
Claude Code Training
Press Space for next page →

Contact Info

Ken Kousen
Kousen IT, Inc.

- ken.kousen@kousenit.com
- http://www.kousenit.com
- http://kousenit.org (blog)
- Social Media:
  - @kenkousen (twitter)
  - @kenkousen@foojay.social (mastodon)
  - @kousenit.com (bluesky)
- *Tales from the jar side* (free newsletter)
  - https://kenkousen.substack.com
  - https://youtube.com/@talesfromthejarside

Course Overview

- **Duration**: 5 hours of hands-on learning
- **Format**: Instructor-led with multiple labs
- **Hands-on Labs**: Real codebases in Python, JavaScript, Java
- **Prerequisites**: Command-line experience, development background

Topics Covered

- **Foundation**: Installation, CLI basics, code exploration
- **Core Skills**: Testing, documentation, git operations
- **Customization**: CLAUDE.md, custom commands, hooks, output styles
- **Extensibility**: Skills, Plugins, MCP integration
- **Advanced**: Plan Mode, Subagents, Extended Thinking, SDKs

Pricing Plans

- **Pro Plan** - $20/month
  - ~10-40 prompts per 5 hours
  - Sonnet 4 only
- **Max Plan 5x** - $100/month
  - ~50-200 prompts per 5 hours
  - Sonnet or Opus 4
- **Max Plan 20x** - $200/month
  - ~200-800 prompts per 5 hours
  - Sonnet or Opus 4
- **Note**: Opus 4 uses 5x more credits than Sonnet 4
- **Limits reset**: Every 5 hours

📖 **Full details**: Using Claude Code with your Pro or Max plan

What is Claude Code?

- Command-line AI tool for development
- Context-aware codebase understanding
- Autonomous and collaborative modes
- Multi-language support
- Integrated git operations
- Test generation and documentation

Installation

- Install via npm: `npm install -g @anthropic-ai/claude-code`
- Or download from GitHub releases
- Set API key: `export ANTHROPIC_API_KEY="your-key"`
- Verify: `claude --version`

Basic Usage

- Start Claude Code: `claude`
- Natural language prompts
- File-specific requests
- Multi-step workflows

Creating Projects from Scratch

- **Start from nothing**: Empty directory + idea = working application
- **Iterative development**: Concept → foundation → enhancements
- **Full-stack creation**: UI, logic, styling, tests in one session
- **Real example**: Our `lyrics-trainer` exercise started exactly this way

```
mkdir my-project && cd my-project && git init
claude
"Create a web app that displays song lyrics one line at a time
with Next, Previous, and Play buttons"
```

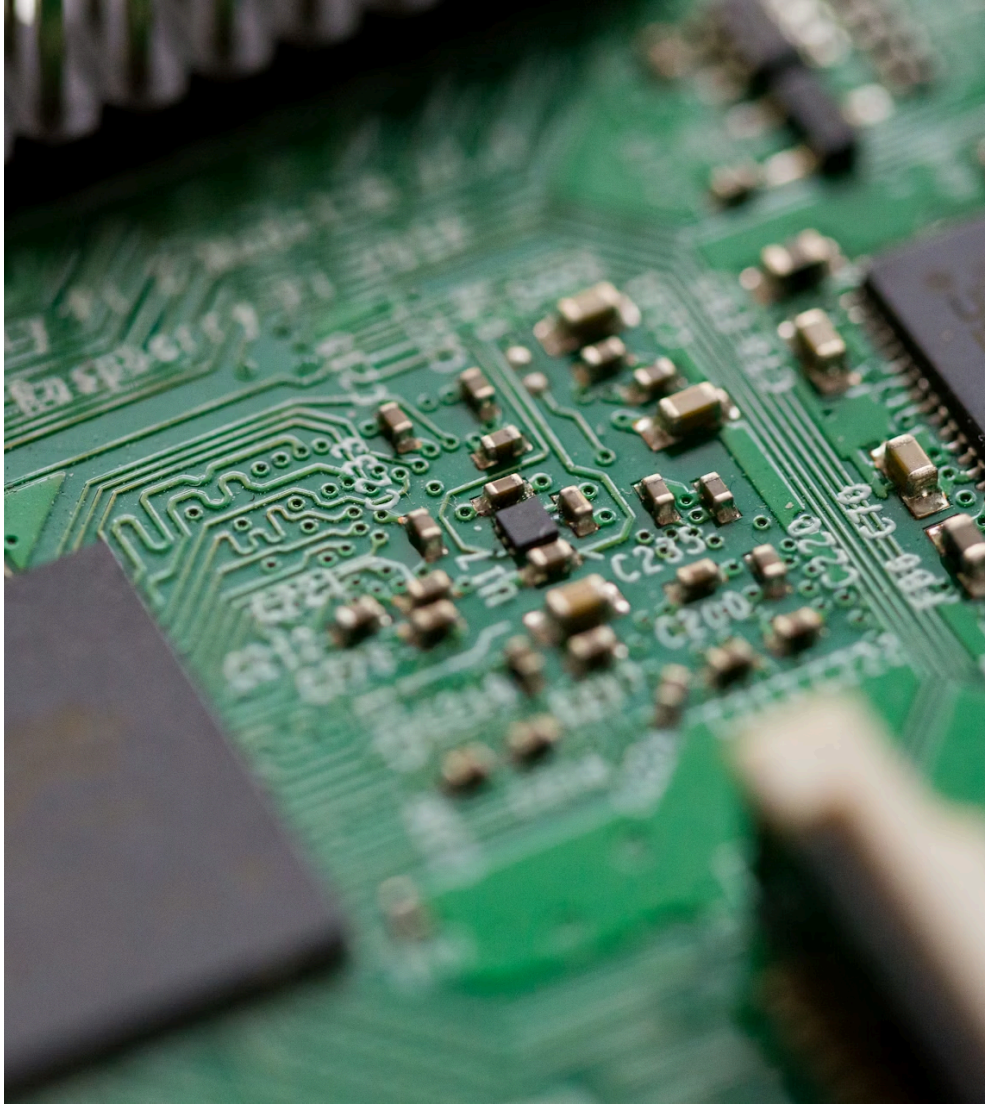
Operation Modes

- **Command Mode** (default) - Interactive conversation
- **Auto-Accept Mode** (Shift+Tab) - Autonomous execution
- **Plan Mode** (Shift+Tab+Tab) - Review plans before execution

Core Productivity Features

Get Productive Immediately

Essential features for daily development work



Code Exploration

- Find files, functions, patterns
- Understand system architecture
- Trace dependencies
- Identify frameworks
- Reference specific files with `@path/to/file.java`

```
"Analyze the UserService class"
"Explain @src/main/java/com/example/UserController.java"
"How does @pom.xml configure Spring Boot?"
```

Test Generation

- Unit test creation
- Edge case identification
- Integration tests
- Mock object setup

```
"Create unit tests for the UserService"
"Add tests for error scenarios"
```

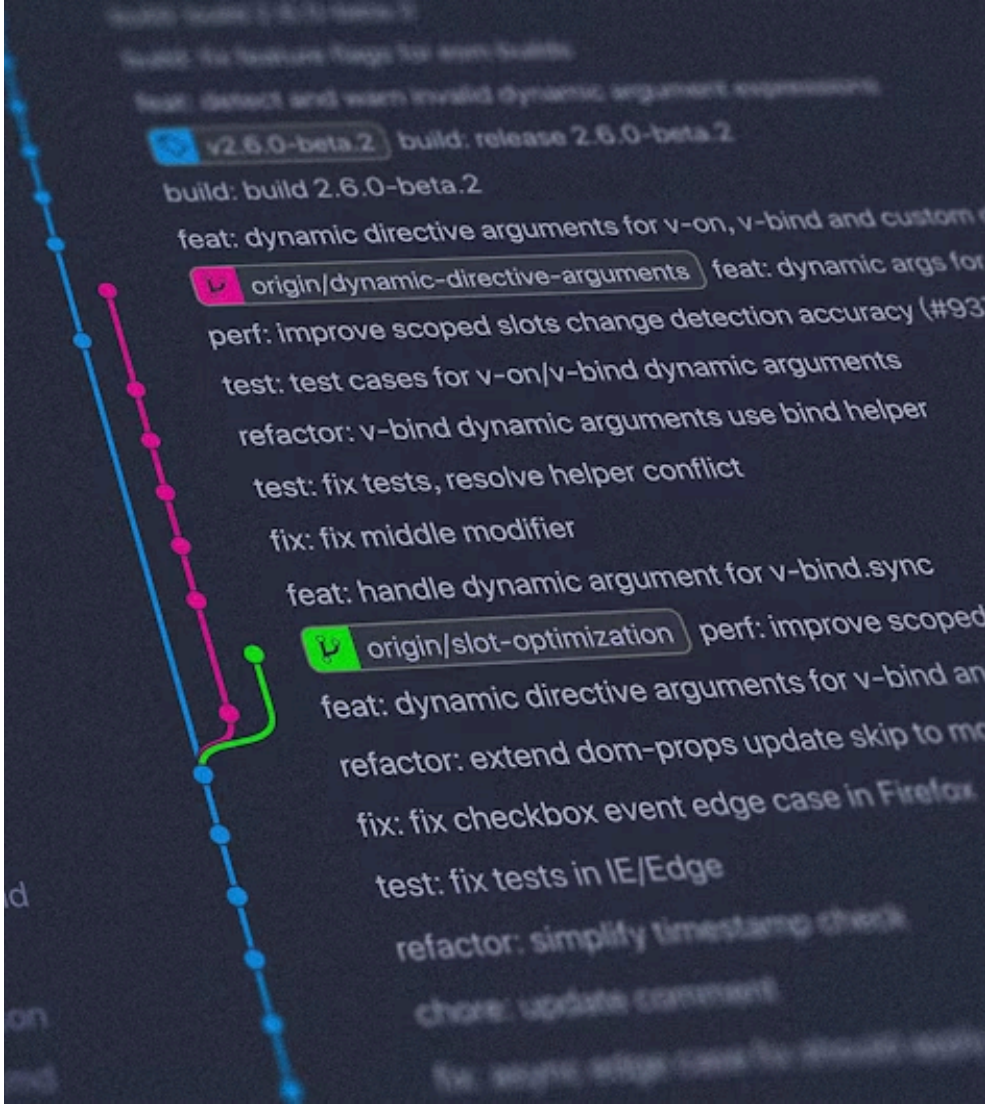
Refactoring Capabilities

- Multi-file operations
- Smart refactoring
- Pattern replacement
- Modern syntax adoption

Git Integration

- Commit message generation
- Branch management
- Merge conflict resolution
- Pull request creation

```
"Commit these changes with an appropriate message"
"Create a pull request for this feature"
```



Multi-Tool Workflows

- **Batch operations**: Call multiple tools in single response
- **Parallel execution**: Run git status + diff + log simultaneously
- **Performance optimization**: Reduces round-trips
- **Complex workflows**: Chain dependent operations

```
# Parallel git operations
"Show me git status, recent commits, and current diff"

# Multi-file analysis
"Check all test files and their coverage simultaneously"
```

💡 **Pro tip**: Request "in parallel" for faster execution

Documentation Generation

- Inline comments
- `README.md` files
- API documentation
- Architecture docs
- `CLAUDE.md` project configuration (covered later)

Debugging Workflows

- Error message analysis
- Stack trace navigation
- Configuration debugging
- Integration debugging

Essential Workflow Tools

Customize Your Experience

Session management and personalization

CLAUDE.md Files

- **Project memory**: `./CLAUDE.md` (shared with team)
- **User memory**: `~/.claude/CLAUDE.md` (personal preferences)
- Auto-discovered up directory tree
- **Quick add**: Start input with `#` to add memory
- **Commands**: `/memory` to edit, `/init` to bootstrap
- **Import files**: Use `@path/to/import` syntax

Custom Statusline

- **Configure context display**: `~/.claude/settings.json` or `/statusline`
- Shows git branch, status, working directory, custom info
- Keeps important context visible without asking
- Reduces repetitive status checks

```
{
  "statusline": {
    "items": [
      {"type": "git_branch"},
      {"type": "git_status"},
      {"type": "cwd"},
      {"type": "custom", "command": "node -v"}
    ]
  }
}
```

Perfect for teams wanting standardized context visibility

Custom Slash Commands

- **Project scope**: `.claude/commands/` (shared with team)
- **User scope**: `~/.claude/commands/` (personal, use `/user:command`)
- **Filename becomes command name** (e.g., `service.md` → `/service`)
- Quick shortcuts for common workflows

Creating Slash Commands

- **Project commands** are shared with the entire team
- **User commands** are personal and require `/user:` prefix
- **Use** `$ARGUMENTS` for dynamic content in commands

```
# Project commands (shared with team)
mkdir -p .claude/commands
echo "Create service for $ARGUMENTS entity" > .claude/commands/service.md

# User commands (personal)
mkdir -p ~/.claude/commands
echo "Fix issue #$ARGUMENTS" > ~/.claude/commands/fix.md

# Usage: /service User  or  /user:fix 123
```

Real-world example - a powerful, personal documentation command:

```
# ~/.claude/commands/docs.md
Update both the README.md and CLAUDE.md files as appropriate.
If either file does not exist, please create it. Generate the
CLAUDE.md file as though the user invoked the init task.
```

Hooks & Automation

- **Event-driven workflow automation** through shell commands
- **Multiple hook types**: SessionEnd, PreToolUse, and custom hooks
- **PreToolUse hooks**: Modify tool inputs before execution (v2.0.10+)
- **SessionEnd hooks**: Clean up or finalize work when session ends (v1.0.85+)
- **Security controls**: Validate and filter operations before they run
- **Post-tool condensing**: Automatically summarize verbose output
- **Configuration**: `~/.claude/settings.json` or `.claude/settings.json`

Hook Example: Security Validation

- **Pre-validate file edits** before they happen
- Block dangerous operations automatically
- Provide feedback that Claude can respond to

```
{
  "hooks": {
    "preToolUse": {
      "Edit": "validate-edit.sh"
    }
  }
}
```

Hook Example: Workflow Automation

- **Auto-format code** on file write
- Run linters, formatters, or validators
- Maintain code quality automatically

```
{
  "hooks": {
    "preToolUse": {
      "Write": "prettier --write $FILE"
    }
  }
}
```

Hook Example: Session Management

- **SessionEnd hooks** run when conversation ends
- Generate summary reports or logs
- Clean up resources or finalize work

```
{
  "hooks": {
    "sessionEnd": "generate-session-report.sh"
  }
}
```

Important: Treat hook feedback as user input - Claude adjusts if blocked

Output Styles

- **Customize how Claude presents solutions** to match learning preferences
- **Built-in styles**: "Explanatory" and "Learning" modes
- **Custom styles**: Create your own in `~/.claude/output-styles/`
- **Configure via settings**: Set default style or switch per-session
- **Use cases**:
  - Educational contexts (verbose explanations)
  - Production work (concise, action-focused)
  - Code review (detailed analysis)
  - Quick fixes (minimal commentary)

Using Built-in Output Styles

- **Explanatory**: Verbose with detailed explanations
- **Learning**: Teaching-focused with step-by-step guidance
- **Configure in settings** or via command line

```
{
  "outputStyle": "explanatory"
}
```

```
claude --output-style learning
```

Creating Custom Output Styles

Create `~/.claude/output-styles/production.md`:

```
---
name: Production
description: Concise output for experienced developers
---

# Instructions for Claude

- Be concise and action-focused
- Skip explanations unless asked
- Show code without lengthy preambles
- Assume expert-level knowledge
```

Then use: `claude --output-style production`

Resuming Conversations

- `--continue` : Automatically resume most recent conversation
- `--resume` : Interactive picker showing conversation history with timestamps and message counts
- **Full history restored**: Complete message context maintained (even hundreds of messages)
- **Original settings preserved**: Model and configuration retained
- **Stored locally**: Complete conversation database maintained on your machine

```
# Continue most recent conversation
claude --continue

# Show conversation picker with details
claude --resume

# Continue with new prompt
claude --continue --print "Continue with my task"
```

Working with Images

- **Drag and drop** images into Claude Code window
- **Copy/paste** with `Ctrl+V` (not `Cmd+V` even on a Mac!)
- **Provide file path**: "Analyze this image: `/path/to/screenshot.png`"
- Analyze UI designs, error screenshots, diagrams
- Generate code from visual mockups
- Debug visual issues and layouts

```
# Common image workflows
"Analyze this error screenshot and suggest fixes"
"Generate HTML/CSS for this UI mockup"
"Explain what this diagram shows"
"Convert this whiteboard sketch to code"
```

Jupyter & Data Science Support

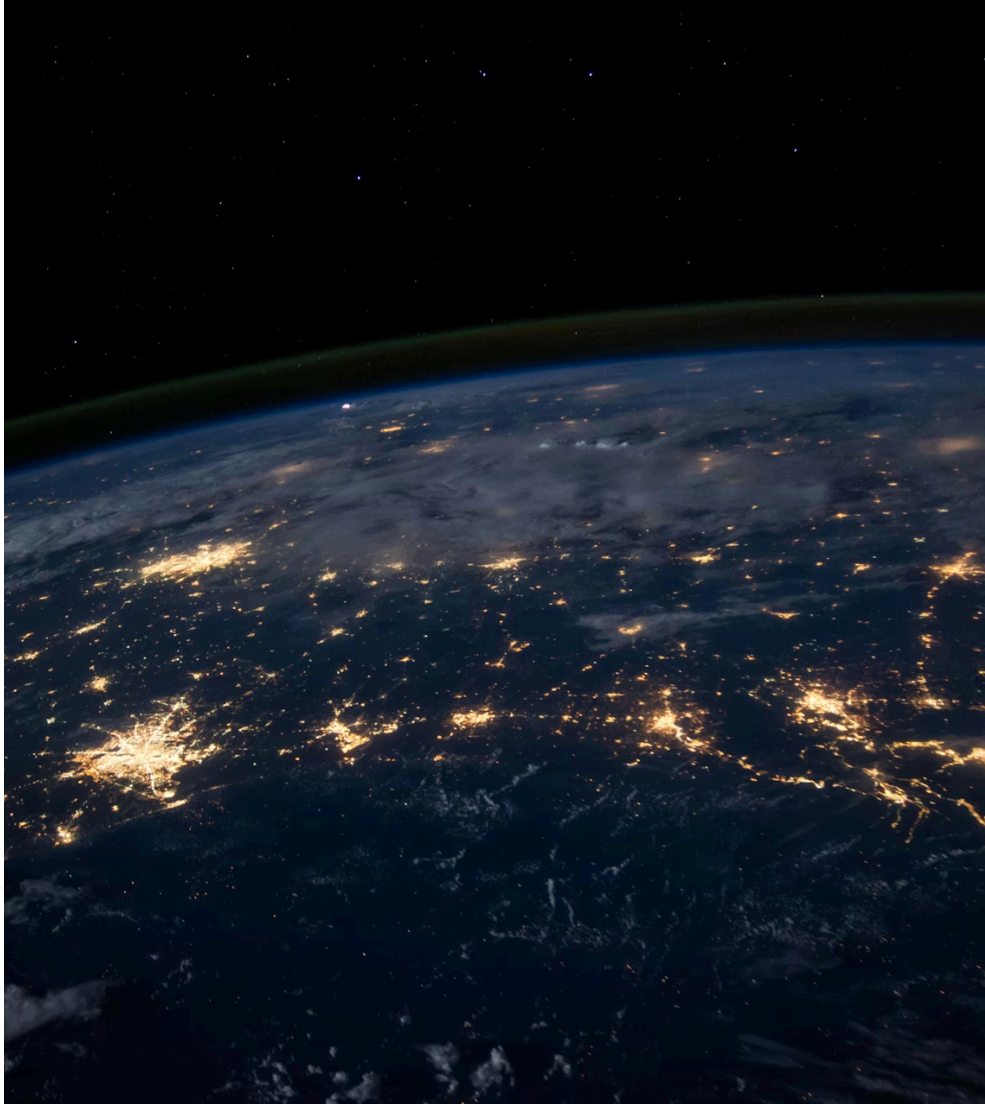
- **Read .ipynb files** with full cell outputs
- **Analyze notebooks**: Code, markdown, and visualizations
- **Edit notebook cells**: Use NotebookEdit tool
- **Data analysis workflows**: Process datasets and results
- **Visualization understanding**: Interpret charts and graphs

```
"Analyze this Jupyter notebook and explain the data pipeline"
"Add error handling to the data processing cells"
"Convert this notebook to a production Python script"
```

Advanced Features

Power User Capabilities

Complex features for sophisticated workflows



Skills: Persistent Domain Expertise

- **Modular capabilities** that extend Claude's functionality beyond the base model
- **Three-tier loading system** for efficiency:
  - Metadata (always loaded): Name and description (~100 tokens)
  - Instructions (triggered): Main SKILL.md with procedures
  - Resources (on-demand): Scripts, templates, reference files
- **Automatic activation** when contextually relevant
- **Filesystem-based** knowledge that persists across conversations
- **Progressive disclosure**: Load only what's needed for each task

Built-in Skills

Anthropic provides four production-ready Agent Skills:

- 📊 **Excel (xlsx)**: Build spreadsheets, generate reports with charts
- 📄 **Word (docx)**: Create and format professional documents
- 📽️ **PowerPoint (pptx)**: Create and edit presentations
- 📑 **PDF (pdf)**: Generate formatted PDF documents and reports

Usage: These skills activate automatically when you reference relevant file types or request document creation

```
# Skills activate automatically
"Create a quarterly report spreadsheet with sales data"
"Generate a PDF proposal document with our company branding"
"Build a presentation deck for the product launch"
```

Creating Custom Skills

Skill Structure

```
~/.claude/skills/my-skill/
├── SKILL.md          # Required: Instructions with YAML frontmatter
├── templates/        # Optional: Reusable templates
├── scripts/          # Optional: Helper scripts
└── reference/        # Optional: Documentation, schemas
```

Example SKILL.md

```
---
name: Java Spring Generator
description: Generate Spring Boot components following team patterns
---

# Instructions

When generating Spring Boot code:
1. Use constructor injection, not @Autowired
2. Follow package conventions: controller/service/repository
3. Include comprehensive JavaDoc
4. Generate corresponding test files with @SpringBootTest
```

Plugins: Team-Wide Extensibility

- **Plugin system** (v2.0.12+) provides installable packages of commands, agents, hooks, and MCP servers
- **Plugin marketplace**: Discover and share team workflows
- **Repository-level config**: `extraKnownMarketplaces` for enterprise control
- **Management commands**:
  - `/plugin install <name>` - Install from marketplace
  - `/plugin enable/disable <name>` - Control active plugins
  - `/plugin marketplace` - Browse available plugins
  - `/plugin list` - View installed plugins

Plugin Use Cases

Enterprise Workflows

- Standardize code generation patterns across teams
- Enforce security review processes
- Automate compliance documentation
- Integrate with internal tools and APIs

Team Collaboration

- Share custom commands and agents
- Distribute MCP server configurations
- Maintain consistent development practices
- Onboard new team members faster

Example

```
# Install company's internal plugin
/plugin install acme-corp-standards

# Plugin provides:
# - Custom slash commands for service generation
```

Extended Thinking

- Trigger deeper analysis with "think" in your prompts
- Use "**think more**", "**think harder**", "**ultrathink**" for deeper reasoning
- Shows thinking process as *italic gray text*
- Perfect for complex architectural decisions
- **Verification pattern**: "Before you finish, verify your solution and fix any issues"
- **Note**: Thinking tokens count toward usage but provide higher quality results

```
# Examples of extended thinking prompts
"Think deeply about the best approach for implement
Before you finish, verify your solution and fix any

"Think harder about potential security vulnerabilit
"Think more about the tradeoffs between these two d
```

Plan Mode

- Press `Shift+Tab+Tab` to activate
- Claude presents implementation plan
- Review strategy before execution
- Approve or modify approach
- Perfect for complex changes
- **Actually uses the Plan subagent** behind the scenes

Subagents: Specialized Task Handlers

- **Autonomous agents** that Claude launches for specialized tasks
- **Dynamic selection** (v2.0.28+): Claude chooses appropriate subagent automatically
- **Model selection**: Different subagents can use different models for optimal performance
- **Common subagent types**:
  - **Plan**: Strategic task decomposition and planning
  - **Explore**: Fast codebase exploration and search
  - **Testing**: Test generation and quality assurance
  - **Documentation**: Technical writing and content creation
- **Transparent operation**: You see subagent activity in the output
- **Efficiency**: Specialized agents work faster than general-purpose conversations

When Claude Uses Subagents

Automatic Activation

Claude launches subagents when tasks match specialized capabilities:

```
# Triggers Explore subagent
"Find all API endpoints in this codebase"
"How does authentication work across the project?"

# Triggers Plan subagent (Plan Mode)
Shift+Tab+Tab or "Create a plan for adding OAuth"

# Triggers Testing subagent
"Generate comprehensive test coverage for UserService"

# Triggers Documentation subagent
"Create API documentation for all REST endpoints"
```

You don't manage this - Claude handles subagent selection automatically for optimal results

Model Context Protocol (MCP)

- Standard protocol for AI-to-system connections
- Tool integration (APIs, databases, services)
- Context enhancement for better AI responses
- Security controls and permissions

MCP Configuration

Import from Claude Desktop

```
claude mcp add-from-claude-desktop
```

Automatically imports MCP servers from Claude Desktop config

Manual Configuration

```
# Project-scoped (default)
claude mcp add <server-name>

# User-scoped (available across all projects)
# Remote HTTP transport
claude mcp add --transport http context7 https://mcp.context7.com/mcp
# Local execution
claude mcp add context7 -- npx -y @upstash/context7-mcp

# Management commands
claude mcp list
claude mcp remove <server-name>
```

Config Locations

- **Claude Desktop**: `~/Library/Application Support/Claude/claude_desktop_config.json`

MCP Server Examples

- **GitHub MCP** - Repository operations, issues, PRs
- **Context7** - Downloads latest API docs and examples for modern code
- **Docker MCP Toolkit** - Container management and operations
- **Playwright MCP** - UI test generation and browser automation
- **Heroku MCP** - Deployment and app management

Setting Up MCP Servers

- Interactive setup: `claude mcp`
- Local servers: Full configuration control
- Remote servers: OAuth authentication, zero maintenance
- Docker MCP Toolkit: `docker mcp gateway run`

```
# List existing MCP servers
claude mcp list

# Add local server
claude mcp add my-server -e API_KEY=123 -- /path/to/server

# Add remote server (HTTP)
claude mcp add --transport http remote-server https://example.com/mcp

# Add Docker MCP toolkit
claude mcp add docker-mcp docker mcp gateway run
```

Claude Code SDKs

- **Available SDKs**: TypeScript, Python, Command Line
- **Build AI-powered coding assistants** into your workflows
- **Multi-turn conversations** and session management
- **Custom system prompts** and flexible I/O formats
- **MCP integration** for extended capabilities

```
# Command line usage
claude -p "Write a function to calculate Fibonacci numbers"
claude -p "Generate a hello world function" --output-format json
```

```
// TypeScript SDK
import { query } from "@anthropic-ai/claude-code";

for await (const message of query({
  prompt: "Write a haiku about foo.py",
  options: { maxTurns: 3 }
})) {
  // Process messages
}
```

SDK: Python Integration

- **Async iteration** over Claude responses
- **Context files** passed directly to queries
- **Turn limits** for controlled execution

```python
from claude_code import query

async for msg in query("Refactor this module",
                       context_files=["app.py"],
                       max_turns=3):
    print(msg.content)
```

SDK: CI/CD Integration

- **Automate code reviews** in pull requests
- **JSON output** for parsing results
- **Integrate with GitHub Actions**, GitLab CI, etc.

```
# GitHub Actions example
- name: AI Code Review
  run: |
    claude -p "Review PR changes" --output-format json > review.json
```

SDK: Git Hooks

- **Pre-commit security checks** before code is committed
- **Limit tools** for focused, fast execution
- **Fail-fast** on security issues

```
#!/bin/bash
# .git/hooks/pre-commit
claude -p "Check for security issues" --allowed-tools read,grep
```

VS Code Extension (Beta)

- **Native IDE integration** bringing Claude Code into your editor
- **In-editor experience**: Work with Claude without leaving VS Code
- **Context-aware**: Accesses your workspace files and settings
- **All Claude Code features**: Skills, MCP, custom commands available
- **Installation**: Search "Claude Code" in VS Code Extensions marketplace
- **Beta status**: Actively developed, new features being added

VS Code Extension Features

Integrated Workflow

- Ask questions about code in sidebar
- Reference files with `@` syntax directly in VS Code
- Generate and edit code without context switching
- View diffs and approve changes inline

Best Use Cases

- Developers who prefer IDE-native workflows
- Teams already standardized on VS Code
- Projects with complex workspace configurations
- Rapid iteration with immediate feedback

Note: Terminal Claude Code remains the primary experience with fullest feature set

Management & Control

Monitor and Control

Cost, context, and permission management

Cost Monitoring

- Use `/cost` command to check usage
- Shows current usage and limits
- Pro Plan: Displays prompt count vs limit
- Max Plans: Shows monthly usage summary
- Limits reset every 5 hours
- Plan ahead for intensive work sessions

```
# Check your current usage
/cost

# Example output (Pro Plan):
# 📊 Cost information:
#    - Input tokens: 1,245
#    - Output tokens: 3,782
#    - Total cost: $0.076

# Example output (Max Plan):
# With your Claude Max subscription, no need to monitor cost
# — your subscription includes Claude Code usage
```

Context Management

- Use `/compact` command to compress conversation history
- Claude Code automatically compacts when context limit approaches
- Warning message appears before auto-compaction
- Preserves essential information while reducing token usage
- Manual compaction gives you control over timing

```
# Manually compact the conversation
/compact

# Warning message example:
# ⚠️ Context limit approaching. Auto-compacting in next response
# to preserve conversation history and continue working.
```

Smart Context Management

- **Auto-compaction**: Automatic when approaching limits
- **Warning messages**: Advance notice before compaction
- **Preservation strategy**: Keeps essential information
- **Manual control**: Use `/compact` proactively
- **Long conversations**: Maintains context over hundreds of messages

```
# Warning you'll see:
⚠️ Context limit approaching. Auto-compacting in next response
to preserve conversation history and continue working.

# Proactive management:
/compact  # Compress conversation manually
```

Configuring Permissions

- **Fine-grained control** over Claude Code's capabilities
- **Use** `/permissions` **UI** to manage tool permissions
- **Allow/Deny rules** for specific tools and actions
- **Enterprise policies** for organization-wide control
- **Permission precedence**: Enterprise → CLI → Project → User

```
# Example permission rules
Bash(npm run test:*)     # Allow npm test commands
Edit(docs/**)           # Allow editing docs directory
Read(src/*)             # Allow reading source files

# Access permissions UI
/permissions
```

Permission Patterns

Common Permission Profiles

- **Development Mode**: Full access for active coding
- **Review Mode**: Read-only for code reviews
- **Safe Mode**: Limited tools for sensitive operations
- **CI/CD Mode**: Specific tools for automation

```
# Quick permission profiles via aliases
alias claude-dev='claude --allowed-tools all'
alias claude-review='claude --allowed-tools read,grep'
alias claude-safe='claude --disabled-tools bash,webfetch'
alias claude-ci='claude --allowed-tools bash,git,test'
```

Team & Best Practices

Collaborate Effectively

Team workflows and professional practices

Git Worktrees for Parallel Sessions

- Check out multiple branches into separate directories
- Run Claude Code sessions independently on each branch
- Share git history while isolating working files
- Perfect for multi-feature development

```
# Create worktrees for parallel work
git worktree add ../project-feature-a -b feature-a
git worktree add ../project-bugfix bugfix-123

# Run Claude Code in each directory
cd ../project-feature-a && claude
cd ../project-bugfix && claude

# Manage worktrees
git worktree list
git worktree remove ../project-feature-a
```

Effective Prompting & Best Practices

Effective Prompting

- **Be specific** about what you want to achieve
- **Provide context** about your goals and constraints
- **Use iterative refinement** for complex tasks
- **Include examples** when possible to show desired patterns

Best Practices

- **Create a git branch first** for any significant changes
- **Commit checkpoints regularly** during development
- **Review all AI-generated code** before accepting
- **Test generated code** thoroughly

Troubleshooting & Configuration

System Health Check

```
claude /doctor  # Diagnose installation issues
```

Global Configuration

```
claude config set -g model claude-sonnet-4
claude config set -g verbose true
claude config set -g max_conversation_turns 10
```

Check Current Settings

```
claude config list  # View all settings
echo $ANTHROPIC_API_KEY  # Verify API key
```

Security & Permissions

Security-Focused Aliases

```
# Safe mode - limited tools
alias claude-safe='claude --allowed-tools read,write,edit'

# Review mode - read-only
alias claude-review='claude --allowed-tools read,grep'

# Development mode - all tools
alias claude-dev='claude --allowed-tools all'
```

Tool Restrictions

```
# Disable specific tools
claude --disabled-tools bash,webfetch

# Allow only specific tools
claude --allowed-tools read,write,edit,task
```

Common Issues: Installation

- **Command not found** → Check PATH: `npm list -g @anthropic/claude-code`
- **Permission denied** → Use correct npm prefix or sudo
- **Windows users** → WSL is required (not native Windows)

```
# Reinstall globally
npm uninstall -g @anthropic/claude-code
npm install -g @anthropic/claude-code
```

Common Issues: Runtime

- **API key not found** → Set `ANTHROPIC_API_KEY` environment variable
- **Rate limits** → Use `/cost` to monitor usage
- **Context too large** → Use `/compact` to reduce conversation size

```
# Alternative installation: Direct binary
curl -fsSL https://storage.googleapis.com/anthropic-releases/claude-cli/install.sh | bash
```

Development Process

- Start with clean git state
- Generate tests if none exist
- Commit checkpoints regularly
- Use Claude for git workflows (commits, issues, merges)
- Use git worktrees for parallel sessions on different branches
- Review changes before accepting
- Test generated code thoroughly

Team Collaboration

- Share `CLAUDE.md` configurations
- Document successful patterns
- Review AI-generated code together
- Establish team conventions

Quick Access

Claude Code Docs

docs.anthropic.com/claude-code

Course Repository

github.com/kousen/claude-code-training

Important Links

📚 Claude Code Documentation

`https://docs.anthropic.com/en/docs/claude-code`

🐙 Official GitHub Repository

`https://github.com/anthropics/claude-code`

💻 Course Source Code & Exercises

`https://github.com/kousen/claude-code-training`

🆘 Support & Issues

`https://github.com/anthropics/claude-code/issues`

Community & Social

Thank You!

Questions?

Kenneth Kousen
Author, Speaker, Java & AI Expert

kousenit.com | @kenkousen