t find your question?

1. Search GitHub Issues: https://github.com/just-every/code/issues
2. Check Discussions: https://github.com/just-every/code/discussions
3. Open a new issue with “Question:” prefix

Before asking: - ✅ Search existing issues/discussions - ✅ Review relevant documentation sections - ✅ Provide version info and error messages - ✅ Include steps to reproduce (if applicable)

Got your answer? → Continue exploring workflows or dive into advanced configuration!

First-Time Setup Guide

Complete setup guide for configuring Code CLI after installation.

Table of Contents

Overview

Time Required: 5-15 minutes

What You’ll Configure: - ✅ Authentication (ChatGPT or API key) - ✅ Basic config.toml settings - ✅ MCP servers (optional) - ✅ Multi-provider agents (optional)

Prerequisites: - Code CLI installed (installation guide) - OpenAI account (ChatGPT Plus/Pro/Team OR API key)

Step 1: Authentication

Code supports two authentication methods. Choose one:

Option A: Sign in with ChatGPT

Best for: ChatGPT Plus, Pro, or Team subscribers

Advantages: - ✅ No per-token billing - ✅ Access to models included in your plan - ✅ Easy setup - ✅ Credentials stored locally (not proxied)

Setup Steps:

1. **Launch Code**:

   ```
   code
   ```

2. **Select authentication method**:

   - You'll see a prompt: `Sign in with ChatGPT` or `Use API key`
   - Select `Sign in with ChatGPT`

3. **Complete browser flow**:

   - Code will start a local server on `localhost:1455`
   - Your browser will open automatically
   - Follow the ChatGPT sign-in flow
   - Authorize Code CLI

4. **Return to terminal**:

   - Once authorized, credentials are saved to `~/.code/auth.json`
   - Code will start automatically

Headless/Remote Setup (SSH, Docker, VPS):

If you're on a remote machine without a browser:

```
# From your LOCAL machine, create SSH tunnel:
ssh -L 1455:localhost:1455 user@remote-host

# Then, in that SSH session, run:
code

# Select "Sign in with ChatGPT"
# Open the printed URL in your LOCAL browser
# The tunnel will forward traffic to the remote server
```

Or authenticate locally and copy credentials:

```
# On LOCAL machine:
code login
# Complete authentication
# This creates ~/.code/auth.json

# Copy to REMOTE via scp:
ssh user@remote 'mkdir -p ~/.code'
scp ~/.code/auth.json user@remote:~/.code/auth.json

# Or one-liner:
ssh user@remote 'mkdir -p ~/.code && cat > ~/.code/auth.json' <
~/.code/auth.json
```

Option B: API Key

Best for: Usage-based billing, automation, CI/CD

Advantages: - ✅ Pay-as-you-go pricing - ✅ No subscription required - ✅ Works in CI/CD environments - ✅ Programmatic access

Setup Steps:

1. **Get API key**:

   - Go to https://platform.openai.com/api-keys
   - Create new secret key
   - Copy the key (starts with `sk-proj-...`)

2. **Set environment variable**:

   Temporary (current session only):

   ```
   export OPENAI_API_KEY="sk-proj-YOUR_KEY_HERE"
   code
   ```

   Permanent (add to shell profile):

   ```
   # For Bash (~/.bashrc)
   echo 'export OPENAI_API_KEY="sk-proj-YOUR_KEY_HERE"' >> ~/.bashrc
   source ~/.bashrc

   # For Zsh (~/.zshrc)
   echo 'export OPENAI_API_KEY="sk-proj-YOUR_KEY_HERE"' >> ~/.zshrc
   source ~/.zshrc
   ```

   Using ~/.code/.env (persistent, secure):

   ```
   mkdir -p ~/.code
   echo 'OPENAI_API_KEY=sk-proj-YOUR_KEY_HERE' > ~/.code/.env
   chmod 600 ~/.code/.env
   ```

3. **Launch Code**:

   ```
   code
   ```

   Code will automatically detect the API key and skip the login screen.

API Key Requirements: - Must have write access to the Responses API - Free tier has rate limits (3 req/min, 200 req/day) - Upgrade to paid tier for higher limits

---

Switching Authentication Methods

From API Key to ChatGPT:

```
# Unset API key
unset OPENAI_API_KEY

# Remove from .env if present
rm ~/.code/.env

# Run Code and select ChatGPT login
code login
```

From ChatGPT to API Key:

```
# Set API key
```

```
export OPENAI_API_KEY="sk-proj-YOUR_KEY"

# Configure preference (optional)
echo 'preferred_auth_method = "apikey"' >> ~/.code/config.toml
```

Force specific method in config.toml:

```
# Always use API key (even if ChatGPT auth exists)
preferred_auth_method = "apikey"

# Or always use ChatGPT (default)
preferred_auth_method = "chatgpt"
```

Step 2: Basic Configuration

Create and customize `~/.code/config.toml`:

Create Configuration File

```
# Create config directory
mkdir -p ~/.code

# Create config file
touch ~/.code/config.toml
```

Minimal Configuration

Recommended starter config:

```
# ~/.code/config.toml

# Model Settings
model = "gpt-5"
model_provider = "openai"

# Behavior
approval_policy = "on_request"  # Model decides when to ask
model_reasoning_effort = "medium"  # low | medium | high
sandbox_mode = "workspace_write"  # read-only | workspace-write |
danger-full-access

# UI Preferences
[tui]
notifications = true  # Desktop notifications for approvals
```

Configuration Options

Model Settings:

```
model = "gpt-5"              # Default model
model_reasoning_effort = "medium"  # Reasoning depth
model_reasoning_summary = "auto"   # auto | concise | detailed |
none
model_verbosity = "medium"         # low | medium | high
```

Approval Policy:

```
# When should Code ask for permission to run commands?
approval_policy = "untrusted"   # Ask for untrusted commands only
# approval_policy = "on-failure" # Ask when commands fail
# approval_policy = "on-request" # Model decides (recommended)
# approval_policy = "never"      # Never ask (full auto, risky)
```

Sandbox Mode:

```
# What can Code modify?
sandbox_mode = "read-only"        # No writes, no network
# sandbox_mode = "workspace_write" # Write to workspace, no network
(recommended)
# sandbox_mode = "danger-full-access" # Full access (use in
Docker/isolated env)

# Fine-tune workspace-write behavior
[sandbox_workspace_write]
allow_git_writes = true           # Allow writing to .git/ (default:
true)
network_access = false            # Enable network (default: false)
writable_roots = ["/tmp"]         # Additional writable paths
```

History and Privacy:

```
# Message history
[history]
persistence = "save-all"  # save-all | none

# Zero Data Retention (for ZDR orgs)
disable_response_storage = false  # Set to true for ZDR
```

Step 3: MCP Server Setup (Optional)

What are MCP Servers? Model Context Protocol (MCP) servers extend Code's capabilities with custom tools: - File operations - Database connections - API integrations - Custom tools

Install MCP Servers

Filesystem Server (file operations):

```
npm install -g @modelcontextprotocol/server-filesystem
```

Git Status Server (git integration):

```
npm install -g @modelcontextprotocol/server-git-status
```

Local Memory Server (persistent memory, recommended for this fork):

```
npm install -g @modelcontextprotocol/server-local-memory
```

Configure MCP Servers in config.toml

Add MCP server configurations to `~/.code/config.toml`:

```
# MCP Servers Configuration
```

```
[mcp_servers.filesystem]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-filesystem",
"/path/to/your/project"]
startup_timeout_sec = 10
tool_timeout_sec = 60

[mcp_servers.local-memory]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-local-memory"]
startup_timeout_sec = 10
tool_timeout_sec = 30

[mcp_servers.git-status]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-git-status"]
startup_timeout_sec = 10
```

Notes: - Replace `/path/to/your/project` with your actual project path - `startup_timeout_sec`: How long to wait for server to start (default: 10) - `tool_timeout_sec`: Max time for each tool call (default: 60)

Verify MCP Servers

```
# List configured MCP servers
code mcp list

# Get details for specific server
code mcp get filesystem

# Test server health
code mcp test filesystem
```

Step 4: Multi-Provider Setup (Optional)

Why Multi-Provider? - Use multiple AI models for consensus and quality gates - Cost optimization (cheap models for simple tasks, premium for critical) - Enhanced reliability (fallback if one provider fails)

Install Provider CLI Tools

```
# Anthropic Claude
npm install -g @anthropic-ai/claude-code

# Google Gemini
npm install -g @google/gemini-cli

# Verify installations
claude "test"
gemini -i "test"
```

Configure API Keys

```
# Anthropic Claude
export ANTHROPIC_API_KEY="sk-ant-api03-YOUR_KEY"
```

```
# Get key: https://console.anthropic.com/settings/keys

# Google Gemini
export GOOGLE_API_KEY="AIza_YOUR_KEY"
# Get key: https://ai.google.dev/

# Add to shell profile for persistence (~/.bashrc or ~/.zshrc)
echo 'export ANTHROPIC_API_KEY="sk-ant-..."' >> ~/.bashrc
echo 'export GOOGLE_API_KEY="AIza..."' >> ~/.bashrc
source ~/.bashrc
```

Configure Agents in config.toml

```
# Multi-Agent Configuration

[[agents]]
name = "gemini-flash"
canonical_name = "gemini"
command = "gemini"
enabled = true

[[agents]]
name = "claude-sonnet"
canonical_name = "claude"
command = "claude"
enabled = true

[[agents]]
name = "gpt-5"
canonical_name = "code"
command = "code"
enabled = true
```

Configure Quality Gates (Spec-Kit Framework)

```
[quality_gates]
# Simple stages: cheap agents
tasks = ["gemini"]  # Gemini Flash: $0.075/1M tokens (12x cheaper)

# Complex stages: multi-agent consensus
plan = ["gemini", "claude", "code"]
validate = ["gemini", "claude", "code"]

# Critical stages: premium agents
audit = ["gemini-pro", "claude-opus", "gpt-5"]
unlock = ["gemini-pro", "claude-opus", "gpt-5"]
```

Cost Comparison: - **Cheap strategy** (Gemini only): ~$0.10/pipeline - **Balanced strategy** (above config): ~$2.70/pipeline - **Premium strategy** (all top models): ~$11/pipeline

---

Step 5: Verify Setup

Test Authentication

```
# Check which auth method is active
```

```
code /status

# Should show:
# - Model: gpt-5
# - Auth: ChatGPT (or API key)
# - Provider: openai
```

Test Basic Functionality

```
# Interactive mode
code

# Type in chat: "What files are in this directory?"
# Code should list files successfully
```

Test MCP Servers (if configured)

```
# In Code chat, ask:
"Use the filesystem MCP server to list files in /home/user"

# Code should invoke the MCP tool and list files
```

Test Multi-Provider (if configured)

```
# Run multi-agent command (requires all providers configured)
code "/plan 'Add user authentication'"

# Should see consensus from multiple models
```

Configuration File Reference

File Locations

File	Purpose	Notes
`~/.code/config.toml`	Main configuration	**Primary** (reads legacy `~/.codex/config.toml`)
`~/.code/auth.json`	Authentication credentials	Auto-generated, read-only (0600 permissions)
`~/.code/.env`	Environment variables	Optional, for API keys
`~/.code/history.jsonl`	Message history	Auto-generated, can disable via config

Backwards Compatibility: - Code reads from both `~/.code/` (primary) and `~/.codex/` (legacy) - Code only writes to `~/.code/` - If migrating from Codex, copy `~/.codex/config.toml` to `~/.code/config.toml`

Configuration Precedence

Order of precedence (highest to lowest):

1. **Command-line flags**: `--model gpt-5`, `--config key=value`
2. **Profile** (via `--profile` or `profile = "name"` in config)
3. **config.toml entries**: Direct settings
4. **Environment variables**: `OPENAI_API_KEY`, `CODEX_HOME`, etc.
5. **Default values**: Built-in Code CLI defaults

Example:

```
# All of these work, in order of precedence:
code --model o3                           # 1. CLI flag (highest)
code --profile premium                    # 2. Profile
export OPENAI_API_KEY="..."               # 3. Environment variable
# config.toml: model = "gpt-5"           # 4. Config file
# (defaults to gpt-5-codex if none set)  # 5. Default (lowest)
```

Profiles

Create named profiles for different workflows:

```
# Default settings
model = "gpt-5"
approval_policy = "on-request"

# Profile for premium reasoning
[profiles.premium]
model = "o3"
model_reasoning_effort = "high"
model_reasoning_summary = "detailed"
approval_policy = "never"

# Profile for fast iteration
[profiles.fast]
model = "gpt-4o-mini"
model_reasoning_effort = "low"
approval_policy = "never"

# Profile for automation/CI
[profiles.ci]
model = "gpt-4o"
approval_policy = "never"
sandbox_mode = "read-only"
disable_response_storage = true
```

Use profiles:

```
code --profile premium "complex refactoring task"
code --profile fast "simple code formatting"
code --profile ci "run tests and generate report"
```

Troubleshooting Setup

Authentication Issues

Error: `Failed to authenticate`

Solution: Check credentials

```
# For ChatGPT: Delete and re-authenticate
```

```
rm ~/.code/auth.json
code login

# For API Key: Verify key is correct
echo $OPENAI_API_KEY
# Should output: sk-proj-...
```

Error: `401 Unauthorized` with API key

Cause: Invalid API key or insufficient permissions.

Solution:

```
# Verify API key at https://platform.openai.com/api-keys
# Ensure key has access to Responses API
# Check account verification status
```

Error: ChatGPT login fails on remote/headless server

Solution: Use SSH tunnel or copy credentials (see Option A: Sign in with ChatGPT)

Configuration Issues

Error: `config.toml: unknown field 'xyz'`

Cause: Typo or unsupported config option.

Solution:

```
# Check config syntax
code --config-check

# Refer to docs/config.md for valid options
# Common typos:
# - mcpServers → mcp_servers
# - modelProvider → model_provider
```

Error: Config changes not taking effect

Cause: Config file not in correct location or profile override.

Solution:

```
# Verify config file location
ls -la ~/.code/config.toml

# Check which config is loaded
code --print-config

# Disable profile override temporarily
code --profile=none
```

MCP Server Issues

Error: `MCP server 'filesystem' failed to start`

Cause: Server not installed or command incorrect.

Solution:

```
# Verify server is installed
npm list -g @modelcontextprotocol/server-filesystem

# If not installed:
npm install -g @modelcontextprotocol/server-filesystem

# Test server manually
npx @modelcontextprotocol/server-filesystem /path/to/project
```

Error: MCP server times out on startup

Cause: Server takes longer than `startup_timeout_sec` to start.

Solution: Increase timeout in config.toml

```
[mcp_servers.slow-server]
command = "npx"
args = ["-y", "slow-mcp-server"]
startup_timeout_sec = 30  # Increase from default 10
```

Multi-Provider Issues

Error: `Command 'claude' not found`

Solution: Install CLI tools

```
npm install -g @anthropic-ai/claude-code @google/gemini-cli

# Verify
which claude
which gemini
```

Error: `API key 'ANTHROPIC_API_KEY' not set`

Solution: Set environment variable

```
export ANTHROPIC_API_KEY="sk-ant-api03-YOUR_KEY"

# Add to shell profile for persistence
echo 'export ANTHROPIC_API_KEY="sk-ant-..."' >> ~/.bashrc
source ~/.bashrc
```

Error: Rate limit exceeded

Cause: Too many requests to API provider.

Solutions: 1. **Wait**: Rate limits reset after time period 2. **Upgrade plan**: Higher tier = higher limits 3. **Use cheaper models**: Gemini Flash has higher limits 4. **Reduce agent count**: Use single agent for simple tasks

Rate Limits (typical free tiers): - OpenAI: 3 requests/min, 200 requests/day - Anthropic: 5 requests/min - Google: 15 requests/min

---

Next Steps

Your setup is complete! Now:

1. **Quick Start Tutorial** → quick-start.md
   - Run your first command
   - Learn the TUI interface
   - Try example prompts
2. **Learn Common Workflows** → workflows.md
   - Spec-kit automation
   - Code refactoring
   - Multi-agent collaboration
3. **Advanced Configuration** → ../config.md
   - Custom model providers
   - Project-specific hooks
   - Validation harnesses

Setup Complete! 🎉 → Continue to Quick Start

Installation Guide

This guide covers all methods for installing the Code CLI tool on your system.

Table of Contents

System Requirements

Before installing Code, ensure your system meets these minimum requirements:

Requirement	Details
Operating System	macOS 12+, Ubuntu 20.04+/Debian 10+, or Windows 11 via WSL2
Node.js	Version 22+ (for npm installation)
RAM	4 GB minimum (8 GB recommended)

Disk Space	500 MB minimum
Git	2.23+ (optional, recommended for built-in PR helpers)

Windows Users: Direct Windows installation is not officially supported. Use Windows Subsystem for Linux (WSL2) for the best experience.

Quick Install (Recommended)

The fastest way to get started is using npm:

```
# Install globally
npm install -g @just-every/code

# Run Code
code
```

Note: If another tool provides a `code` command (e.g., VS Code), use `coder` instead to avoid conflicts.

Installation Methods

Method 1: NPM (Recommended)

Best for: Most users, especially those familiar with Node.js ecosystems.

Prerequisites: - Node.js 22+ installed - npm or pnpm package manager

Installation Steps:

```
# Install using npm
npm install -g @just-every/code

# Or using pnpm (faster)
pnpm add -g @just-every/code
```

Verify Installation:

```
# Check version
code --version

# If 'code' conflicts with VS Code, use:
coder --version
```

Command Aliases: - `code` - Primary command - `coder` - Alternative command (avoids conflicts with VS Code)

Both commands are functionally identical.

Method 2: One-Time Execution (No Install)

Best for: Quick testing, CI/CD pipelines, temporary use.

No installation required - uses `npx` to download and run:

```
# Run directly without installing
npx -y @just-every/code

# With a prompt
npx -y @just-every/code "explain this codebase"
```

Advantages: - ✅ No global installation - ✅ Always uses latest version - ✅ Perfect for CI/CD - ✅ No conflicts with existing tools

Disadvantages: - ❌ Slower startup (downloads each time) - ❌ Requires internet connection

---

Method 3: Homebrew (macOS/Linux)

Best for: macOS and Linux users who prefer Homebrew package management.

Prerequisites: - Homebrew installed

Installation Steps:

```
# Add the tap (if not already added)
brew tap just-every/code

# Install Code
brew install code-cli

# Run Code
code
```

Update via Homebrew:

```
brew upgrade code-cli
```

Uninstall:

```
brew uninstall code-cli
```

Method 4: Build from Source

Best for: Contributors, advanced users, custom builds, or testing unreleased features.

Prerequisites: - Git 2.23+ - Rust toolchain (will be installed automatically) - Node.js 22+ (for TypeScript CLI wrapper)

Installation Steps:

Step 1: Clone Repository

```
# Clone from GitHub (upstream community fork)
git clone https://github.com/just-every/code.git
cd code
```

For fork contributors (theturtlecsz/code):

```
# Clone the fork
git clone https://github.com/theturtlecsz/code.git
cd code

# Add upstream remote
git remote add upstream https://github.com/just-every/code.git
```

Step 2: Install Rust Toolchain

```
# Install Rust (if not already installed)
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s --
-y

# Load Rust environment
source "$HOME/.cargo/env"

# Install required components
rustup component add rustfmt clippy
```

Step 3: Setup Git Hooks (Contributors Only)

```
# Required for contributors to theturtlecsz/code fork
bash scripts/setup-hooks.sh
```

This installs pre-commit hooks that ensure: - Code formatting (`cargo fmt`) - Linting (`cargo clippy`) - Tests compile successfully - Documentation structure is valid

Step 4: Build Code

Fast Build (recommended for development):

```
# Build with fast profile (optimized for iteration speed)
./build-fast.sh

# Binary location
./codex-rs/target/dev-fast/code
```

Release Build (optimized for performance):

```
# Navigate to Rust workspace
cd codex-rs

# Build release binaries
cargo build --release --bin code --bin code-tui --bin code-exec

# Binary location
./target/release/code
```

Quick Build (Code CLI only):

```
cd codex-rs
cargo build --release --bin code
```

Step 5: Run Locally

```
# From fast build
./codex-rs/target/dev-fast/code

# From release build
```

```
./codex-rs/target/release/code

# With a prompt
./codex-rs/target/release/code "explain this codebase to me"
```

Step 6: Install Globally (Optional)

```
# Install the binary to ~/.cargo/bin (in your PATH)
cd codex-rs
cargo install --path cli --bin code

# Now you can run 'code' from anywhere
code --version
```

Verify Build Quality

```
cd codex-rs

# Format code
cargo fmt --all

# Run linter
cargo clippy --workspace --all-targets --all-features -- -D warnings

# Build all binaries
cargo build --workspace --all-features

# Run test suite
cargo test
```

Verification

After installation, verify Code is working correctly:

Basic Verification

```
# Check version
code --version
# Expected output: code 0.0.0 (or current version)

# Display help
code --help

# Generate shell completions (optional)
code completion bash   # for Bash
code completion zsh    # for Zsh
code completion fish   # for Fish
```

Test Interactive Mode

```
# Start interactive TUI
code
```

You should see: - Authentication prompt (if first run) - Chat interface with composer - Status bar showing model and configuration

Exit: Press `Ctrl+C` or type `/exit`

Test Non-Interactive Mode

```
# Run a simple prompt (requires authentication)
code "What are the files in this directory?"
```

Next Steps

After successful installation:

1. **First-Time Setup** → See first-time-setup.md
   - Configure authentication (ChatGPT or API key)
   - Set up config.toml
   - Configure MCP servers (optional)
2. **Quick Start Tutorial** → See quick-start.md
   - Run your first command
   - Understand the TUI interface
   - Try example workflows
3. **Learn Common Workflows** → See workflows.md
   - Spec-kit automation
   - Code refactoring
   - Testing and validation

Troubleshooting Installation

NPM Installation Issues

Error: `EACCES: permission denied`

Solution 1: Use `--prefix` to install to user directory:

```
npm install -g --prefix ~/.npm-global @just-every/code
export PATH=~/.npm-global/bin:$PATH
```

Solution 2: Fix npm permissions:

```
mkdir -p ~/.npm-global
npm config set prefix '~/.npm-global'
echo 'export PATH=~/.npm-global/bin:$PATH' >> ~/.bashrc
source ~/.bashrc
npm install -g @just-every/code
```

Error: `npm ERR! code E404` or `npm ERR! 404 Not Found`

Cause: Package name incorrect or npm registry issue.

Solution:

```
# Verify package name
npm info @just-every/code

# Clear npm cache
npm cache clean --force

# Try again
npm install -g @just-every/code
```

Build from Source Issues

Error: `rustc: command not found`

Cause: Rust toolchain not installed or not in PATH.

Solution:

```
# Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s --
-y

# Reload shell or source manually
source "$HOME/.cargo/env"

# Verify
rustc --version
cargo --version
```

Error: `cargo build` fails with “linking with `cc` failed”

Cause: Missing system dependencies (especially on Linux).

Solution (Ubuntu/Debian):

```
sudo apt-get update
sudo apt-get install -y build-essential pkg-config libssl-dev
```

Solution (macOS):

```
# Install Xcode Command Line Tools
xcode-select --install
```

Error: `error: linker 'cc' not found` on Alpine Linux

Cause: Alpine uses musl libc, requires additional setup.

Solution:

```
# Install musl-dev
apk add musl-dev

# Or use rustup to add musl target
rustup target add x86_64-unknown-linux-musl
cargo build --target x86_64-unknown-linux-musl
```

Error: Pre-commit hook blocks commit with `cargo clippy` warnings

Cause: Code quality checks failed.

Solution:

```
# Fix formatting
cargo fmt --all

# Fix clippy warnings
cargo clippy --workspace --all-targets --all-features --fix --allow-
dirty
```

```
# Or temporarily skip hooks (NOT recommended for regular commits)
PRECOMMIT_FAST_TEST=0 git commit -m "your message"
```

Windows (WSL2) Issues

Error: `code: command not found` after installation in WSL2

Cause: PATH not updated or npm global bin not in PATH.

Solution:

```
# Find npm global bin path
npm config get prefix

# Add to PATH (add to ~/.bashrc or ~/.zshrc)
export PATH="$(npm config get prefix)/bin:$PATH"

# Reload shell
source ~/.bashrc
```

Error: Git operations fail with "permission denied" in WSL2

Cause: Windows file permissions issue.

Solution:

```
# Clone repos into WSL filesystem (not /mnt/c/)
cd ~
git clone https://github.com/just-every/code.git
```

Command Conflicts

Error: `code` opens VS Code instead of Code CLI

Cause: VS Code's `code` command takes precedence in PATH.

Solution 1: Use `coder` alias

```
coder --version
coder "your prompt"
```

Solution 2: Create shell alias

```
# Add to ~/.bashrc or ~/.zshrc
alias codex-cli='code'

# Or point directly to binary
alias codex-cli='/path/to/code'
```

Solution 3: Adjust PATH order (advanced)

```
# Find Code CLI location
which -a code

# Add to beginning of PATH in ~/.bashrc
export PATH="/path/to/code/bin:$PATH"
```

Verification Failures

Error: `code --version` shows old version after update

Cause: Multiple installations or cached binary.

Solution:

```
# Find all 'code' binaries
which -a code

# Clear npm cache
npm cache clean --force

# Reinstall
npm uninstall -g @just-every/code
npm install -g @just-every/code

# Verify
code --version
```

Additional Resources

- **Official Documentation**: docs/README.md
- **Configuration Guide**: config.md
- **Troubleshooting**: troubleshooting.md
- **GitHub Repository**: https://github.com/just-every/code
- **Fork Repository** (enhanced features): https://github.com/theturtlecsz/code

---

Installation Complete! → Continue to First-Time Setup

Quick Start Tutorial

Get productive with Code CLI in 5 minutes.

Table of Contents

Prerequisites

Before starting, ensure: - ✅ Code CLI installed (installation guide) - ✅ Authentication configured (setup guide) - ✅ You're in a directory with code files (for testing)

Time Required: 5 minutes

Your First Command (1 minute)

Interactive Mode

Launch Code and ask a simple question:

```
# Start interactive TUI
code
```

In the chat composer, type:

```
What files are in this directory?
```

Press Enter and watch Code: 1. Analyze your request 2. Execute appropriate tool (file listing) 3. Return results in the chat

Exit: Press `Ctrl+C` or type `/exit`

Non-Interactive Mode

Run a command directly without opening the TUI:

```
# Execute a single task
code "list all Python files in this directory"
```

Code will: - Process your request - Show output in the terminal - Exit automatically when done

---

With Initial Prompt

Start the TUI with a pre-filled prompt:

```
# Launch with initial prompt (doesn't auto-execute)
code "explain the architecture of this codebase"
```

This opens the TUI with your prompt ready to send (press Enter to submit).

Understanding the TUI (2 minutes)

TUI Layout

When you launch `code`, you’ll see:

```
┌──────────────────────────────────────────────┐
│  📝  Chat History                              │  ← Conversation
history
│                                              │
│  User: What files are here?                  │
│  Assistant: I found 3 Python files:          │
│  - main.py                                   │
│  - utils.py                                      │
│  - tests.py                                      │
│                                                  │
├──────────────────────────────────────────────────┤
│  ✏  Composer (Type here)                        │  ← Your input area
│  > |                                             │
│                                                  │
├──────────────────────────────────────────────────┤
│  ℹ  Status: Model: gpt-5 | Auth: ChatGPT  │  ← Status bar
└──────────────────────────────────────────────────┘
```

Key Components

1. Chat History (top): - Shows conversation with the AI - Model responses with reasoning (if enabled) - Tool executions and results - Scroll with arrow keys or mouse

2. Composer (middle): - Type your prompts here - Multi-line input supported (Shift+Enter for new line) - Submit with Enter

3. Status Bar (bottom): - Current model - Authentication method - Workspace directory - Sandbox mode

---

Essential Keyboard Shortcuts

Shortcut	Action
Enter	Send message (submit prompt)
Shift+Enter	New line in composer (multi-line input)
Ctrl+C	Exit Code
Esc	Clear composer (or cancel current operation)
Esc Esc	Edit previous message (backtrack)
Ctrl+V / Cmd+V	Paste image (from clipboard)
@	Fuzzy file search (type @ then filename)
Up/Down	Scroll chat history
Tab	Auto-complete (in file search)

Special Commands (Slash Commands)

Type `/` followed by a command name:

Command	Purpose	Example
/new	Start new conversation	`/new`
/model	Switch model or reasoning level	`/model`
/reasoning	Adjust reasoning effort	`/reasoning high`
/themes	Change TUI theme	`/themes`
/status	Show current configuration	`/status`
/exit	Exit Code	`/exit`
/help	Show help	`/help`

Spec-Kit Commands (multi-agent automation): - `/speckit.new` - Create new specification - `/speckit.auto` - Full automation pipeline - `/speckit.plan` - Generate work breakdown - `/speckit.implement` - Code generation - See workflows.md for details

---

Example Workflows (2 minutes)

Example 1: Code Explanation

Goal: Understand a code file

```
code
```

In chat:

```
Explain what src/main.py does and summarize its key functions.
```

Expected Output: - File overview - Function summaries - Dependencies identified - Potential improvements

---

Example 2: Code Refactoring

Goal: Refactor code for better readability

Prompt:

```
Refactor the calculate_total() function in utils.py to use more
descriptive variable names and add docstrings.
```

Code will: 1. Read `utils.py` 2. Identify the function 3. Propose refactored version 4. Show diff (before/after) 5. Ask for approval (unless in auto mode) 6. Apply changes if approved

Review the diff:

```diff
- def calculate_total(items):
-     t = 0
-     for i in items:
-         t += i.price
-     return t
+ def calculate_total(items):
+     """Calculate the total price of all items.
+
+     Args:
+         items: List of items with price attribute
+
+     Returns:
+         Total sum of all item prices
+     """
+     total_price = 0
+     for item in items:
+         total_price += item.price
+     return total_price
```

Example 3: Writing Tests

Goal: Generate unit tests for a function

Prompt:

```
Write comprehensive unit tests for the validate_email() function in
validators.py. Include edge cases and error conditions.
```

Code will: 1. Analyze the function 2. Generate test file (e.g., `test_validators.py`) 3. Include test cases: - Valid emails - Invalid formats - Edge cases (empty, null, special chars) - Boundary conditions 4. Ask for approval 5. Run tests to verify they pass

---

Example 4: Bug Investigation

Goal: Find and fix a bug

Prompt:

```
Users are reporting that the login function returns "500 Internal
Server Error". Investigate the issue in auth.py and propose a fix.
```

Code will: 1. Read `auth.py` 2. Identify potential issues (e.g., unhandled exceptions) 3. Propose fix with explanation 4. Show before/after diff 5. Suggest additional error handling

---

Example 5: Documentation

Goal: Generate documentation for a module

Prompt:

```
Generate a comprehensive README.md for the database/ module,
including setup instructions, API reference, and usage examples.
```

Code will: 1. Analyze all files in `database/` 2. Extract function signatures and purposes 3. Generate structured README: - Overview - Installation - API reference - Examples - Troubleshooting 4. Ask for approval 5. Create `database/README.md`

Essential Commands

CLI Usage

Basic:

```
code                           # Interactive TUI
code "prompt"                  # TUI with initial prompt
code exec "prompt"             # Non-interactive execution
```

With Options:

```
code --model o3                # Use specific model
code --read-only "explain"     # Read-only mode (no writes)
code --no-approval "task"      # Skip approval prompts (auto mode)
code --debug                   # Enable debug logging
code --sandbox workspace-write # Set sandbox mode
code --cd /path/to/project     # Change working directory
```

Configuration:

```
code --config model=o3                    # Override config value
code --config approval_policy=never       # Full auto mode
code --profile premium                    # Use named profile
code --version                            # Show version
code --help                               # Show help
```

In-TUI Slash Commands

Conversation:

```
/new                           # Start new conversation
/exit                          # Exit Code
```

Model & Settings:

```
/model                         # Switch model
/reasoning low                 # Set reasoning effort
(low/medium/high)
/themes                        # Change theme
/status                        # Show configuration
```

Browser Integration (if enabled):

```
/chrome                        # Connect to external Chrome
/chrome 9222                   # Connect to Chrome on port 9222
/browser https://example.com   # Open URL in internal browser
```

Multi-Agent Commands (requires multi-provider setup):

```
/plan "task"                   # Multi-agent planning (Claude,
Gemini, GPT-5)
/solve "problem"               # Race multiple models (fastest wins)
/code "feature"                # Multi-agent code generation
```

Spec-Kit Automation (fork feature):

```
/speckit.new "description"     # Create new spec
/speckit.auto SPEC-ID          # Run full automation pipeline
/speckit.plan SPEC-ID          # Generate plan
/speckit.implement SPEC-ID     # Generate code
/speckit.validate SPEC-ID      # Run tests
```

File Operations

Attach files/images:

```
# Via CLI
code --image screenshot.png "explain this error"
code -i img1.png,img2.png "compare these diagrams"

# Via TUI
# Ctrl+V / Cmd+V to paste image from clipboard
```

File search in composer:

```
# Type @ to trigger fuzzy file search
@main.py       # Searches for main.py
@test          # Searches for files matching "test"
# Use Up/Down to select, Tab/Enter to insert
```

Next Steps

Now that you've completed the quick start:

1. **Learn Common Workflows** → workflows.md
   - Spec-kit automation (multi-agent PRD workflows)
   - Code review and refactoring
   - Test generation and validation
   - CI/CD integration
2. **Explore Configuration** → first-time-setup.md
   - Custom model providers
   - MCP servers (extend functionality)
   - Multi-provider setup
   - Quality gates
3. **Read FAQ** → faq.md
   - Common questions
   - Comparison with other tools
   - Cost management
   - Privacy and data handling
4. **Troubleshooting** → troubleshooting.md
   - Installation errors
   - Authentication issues
   - Performance problems
   - Common mistakes

Quick Reference Card

Most Common Tasks

Task	Command
Explain code	`code "explain main.py"`
Refactor function	`code "refactor calculate() in utils.py"`
Write tests	`code "write tests for auth.py"`
Fix bug	`code "fix the login bug in auth.py"`
Generate docs	`code "generate README for api/ module"`
Code review	`code "review the changes in src/"`
Add feature	`code "add user authentication"`

Keyboard Shortcuts

Shortcut	Action
Enter	Send message
Shift+Enter	New line
Ctrl+C	Exit
Esc Esc	Edit previous message

@	File search

Essential Slash Commands

Command	Purpose
/new	New conversation
/model	Switch model
/reasoning high	Increase reasoning
/status	Show config
/exit	Exit Code

Ready to dive deeper? → Continue to Common Workflows

Troubleshooting Guide

Comprehensive error resolution guide for Code CLI.

Table of Contents

Installation Errors

Error: `npm: command not found`

Cause: Node.js/npm not installed

Solution:

```
# Install Node.js (includes npm)
# Visit https://nodejs.org/ or use package manager:

# macOS (Homebrew)
brew install node

# Ubuntu/Debian
curl -fsSL https://deb.nodesource.com/setup_22.x | sudo -E bash -
sudo apt-get install -y nodejs
```

```
# Verify
node --version
npm --version
```

Error: `EACCES: permission denied` (npm install)

Cause: Insufficient permissions to install global npm packages

Solution 1: Install to user directory

```
# Create npm global directory
mkdir -p ~/.npm-global

# Configure npm to use it
npm config set prefix '~/.npm-global'

# Add to PATH
echo 'export PATH=~/.npm-global/bin:$PATH' >> ~/.bashrc
source ~/.bashrc

# Install Code
npm install -g @just-every/code
```

Solution 2: Fix npm permissions

```
# Change npm directory ownership
sudo chown -R $(whoami) ~/.npm
sudo chown -R $(whoami) /usr/local/lib/node_modules

# Retry installation
npm install -g @just-every/code
```

Solution 3: Use `sudo` (not recommended)

```
sudo npm install -g @just-every/code
```

Error: `cargo: command not found` (build from source)

Cause: Rust toolchain not installed

Solution:

```
# Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s --
-y

# Reload environment
source "$HOME/.cargo/env"

# Install required components
rustup component add rustfmt clippy

# Verify
cargo --version
rustc --version
```

Error: `linking with cc failed` (Rust build)

Cause: Missing C compiler or system libraries

Solution (Ubuntu/Debian):

```
sudo apt-get update
sudo apt-get install -y build-essential pkg-config libssl-dev
```

Solution (macOS):

```
xcode-select --install
```

Solution (Alpine Linux):

```
apk add build-base openssl-dev
```

Error: `code: command not found` after installation

Cause: npm global bin directory not in PATH

Solution:

```
# Find npm global bin path
npm config get prefix

# Expected output: /home/user/.npm-global (or similar)

# Add to PATH in ~/.bashrc or ~/.zshrc
echo 'export PATH="$(npm config get prefix)/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc

# Verify
code --version
```

Alternative: Use full path

```
# Find code binary location
npm list -g @just-every/code | grep code

# Run with full path
/path/to/code --version
```

Authentication Issues

Error: `Failed to authenticate` (ChatGPT login)

Cause: Browser flow failed or credentials not saved

Solution:

```
# Delete existing auth and retry
rm ~/.code/auth.json
code login

# Follow browser prompts carefully
# Ensure you're logged into ChatGPT
# Authorize Code CLI when prompted
```

For headless/remote servers:

```
# Use SSH tunnel (from local machine)
ssh -L 1455:localhost:1455 user@remote-host

# In SSH session, run:
code

# Open localhost:1455 in LOCAL browser
```

Error: `401 Unauthorized` (API key)

Cause: Invalid API key or insufficient permissions

Solution:

```
# Verify API key format (should start with sk-proj-)
echo $OPENAI_API_KEY

# Check key at https://platform.openai.com/api-keys
# Ensure key has access to Responses API
# Check account verification status

# Set correct key
export OPENAI_API_KEY="sk-proj-CORRECT_KEY_HERE"

# Or add to ~/.code/.env
mkdir -p ~/.code
echo 'OPENAI_API_KEY=sk-proj-YOUR_KEY' > ~/.code/.env
chmod 600 ~/.code/.env
```

Error: `403 Forbidden` (ChatGPT)

Cause: Account not eligible or subscription expired

Solution:

```
# Verify ChatGPT subscription status at https://chat.openai.com/

# Check account type:
# - ChatGPT Plus: ✅ Supported
# - ChatGPT Pro: ✅ Supported
# - ChatGPT Team: ✅ Supported
# - Free tier: ❌ Not supported for CLI

# Switch to API key if needed
export OPENAI_API_KEY="sk-proj-YOUR_KEY"
```

Error: `Rate limit exceeded`

Cause: Too many requests to API provider

Solution:

Wait for reset:

```
# Rate limits reset after time period
# Free tier: ~1 hour
# Paid tier: ~1 minute
```

Upgrade plan:

```
# Visit https://platform.openai.com/settings/organization/billing
# Upgrade to paid tier for higher limits
```

Use retry logic (automatic in Code):

```
# Code automatically retries with exponential backoff
# Wait for retry to complete
```

Rate Limits by Provider:

Provider	Free Tier	Paid Tier
OpenAI	3 req/min, 200 req/day	60-90 req/min
Anthropic	5 req/min	50 req/min
Google Gemini	15 req/min	60 req/min

Error: `OPENAI_API_KEY not set` (but it is set)

Cause: Environment variable not loaded or scoping issue

Solution:

```
# Check if variable is actually set
echo $OPENAI_API_KEY

# If empty, set it
export OPENAI_API_KEY="sk-proj-YOUR_KEY"

# Permanently set in shell profile
echo 'export OPENAI_API_KEY="sk-proj-YOUR_KEY"' >> ~/.bashrc
source ~/.bashrc

# Or use ~/.code/.env
mkdir -p ~/.code
echo 'OPENAI_API_KEY=sk-proj-YOUR_KEY' > ~/.code/.env
chmod 600 ~/.code/.env

# Verify Code sees it
code --print-config | grep OPENAI_API_KEY
```

MCP Connection Problems

Error: `MCP server 'filesystem' failed to start`

Cause: MCP server not installed or command incorrect

Solution:

```
# Check if server is installed globally
npm list -g @modelcontextprotocol/server-filesystem

# If not installed
npm install -g @modelcontextprotocol/server-filesystem
```

```
# Verify command works
npx @modelcontextprotocol/server-filesystem /tmp

# Check config.toml syntax
cat ~/.code/config.toml
# Look for [mcp_servers.filesystem] section
```

Correct configuration:

```
[mcp_servers.filesystem]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-filesystem", "/path/to/project"]
startup_timeout_sec = 10
```

Error: `MCP server timeout`

Cause: Server takes too long to start

Solution: Increase timeout

```
# ~/.code/config.toml

[mcp_servers.slow-server]
command = "npx"
args = ["-y", "slow-mcp-server"]
startup_timeout_sec = 30  # Increase from default 10
tool_timeout_sec = 120    # Increase tool timeout too
```

Error: `Tool 'filesystem' not found`

Cause: MCP server not configured or failed to start silently

Solution:

```
# List configured MCP servers
code mcp list

# Test server health
code mcp test filesystem

# Check logs for startup errors
code --debug
# Then try invoking the tool
# Check debug output for MCP server errors
```

Error: `MCP server crashed` (mid-session)

Cause: Server bug or resource exhaustion

Solution:

```
# Check server output/logs
code --debug

# Restart Code (MCP servers restart automatically)
code
```

```
# If persistent, try running server manually to see errors
npx @modelcontextprotocol/server-filesystem /tmp

# Report issue to MCP server maintainers
```

Agent Execution Failures

Error: `Command 'claude' not found` (multi-agent)

Cause: CLI tools not installed

Solution:

```
# Install Claude CLI
npm install -g @anthropic-ai/claude-code

# Install Gemini CLI
npm install -g @google/gemini-cli

# Verify installations
which claude
which gemini

# Test commands
claude "test"
gemini -i "test"
```

Error: `ANTHROPIC_API_KEY not set`

Cause: API key not configured for multi-agent setup

Solution:

```
# Set API keys for all providers
export ANTHROPIC_API_KEY="sk-ant-api03-YOUR_KEY"
export GOOGLE_API_KEY="AIza_YOUR_KEY"

# Add to shell profile
echo 'export ANTHROPIC_API_KEY="sk-ant-..."' >> ~/.bashrc
echo 'export GOOGLE_API_KEY="AIza..."' >> ~/.bashrc
source ~/.bashrc

# Verify
echo $ANTHROPIC_API_KEY
echo $GOOGLE_API_KEY
```

Get API keys: - Anthropic: https://console.anthropic.com/settings/keys - Google: https://ai.google.dev/

Error: `/speckit.auto` fails with "agent missing"

Cause: Quality gates configured with unavailable agents

Solution:

Check configuration:

```
# ~/.code/config.toml

[quality_gates]
# Ensure agents are installed and configured
plan = ["gemini", "claude", "code"]  # All must be available

# If you only have OpenAI configured:
plan = ["code"]  # Single agent
tasks = ["code"]
validate = ["code"]
audit = ["code"]
unlock = ["code"]
```

Or install missing providers:

```
npm install -g @anthropic-ai/claude-code @google/gemini-cli
export ANTHROPIC_API_KEY="sk-ant-..."
export GOOGLE_API_KEY="AIza..."
```

Error: `Consensus failed: 0/3 agents responded`

Cause: All agents failed (network, rate limits, or bugs)

Solution:

```
# Check network connectivity
ping api.openai.com
ping api.anthropic.com

# Check rate limits (may need to wait)
# Try with single agent temporarily:

# In Code:
/speckit.plan SPEC-ID --agents code

# Or update config to use single agent
[quality_gates]
plan = ["code"]  # Temporarily use only OpenAI
```

Enable debug mode to see detailed errors:

```
code --debug
```

Performance Issues

Issue: Code CLI is slow to start

Cause: Large history file or MCP servers slow to initialize

Solution:

Reduce history size:

```
# Check history size
ls -lh ~/.code/history.jsonl

# If large (>100MB), truncate
mv ~/.code/history.jsonl ~/.code/history.jsonl.backup
touch ~/.code/history.jsonl

# Or disable history
# In ~/.code/config.toml:
[history]
persistence = "none"
```

Disable slow MCP servers temporarily:

```
# Comment out slow servers in config.toml
# [mcp_servers.slow-server]
# command = "..."
```

Issue: Model responses are very slow

Cause: Complex reasoning, large context, or network issues

Solution:

Reduce reasoning effort:

```
# ~/.code/config.toml
model_reasoning_effort = "low"  # or "minimal"
```

Use faster model:

```
code --model gpt-4o-mini "simple task"
```

Check network:

```
# Test API endpoint connectivity
curl -I https://api.openai.com

# Check if using proxy
echo $HTTP_PROXY
echo $HTTPS_PROXY
```

Issue: High memory usage

Cause: Large conversation history or MCP server memory leaks

Solution:

Start new conversation:

```
# In Code:
/new
```

Restart Code to free memory:

```
# Exit and restart
code
```

Monitor memory:

```
# Check Code process memory
ps aux | grep code
```

Configuration Mistakes

Error: `config.toml: unknown field 'xyz'`

Cause: Typo or invalid configuration option

Solution:

Common typos:

```
# ❌ Wrong (JSON style)
mcpServers.filesystem.command = "npx"

# ✅ Correct (TOML style, snake_case)
[mcp_servers.filesystem]
command = "npx"

# ❌ Wrong
modelProvider = "openai"

# ✅ Correct
model_provider = "openai"
```

Validate config:

```
# Check config syntax
code --config-check

# Print loaded config to verify
code --print-config
```

Refer to docs:

- See config.md for all valid options
- See examples/config.toml for templates

Error: Config changes not taking effect

Cause: Wrong config file location or profile override

Solution:

Verify config file location:

```
# Check which config is loaded
code --print-config | head -20

# Verify file exists
ls -la ~/.code/config.toml

# Not ~/.codex/ (legacy location, read but not written to)
```

Check for profile override:

```
# If you have a profile set:
```

```
profile = "premium"

# It overrides root config
[profiles.premium]
model = "o3"  # This takes precedence
```

Test without profile:

```
code --profile=none
```

Error: `sandbox_mode 'xyz' invalid`

Cause: Invalid sandbox mode value

Solution:

Valid values only:

```
# Valid options:
sandbox_mode = "read-only"         # ✅ No writes
sandbox_mode = "workspace_write"   # ✅ Write to workspace
sandbox_mode = "danger-full-access" # ✅ Full access (risky)

# Invalid:
sandbox_mode = "read_only"   # ❌ Wrong (use dash not underscore)
sandbox_mode = "full-access" # ❌ Wrong (missing "danger-")
```

File Operation Errors

Error: `Permission denied` when writing files

Cause: Sandbox mode prevents writes

Solution:

Check current sandbox mode:

```
# In Code:
/status

# Shows current sandbox_mode
```

Adjust sandbox mode:

```
# Allow workspace writes
code --sandbox workspace-write

# Or update config.toml:
sandbox_mode = "workspace_write"
```

For specific directories, add to writable_roots:

```
sandbox_mode = "workspace_write"

[sandbox_workspace_write]
writable_roots = ["/path/to/additional/dir"]
```

Error: `File not found` when Code tries to read

Cause: File doesn't exist or path incorrect

Solution:

Use absolute paths:

```
# Instead of relative paths:
code "read ~/project/main.py"

# Use absolute path
code "read /home/user/project/main.py"
```

Verify file exists:

```
ls -la /path/to/file
```

Use `--cd` flag to set working directory:

```
code --cd /home/user/project "read main.py"
```

Error: `Git operation failed`

Cause: `.git` directory not writable in workspace-write mode

Solution:

Enable git writes:

```
# ~/.code/config.toml

sandbox_mode = "workspace_write"

[sandbox_workspace_write]
allow_git_writes = true  # Default: true
```

Or use danger-full-access (in isolated environment):

```
sandbox_mode = "danger-full-access"
```

Network and Connectivity

Error: `Connection timeout` or `Network error`

Cause: Network issues, proxy, or firewall

Solution:

Check connectivity:

```
# Test API endpoints
curl -I https://api.openai.com
curl -I https://api.anthropic.com
```

Configure proxy (if behind corporate proxy):

```
# Set proxy environment variables
```

```
export HTTP_PROXY="http://proxy.company.com:8080"
export HTTPS_PROXY="http://proxy.company.com:8080"

# Run Code
code
```

Check firewall:

```
# Ensure outbound HTTPS (port 443) is allowed
# Contact IT if corporate firewall blocks OpenAI/Anthropic domains
```

Error: `SSL certificate verification failed`

Cause: Corporate SSL inspection or outdated certificates

Solution (NOT recommended for production):

```
# Disable SSL verification (use only in development)
export NODE_TLS_REJECT_UNAUTHORIZED=0

# Better: Install corporate CA certificate
# Contact IT for proper certificate installation
```

Error: `502 Bad Gateway` or `503 Service Unavailable`

Cause: OpenAI/Anthropic/Google API outage

Solution:

Check status pages: - OpenAI: https://status.openai.com/ - Anthropic: https://status.anthropic.com/ - Google: https://status.cloud.google.com/

Wait and retry: Services usually recover within minutes

Switch providers temporarily:

```
# Use Anthropic if OpenAI is down
code --model claude-sonnet-3-5 "task"

# Or configure fallback in quality gates
[quality_gates]
plan = ["claude", "gemini"]  # Exclude OpenAI temporarily
```

Platform-Specific Issues

Windows (WSL2)

Issue: `code: command not found` after installation

Solution:

```
# Ensure npm global bin is in PATH
export PATH="$(npm config get prefix)/bin:$PATH"

# Add to ~/.bashrc for persistence
```

```
echo 'export PATH="$(npm config get prefix)/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

Issue: Git operations fail with "permission denied"

Solution:

```
# Clone repos into WSL filesystem (not /mnt/c/)
cd ~
git clone https://github.com/just-every/code.git

# Avoid working in /mnt/c/Users/... (Windows filesystem)
# Use native WSL paths like /home/user/
```

---

macOS

Issue: `xcode-select: command not found`

Solution:

```
# Install Xcode Command Line Tools
xcode-select --install

# Follow prompts to complete installation

# Verify
xcode-select -p
# Should output: /Library/Developer/CommandLineTools
```

Issue: Homebrew installation fails

Solution:

```
# Install Homebrew first
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

# Add Homebrew to PATH (M1/M2 Macs)
echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >> ~/.zprofile
eval "$(/opt/homebrew/bin/brew shellenv)"

# Retry Code installation
brew tap just-every/code
brew install code-cli
```

Linux (Specific Distributions)

Alpine Linux: Build issues with musl libc

Solution:

```
# Install build dependencies
apk add build-base openssl-dev pkgconfig

# Use musl target for Rust
rustup target add x86_64-unknown-linux-musl
cd codex-rs
```

```
cargo build --target x86_64-unknown-linux-musl --release
```

Ubuntu/Debian: Missing libraries

Solution:

```
sudo apt-get update
sudo apt-get install -y build-essential pkg-config libssl-dev
```

Getting Help

Before Asking for Help

1. **Check error message carefully**: Error messages often contain the solution
2. **Review this troubleshooting guide**: Search for your specific error
3. **Check GitHub Issues**: https://github.com/just-every/code/issues
4. **Enable debug mode**: `code --debug` for detailed logs

Gathering Debug Information

When reporting issues, provide:

```
# 1. Version information
code --version

# 2. System information
uname -a                    # OS version
node --version              # Node.js version
npm --version               # npm version
rustc --version             # Rust version (if building from source)

# 3. Configuration (sanitize secrets!)
code --print-config

# 4. Debug logs
code --debug 2>&1 | tee debug.log
# Reproduce issue
# Share debug.log (after removing any API keys!)

# 5. Environment variables
env | grep -E 'OPENAI|ANTHROPIC|GOOGLE|CODE|CODEX'
```

Where to Get Help

Official Documentation: - Installation: installation.md - Setup: first-time-setup.md - Configuration: ../../config.md - FAQ: faq.md

Community Support: - **GitHub Issues**: https://github.com/just-every/code/issues - Search existing issues first - Provide debug information - Include steps to reproduce

- **GitHub Discussions**: https://github.com/just-

every/code/discussions
  - Ask questions
  - Share workflows
  - Request features

Fork-Specific (theturtlecsz/code): - Issues: https://github.com/theturtlecsz/code/issues - Spec-Kit documentation: ../../spec-kit/README.md

---

Reporting Bugs

Good bug report includes:

1. **Clear title**: "MCP server fails to start on Ubuntu 22.04"
2. **Expected behavior**: What should happen
3. **Actual behavior**: What actually happens
4. **Steps to reproduce**:

```
1. Install Code via npm
2. Configure MCP server in config.toml
3. Run `code`
4. Server fails to start
```

5. **Environment**: OS, Code version, Node version
6. **Logs**: Debug logs, error messages
7. **Config** (sanitized): Relevant config.toml sections

Common Error Reference

Quick lookup table for frequent errors:

Error	Common Cause	Quick Fix
`npm: command not found`	Node.js not installed	Install Node.js from nodejs.org
`EACCES: permission denied`	npm permissions	Use `npm install -g --prefix ~/.npm-global`
`code: command not found`	Not in PATH	Add npm global bin to PATH
`401 Unauthorized`	Invalid API key	Check API key at platform.openai.com
`403 Forbidden`	No ChatGPT subscription	Upgrade plan or use API key
`Rate limit exceeded`	Too many requests	Wait or upgrade plan
`MCP server failed to start`	Server not installed	`npm install -g @modelcontextprotocol/server-*`
`Permission`	Wrong sandbox mode	Use `--sandbox workspace-`

`denied` (files)		`write`
`config.toml: unknown field`	Typo in config	Check snake_case: `model_provider` not `modelProvider`
`Connection timeout`	Network/proxy issue	Check connectivity, configure proxy
`Command 'claude' not found`	CLI not installed	`npm install -g @anthropic-ai/claude-code`

Still stuck? → Open an issue on GitHub

Common Workflows

Comprehensive guide to common Code CLI usage patterns and workflows.

Table of Contents

Overview

Code CLI supports two main workflow categories:

1. Spec-Kit Automation (Fork Feature) - Multi-agent consensus-driven development - Full PRD → Plan → Tasks → Implementation → Validation → Audit pipeline - Quality gates at each stage - Automated or manual step-through

2. Manual Coding Assistance - Interactive chat for code questions - Code generation and refactoring - Bug fixing and debugging - Documentation and testing

---

Spec-Kit Automation Framework

What is Spec-Kit? A unique fork feature that provides automated, multi-agent software development workflows with quality gates and consensus validation.

Full Automation Pipeline

Use Case: Complete feature implementation with automated quality checks

Command: `/speckit.auto SPEC-ID`

What It Does: 1. **Specify** - PRD refinement (single agent) 2. **Clarify** - Ambiguity detection (native heuristics, FREE) 3. **Plan** - Work breakdown (3 agents: gemini, claude, gpt-5) 4. **Tasks** - Task decomposition (single agent) 5. **Implement** - Code generation (gpt-5-codex + validator) 6. **Validate** - Test strategy (3 agents) 7. **Audit** - Compliance check (3 premium agents) 8. **Unlock** - Final approval (3 premium agents)

Example:

```
# Step 1: Create new SPEC
code
/speckit.new Add OAuth2 user authentication with JWT tokens

# Output: Created SPEC-KIT-123

# Step 2: Run full automation
/speckit.auto SPEC-KIT-123

# The pipeline will:
# - Generate comprehensive PRD
# - Detect ambiguities and inconsistencies
# - Create multi-agent consensus plan
# - Break down into tasks
# - Generate implementation code
# - Design test strategy
# - Run compliance checks
# - Provide ship/no-ship recommendation
```

Time: ~45-50 minutes **Cost**: ~$2.70 (75% cheaper than previous $11)

Manual Stage-by-Stage Workflow

Use Case: Greater control over each stage, inspect outputs before proceeding

Example Workflow:

```
# 1. Create SPEC
/speckit.new Implement rate limiting for API endpoints

# Output: SPEC-KIT-124

# 2. Optional: Run quality checks
/speckit.clarify SPEC-KIT-124    # Detect vague requirements (FREE)
/speckit.analyze SPEC-KIT-124    # Check consistency (FREE)
/speckit.checklist SPEC-KIT-124  # Quality scoring (FREE)

# 3. Plan stage (multi-agent consensus)
/speckit.plan SPEC-KIT-124
# Review plan.md output
# Time: ~10-12 min, Cost: ~$0.35
# 4. Tasks stage (task decomposition)
/speckit.tasks SPEC-KIT-124
# Review tasks.md output
# Time: ~3-5 min, Cost: ~$0.10

# 5. Implementation (code generation)
/speckit.implement SPEC-KIT-124
# Review generated code
# Time: ~8-12 min, Cost: ~$0.11

# 6. Validation (test strategy)
/speckit.validate SPEC-KIT-124
# Review test plan
# Time: ~10-12 min, Cost: ~$0.35

# 7. Audit (compliance)
/speckit.audit SPEC-KIT-124
# Review security/compliance report
# Time: ~10-12 min, Cost: ~$0.80

# 8. Unlock (ship decision)
/speckit.unlock SPEC-KIT-124
# Review final recommendation
# Time: ~10-12 min, Cost: ~$0.80
```

Advantages: - ✅ Inspect outputs at each stage - ✅ Iterate on specific stages - ✅ Stop early if issues found - ✅ Greater understanding of process

---

Quality Gates Configuration

Customize agent selection per stage to balance cost and quality:

```toml
# ~/.code/config.toml

[quality_gates]
# Native (FREE, instant)
# - /speckit.new, /speckit.clarify, /speckit.analyze,
/speckit.checklist

# Single agent (cheap, ~$0.10, 3-5 min)
tasks = ["gemini"]  # Gemini Flash: 12x cheaper than GPT-4o

# Multi-agent consensus (balanced, ~$0.35, 10-12 min)
plan = ["gemini", "claude", "code"]
validate = ["gemini", "claude", "code"]

# Premium agents (critical, ~$0.80, 10-12 min)
audit = ["gemini-pro", "claude-opus", "gpt-5"]
unlock = ["gemini-pro", "claude-opus", "gpt-5"]
```

Cost Strategies: - **Minimum cost**: Use `["gemini"]` for all stages (~$0.50 total) - **Balanced** (recommended): Above configuration (~$2.70 total) - **Maximum quality**: Premium for all stages (~$11 total)

Spec-Kit Best Practices

1. **Write Clear Descriptions**:

```
# ❌ Bad: Vague
/speckit.new Add auth

# ✅ Good: Specific
/speckit.new Add OAuth2 authentication with Google and GitHub
providers, JWT token management, and session persistence
```

2. **Run Quality Checks Early**:

```
# After creating SPEC, run native checks (FREE)
/speckit.clarify SPEC-KIT-125   # Finds vague language
/speckit.analyze SPEC-KIT-125   # Finds inconsistencies
/speckit.checklist SPEC-KIT-125 # Quality score

# Fix issues before running expensive multi-agent stages
```

3. **Monitor Evidence Footprint**:

```
# Check evidence size after large runs
/spec-evidence-stats --spec SPEC-KIT-125

# Output shows per-SPEC evidence sizes
# Recommended limit: 25 MB per SPEC
```

4. **Use Guardrail Commands** for validation:

```
# Run guardrail checks before implementation
/guardrail.plan SPEC-KIT-125
/guardrail.implement SPEC-KIT-125

# Full guardrail pipeline
/guardrail.auto SPEC-KIT-125 --from plan
```

Manual Coding Workflows

Code Generation

Use Case: Generate new code from scratch

Example 1: Create a new function

```
code
```

Prompt:

```
Create a Python function that validates email addresses using regex.
Include:
- Comprehensive regex pattern (RFC 5322 compliant)
- Docstring with examples
- Type hints
- Error handling for invalid inputs
```

Expected Output:

```
import re
from typing import Union
```

```python
def validate_email(email: str) -> bool:
    """
    Validate email address using RFC 5322 compliant regex.

    Args:
        email: Email address string to validate

    Returns:
        True if email is valid, False otherwise

    Raises:
        TypeError: If email is not a string

    Examples:
        >>> validate_email("user@example.com")
        True
        >>> validate_email("invalid.email")
        False
    """
    if not isinstance(email, str):
        raise TypeError("Email must be a string")

    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email))
```

Example 2: Generate a complete module

Prompt:

```
Create a Python module called database.py that provides:
- Database connection manager (singleton pattern)
- CRUD operations for a User model
- Transaction support
- Connection pooling
- Error handling
Use SQLAlchemy and include docstrings.
```

Code will: 1. Create `database.py` 2. Implement all requested features 3. Add comprehensive docstrings 4. Include type hints 5. Ask for approval before writing

---

Code Modification

Use Case: Modify existing code

Example 1: Refactor for readability

Prompt:

```
Refactor src/utils.py to improve readability:
- Use descriptive variable names
- Add type hints
- Add docstrings to all functions
- Break long functions into smaller ones
- Add comments for complex logic
```

Example 2: Optimize performance

Prompt:

```
Optimize the data processing pipeline in pipeline.py:
- Replace loops with vectorized operations (NumPy/Pandas)
- Add caching for expensive operations
- Use multiprocessing for parallel tasks
- Profile the code and identify bottlenecks
```

Example 3: Add error handling

Prompt:

```
Add comprehensive error handling to api.py:
- Catch specific exceptions (not bare except)
- Add logging for errors
- Return appropriate HTTP status codes
- Add retry logic for network errors
- Validate inputs before processing
```

Code Review and Refactoring

Code Review

Use Case: Review code changes for quality, bugs, security issues

Example:

```
        # Review specific file
        code "Review auth.py for security vulnerabilities, coding best
practices, and potential bugs. Provide specific recommendations."

        # Review changes in git
        code "Review the changes in the last commit and suggest
improvements."

        # Review entire module
        code "Perform a comprehensive code review of the api/ module. Focus
on:
        - Security vulnerabilities
        - Performance issues
        - Code duplication
        - Missing error handling
        - Potential bugs"
```

Code will provide: - Identified issues with severity (critical, high, medium, low) - Specific line numbers - Recommended fixes - Code examples

---

Refactoring Patterns

Example 1: Extract method

Prompt:

```
Refactor the process_order() function in orders.py:
- Extract validation logic into validate_order()
- Extract payment logic into process_payment()
- Extract shipping logic into create_shipment()
- Ensure each function has single responsibility
```

Example 2: Design patterns

Prompt:

```
Refactor database.py to use the Repository pattern:
- Create UserRepository class
- Implement CRUD operations
- Separate database logic from business logic
- Add interface for easy testing/mocking
```

Example 3: Remove code duplication

Prompt:

```
Identify and refactor code duplication in:
- controllers/user_controller.py
- controllers/admin_controller.py
- controllers/api_controller.py

Extract common logic into base controller or helper functions.
```

Testing and Validation

Test Generation

Example 1: Unit tests

Prompt:

```
Generate comprehensive unit tests for validators.py. Include:
- Happy path tests
- Edge cases (empty, null, boundary values)
- Error conditions
- Parametrized tests for multiple inputs
- Mock external dependencies
Use pytest framework.
```

Example 2: Integration tests

Prompt:

```
Create integration tests for the API endpoints in api/users.py:
- Test GET /users (list, pagination, filtering)
- Test POST /users (create, validation errors)
- Test PUT /users/:id (update, not found errors)
- Test DELETE /users/:id (delete, cascade behavior)
Use pytest and FastAPI TestClient.
```

Example 3: End-to-end tests

Prompt:

```
Generate E2E tests for the user registration flow:
1. Navigate to registration page
2. Fill form with valid data
3. Submit and verify success message
4. Verify email confirmation sent
5. Activate account via email link
6. Verify user can login
Use Playwright for browser automation.
```

Test Execution and Debugging

Example 1: Run tests and fix failures

Prompt:

```
Run the test suite and fix any failing tests. For each failure:
- Identify root cause
- Propose fix
- Show before/after diff
- Re-run tests to verify fix
```

Example 2: Improve test coverage

Prompt:

```
Analyze test coverage for src/auth/ module and:
- Identify untested code paths
- Generate tests for uncovered lines
- Target 90%+ line coverage
- Include branch coverage for conditionals
```

Documentation Generation

API Documentation

Example:

Prompt:

```
Generate comprehensive API documentation for api/ module:
- OpenAPI/Swagger spec
- Endpoint descriptions
- Request/response examples
- Authentication requirements
- Error codes and meanings
- Rate limiting info
Output as docs/api.md
```

README Generation

Example:

Prompt:

```
Generate README.md for this project including:
```

```
- Project overview and purpose
- Features list
- Installation instructions
- Quick start guide
- Usage examples
- Configuration options
- Contributing guidelines
- License information
```

Code Documentation

Example:

Prompt:

```
Add comprehensive docstrings to all functions in utils.py:
- Google-style docstrings
- Parameter descriptions with types
- Return value descriptions
- Raises section for exceptions
- Examples section
- Notes for non-obvious behavior
```

CI/CD Integration

Non-Interactive Mode

Code CLI can run in non-interactive mode for automation:

```bash
# Run tests and fix failures (no approval prompts)
code exec "run the test suite and fix any failures"

# Code quality check
code exec --read-only "analyze code quality and generate report"

# Generate reports
code exec --config output_format=json "list all TODO comments"
```

GitHub Actions Integration

Example workflow (`.github/workflows/ai-code-review.yml`):

```yaml
name: AI Code Review

on:
  pull_request:
    types: [opened, synchronize]

jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Setup Code CLI
```

```
        run: |
          npm install -g @just-every/code

      - name: Run AI Code Review
        env:
          OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
        run: |
          code exec --read-only --config output_format=json \
            "Review the changes in this PR for:
             - Security vulnerabilities
             - Code quality issues
             - Performance problems
             - Missing tests
             Output as structured JSON." > review.json

      - name: Post Review Comment
        uses: actions/github-script@v6
        with:
          script: |
            const fs = require('fs');
            const review = JSON.parse(fs.readFileSync('review.json', 'utf8'));

            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: review.summary
            });
```

Pre-Commit Integration

Example (`.git/hooks/pre-commit`):

```bash
#!/bin/bash

# Run Code CLI to check for common issues
code exec --read-only --no-approval \
  "Check the staged changes for:
   - Console.log statements
   - Hardcoded secrets
   - TODOs without issue references
   - Missing error handling
   Exit 1 if issues found."

exit $?
```

Multi-Agent Workflows

Prerequisites: Multi-provider setup (setup guide)

Multi-Agent Planning

Command: `/plan "task description"`

What It Does: - All agents (Claude, Gemini, GPT-5) review the task - Each agent proposes a plan independently - Code synthesizes a consolidated plan from all perspectives - Provides consensus view with highlighted disagreements

Example:

```
        code
        /plan "Migrate the authentication system from session-based to JWT
tokens"
```

Output:

```
📋 Multi-Agent Plan (Consensus: 3/3 agents)

## Agreed Approach:
1. Create JWT token service module
2. Update authentication middleware
3. Migrate existing sessions to JWT
4. Update frontend to use JWT
5. Add token refresh logic
6. Deprecate session storage

## Points of Disagreement:
- Gemini suggests immediate cutover
- Claude recommends gradual migration with feature flag
- GPT-5 proposes dual-auth support during transition

## Recommended: Gradual migration (2 agents in favor)

[Detailed plan follows...]
```

Multi-Agent Problem Solving

Command: `/solve "problem description"`

What It Does: - Multiple agents race to solve the problem - Fastest solution is presented first - Other solutions shown for comparison - Best solution selected based on correctness and speed

Example:

```
        /solve "Why does the user login fail with 'NoneType object has no
attribute email'?"
```

Output:

```
🏁 Solution Race Results:

🥇 Claude (4.2s): Root cause found
   - auth.py line 45: User.query.filter() returns None
   - Fix: Add null check before accessing user.email
   - Proposed fix: [code shown]

🥈 Gemini (5.8s): Same root cause
   - Additional suggestion: Add logging for failed login attempts

🥉 GPT-5 (7.1s): Same root cause
   - Additional suggestion: Consider using user.get('email') with
default
```

```
Recommended fix: Claude's solution with Gemini's logging addition
```

Multi-Agent Code Generation

Command: `/code "feature description"`

What It Does: - Multiple agents generate code independently - Creates separate git worktrees for each implementation - Evaluates all implementations - Merges the optimal solution

Example:

```
/code "Add dark mode support with theme toggle and persistence"
```

Output:

```
⛏ Multi-Agent Code Generation

Creating worktrees:
- worktree/claude: Claude's implementation
- worktree/gemini: Gemini's implementation
- worktree/gpt5: GPT-5's implementation

Evaluation results:
✅ Claude: Clean implementation, uses CSS variables
✅ Gemini: Similar approach, includes transition animations
✅ GPT-5: More complex, includes automatic theme detection

Selected: Claude's implementation
Enhancements added from others:
- Gemini's transition animations
- GPT-5's automatic theme detection

Merging to main worktree...
```

Browser Integration

External Chrome Connection

Use Case: Control existing Chrome browser for debugging, testing, scraping

Setup:

```
# Launch Chrome with remote debugging
google-chrome --remote-debugging-port=9222

# Or on macOS
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --remote-debugging-port=9222
```

In Code:

```
code
/chrome 9222
```

Example Tasks:

```
Navigate to https://example.com and screenshot the homepage

Fill the login form with username "test@example.com" and click
submit

Extract all product prices from the current page

Monitor network requests and identify slow API calls
```

Internal Headless Browser

Use Case: Automated testing, scraping without visible browser

In Code:

```
        /browser https://example.com
```

Example Tasks:

```
Navigate to the search page, search for "rust programming", and
extract the first 10 results

Test the user registration flow:
1. Fill form with test data
2. Submit
3. Verify success message appears
4. Screenshot the confirmation page

Check if the website renders correctly on mobile (375x667 viewport)
```

Best Practices

General Best Practices

1. Be Specific in Prompts:

❌ Bad:

```
Fix the bug
```

✅ Good:

```
Fix the NullPointerException in UserService.authenticate() at line
42. The error occurs when the email parameter is null. Add
validation and return appropriate error message.
```

---

2. Provide Context:

❌ Bad:

```
Optimize this
```

✅ Good:

```
Optimize the data processing pipeline in pipeline.py. Current
```

```
performance: 10 records/second. Target: 100 records/second. Focus on
database queries (N+1 problem) and data transformations (use
vectorization).
```

---

3. Review Before Approving:

- Always review diffs before approving changes
- Understand why changes were made
- Check for unintended side effects
- Verify tests still pass

---

4. Use Appropriate Approval Policy:

```
        # For exploration/learning
        approval_policy = "untrusted"  # Ask before running untrusted
commands

        # For development
        approval_policy = "on-request"  # Model decides when to ask
(recommended)

        # For automation/CI
        approval_policy = "never"  # Full auto (use with read-only or in
isolated environment)
```

---

5. Monitor Costs (API key usage):

```
        # Use cheaper models for simple tasks
        code --model gpt-4o-mini "simple formatting task"

        # Use premium models for complex tasks
        code --model o3 --config model_reasoning_effort=high "complex
architectural decision"

        # For Spec-Kit: Use balanced quality gates configuration
        # Cheap agents for simple stages, premium for critical
```

Spec-Kit Best Practices

1. Start with Quality Checks (FREE):

```
        # Before running expensive multi-agent stages
        /speckit.clarify SPEC-ID    # Find vague requirements
        /speckit.analyze SPEC-ID    # Find inconsistencies
        /speckit.checklist SPEC-ID  # Quality score

        # Fix issues, then proceed
        /speckit.auto SPEC-ID
```

2. Use Manual Workflow for Learning:

```
        # Step through each stage to understand the process
        /speckit.plan SPEC-ID
        # Review plan.md
        /speckit.tasks SPEC-ID
        # Review tasks.md
        # etc.
```

3. Monitor Evidence Footprint:

```
# Check after large runs
/spec-evidence-stats --spec SPEC-ID

# Keep evidence under 25 MB per SPEC
# Archive or clean old evidence if needed
```

Security Best Practices

1. Use Read-Only Mode for Untrusted Code:

```
code --read-only "analyze this repository for security issues"
```

2. Review Sandbox Mode:

```
# Default: safe for most workflows
sandbox_mode = "workspace_write"

# More restrictive
sandbox_mode = "read-only"

# Only in isolated environments (Docker, etc.)
sandbox_mode = "danger-full-access"
```

3. Never Commit Secrets:

```
# Code will warn if detecting potential secrets
# But always review diffs before committing

# Use environment variables for secrets
export API_KEY="secret"

# Not hardcoded in files
api_key = "secret"  # ✖ Bad
```

Next Steps

Now that you understand common workflows:

1. **FAQ** → faq.md
   - Common questions
   - Cost management
   - Privacy and security
   - Comparison with other tools
2. **Troubleshooting** → troubleshooting.md
   - Installation errors
   - Authentication issues
   - Performance problems
   - Common mistakes
3. **Advanced Configuration** → ../../config.md
   - Custom model providers
   - Project hooks
   - Validation harnesses

Master the workflows! → Continue to FAQ
