COMPREHENSIVE TRAINING PROGRAM

Mastering Claude Code

From Zero to Power User

AI-Powered
Development Assistant

Terminal-Based
Command Line Interface

Context-Aware
Codebase Understanding

Based on Documentation by Florian BRUNIAUX
Inspired by Claudelog.com

January 2025
90-120 Minutes

Training Roadmap

01 **Introduction & Setup**
Slides 3-6

02 **Quick Start Guide**
Slides 7-14

03 **Core Concepts**
Slides 15-22

04 **Configuration System**
Slides 23-30

05 **Agents & Specialization**
Slides 31-38

06 **Advanced Features**
Slides 39-46

07 **Best Practices & Troubleshooting**
Slides 47-52

08 **Daily Workflows**
Slides 53-56

09 **Maturity & Success**
Slides 57-60

Total Duration: 90-120 minutes

01

Introduction & Setup

Understanding Claude Code and getting started

What is Claude Code?

AI-powered development assistant capabilities and use cases

Learning Path

4-week progression from beginner to power user

Installation

Prerequisites and setup procedures

01

What is Claude Code?

Definition

Claude Code is an **AI-powered development assistant** by Anthropic that lives in your terminal, understands your codebase, and helps you code faster through natural language commands.

Core Philosophy

Execute routine tasks, explain complex code, handle git workflows

Key Capabilities

- ✓ **Code Analysis:** Understands complex codebases, identifies patterns
- ✓ **File Editing:** Proposes changes with diffs, multi-file operations
- ✓ **Debugging:** Systematic error investigation, root cause analysis
- ✓ **Automation:** Git workflows, testing, repetitive tasks

Claude Code vs Traditional AI Assistants

Context

GitHub Copilot: Limited context

Claude Code: Full conversation context

Operations

Traditional: Single file focus

Claude Code: Multi-file operations

Execution

Most tools: Code suggestions only

Claude Code: Command execution

01

Learning Path Overview

1 Day 1

Installation & Basic Workflow

- ✓ Install Claude Code using preferred method
- ✓ Complete first launch and authentication
- ✓ Master essential commands (/help, /status, /clear)
- ✓ Practice basic interaction loop

Expected: 2-4 hours

2 Week 1

Configuration & Context Management

- ✓ Create project-specific CLAUDE.md files
- ✓ Master context management strategies
- ✓ Understand configuration hierarchy
- ✓ Set up permission modes and safety rules

Expected: 3-5 hours

3 Week 2-3

Agents, Skills & Customization

- ✓ Create custom agents for specialized tasks
- ✓ Build reusable skills and knowledge modules
- ✓ Set up hooks for automation
- ✓ Configure MCP servers for external tools

Expected: 4-6 hours

4 Month 1+

Advanced Patterns & Integration

- ✓ Implement CI/CD integration
- ✓ Master advanced composition patterns
- ✓ Optimize team workflows
- ✓ Contribute to community knowledge

Continuous Learning

01

Prerequisites & Installation

Prerequisites

Basic Command Line Knowledge
Navigating directories, running commands

Git Fundamentals
Commits, branches, basic workflows

Text Editor Familiarity
VS Code, JetBrains, or similar

Anthropic API Access
Account with Claude subscription

Installation Methods

Recommended - Shell Script

```
curl -fsSL https://claude.ai/install.sh | sh
```

Option B - npm

```
npm install -g @anthropic-ai/claude-code
```

Option C - Homebrew (macOS)

```
brew install claude-code
```

Verification

```
claude --version
```

Should show version number if installed correctly

First Launch

```
cd your-project
claude
```

Navigate to project directory first

Setup Steps

1. Authenticate with Anthropic
2. Accept terms of service
3. Wait for project indexing

02

Quick Start Guide

From installation to your first productive session

First Launch

Setup and authentication

Interaction Loop

7-step workflow pattern

Essential Commands

Core 7 commands mastery

First Task

Hands-on bug fixing

02

First Launch & Setup

Getting Started

Step 1: Navigate to Project

```
cd your-project
```

Always launch from your project root directory

Step 2: Start Claude Code

```
claude
```

Initial launch triggers setup process

First-Time Setup

1. **Authenticate**
   You'll be redirected to Anthropic to log in with your account
2. **Accept Terms**
   Review and accept the terms of service
3. **Project Indexing**
   Claude automatically indexes your codebase for understanding

What Happens During Indexing?

File Structure
Analyzes directory tree and file relationships

Code Patterns
Identifies languages, frameworks, and conventions

Git History
Recent commits, branches, and change patterns

Configuration
Package.json, config files, and project setup

Indexing Time: Small projects: 5-10 seconds | Large codebases: 30-60 seconds

02

The Interaction Loop

Every Claude Code interaction follows this 7-step pattern

1 DESCRIBE
You explain what you need in natural language

2 ANALYZE
Claude explores codebase, reads relevant files, understands context

3 PROPOSE
Claude suggests changes with clear diff showing additions/removals

4-5 REVIEW → DECIDE
You evaluate proposal, then accept (y), reject (n), or modify (e)

6 VERIFY
Run tests, check behavior, ensure changes work correctly

7 COMMIT
Save changes to git with appropriate message (optional)

Key Insight: You Remain in Control

The loop is designed so that **you remain in control**. Claude proposes changes, but you decide whether to accept, reject, or modify them. The diff is your safety net—always read it before accepting.

02

Essential Commands (The Core 7)

These 7 commands cover 90% of daily usage

Command	Action	When to Use
/help	Show all commands	When you're lost or need reference
/clear	Clear conversation	Start fresh or between tasks
/compact	Compress context	Running low on context (>70%)
/status	Show session info	Check context usage and session state
/plan	Read-only mode	Safe exploration without modifications
/rewind	Undo changes	Made a mistake or want to back up
/exit	Quit Claude Code	Done working (or Ctrl+D)

Keyboard Shortcuts

Shift+Tab Cycle permission modes	Esc × 2 Rewind (undo)
Ctrl+C Interrupt operation	Tab Autocomplete
Shift+Enter New line	Ctrl+D Exit

@ File References

@file.ts	Reference specific file
@agent-name	Call custom agent
!command	Run shell command

02

File References & Quick Actions

@File Reference Syntax

@path/to/file.ts

Reference a specific file for context or editing

```
Fix the validation in @src/components/LoginForm.tsx
```

@agent-name

Invoke a custom agent for specialized tasks

```
@code-reviewer audit this for security issues
```

!shell-command

Execute shell commands directly

```
!npm test
```

IDE Integration

VS Code — Alt+K

Launch Claude Code with current file context

JetBrains — Cmd+Option+K

Open selected code in Claude Code terminal

Terminal — claude

Direct terminal access with full project context

Practical Examples

Example 1: File Reference

```
Add validation to @src/login.ts
```

Claude focuses on specific file

Example 2: Agent + Command

```
@debugger find memory leak
```

Specialized debugging approach

Example 3: Shell Integration

```
!git branch && !npm test
```

Combine git and testing workflows

02

Permission Modes Explained

Three modes that control how much autonomy Claude has

Mode	Editing	Execution	Use Case
Default	Asks	Asks	Learning and safety-first environments
Auto-accept	Auto	Asks	Trusted operations and speed workflows
Plan Mode	Blocked	Blocked	Safe exploration and code review

Default Mode

Claude asks permission before editing files or running commands. This is the safest mode for learning.

Best for:

- New users
- Production code
- Cautious environments

Auto-accept Mode

Claude automatically executes file changes but still asks before running shell commands.

Best for:

- Well-defined tasks
- Rapid prototyping
- Trusted projects

Plan Mode

Claude can only read and analyze—no modifications allowed. Perfect for safe exploration.

Best for:

- Code review
- Understanding code
- Planning features

How to Switch Modes

Press Shift+Tab to cycle through modes, or use /plan for Plan Mode

02

Your First Task: Step-by-Step

Complete walkthrough: fixing a real bug from start to finish

1 Describe the Problem

You type:

> There's a bug in the login function - users can't log in with email addresses containing a plus sign

2 Claude Analyzes

- Searches codebase for login-related files
- Reads login implementation code
- Identifies email validation regex issue

3 Claude Proposes Fix

Proposed change (diff):

```
- const emailRegex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

+ const emailRegex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
```

4 Review Carefully

You examine the diff:

- ✓ Added `%+` to allow plus signs
- ✓ Change looks correct and minimal

5 Accept & Verify

Press 'y' to accept

Claude applies the change to the file

Run tests to verify:

```
!npm test
```

6 Commit (Optional)

Commit the fix:

```
Commit this fix
```

✓ Claude creates appropriate commit message

02

Day 1 Productivity Checklist

You're ready for Day 2 when you can complete all 8 items

Basic Operations

- [x] **Launch Claude Code in your project**
  Navigate to project directory, run 'claude', complete setup
- [x] **Describe a task and review proposed changes**
  Use natural language to request a change, examine the diff
- [x] **Accept or reject after reading diff**
  Press 'y' to accept, 'n' to reject, 'e' to edit
- [x] **Run shell command with !**
  Execute commands like !npm test or !git status

Context Management

- [x] **Reference a file with @**
  Use @path/to/file.ts to focus on specific files
- [x] **Use /clear to start fresh**
  Reset conversation when switching tasks or context is cluttered
- [x] **Use /status to check context**
  Monitor Ctx(u) percentage and manage context proactively
- [x] **Exit cleanly with /exit**
  Proper session cleanup (or use Ctrl+D)

What to Expect After Completing This Checklist

You'll be comfortable with Claude Code's basic workflow, understand the interaction loop, and be ready for more advanced concepts like context management and configuration.

03

Core Concepts

Understanding context, Plan Mode, and recovery strategies

Context

Working memory management

Budget Crisis

Context zones and warnings

Plan Mode

Look but don't touch mode

Recovery

Strategies for common issues

03 What is Context?

Understanding Claude's working memory

Definition

Context is Claude's **"working memory"** for your conversation. Think of it like RAM in a computer—when it fills up, things slow down or fail.

200,000 Token Budget

Enough for ~150,000 words or ~500 pages of code

What's Included

- **Messages:** All conversation history
- **Files Read:** Code Claude has examined
- **Command Outputs:** Results from shell commands
- **Tool Results:** Tool execution results

The RAM Analogy

Like RAM

Fast access working memory

Fills Up

Every interaction adds data

Performance

Slower when near capacity

Critical

Errors when full

🔑 **Key Insight:** Context management is the most important skill for effective Claude Code usage

03

The Context Budget Crisis

Learn to recognize and manage context usage

Context Zones

🟢 Green Zone — 0-50%

Work freely. Plenty of context available for complex operations.

🟡 Yellow Zone — 50-75%

Be selective. Avoid reading large files unnecessarily.

🔴 Red Zone — 75-90%

Use /compact or /clear immediately. Risk of errors increases.

⚫ Critical Zone — 90%+

Must clear immediately. Claude may start making errors.

Reading the Statusline

```
Claude Code | Ctx(u): 67% | Cost: $2.11 | Session: 1h 23m
```

Ctx(u): 67%	You've used 67% of context (Yellow Zone)
Cost: $2.11	API cost accumulated so far
Session: 1h 23m	Time elapsed in current session

Watch Ctx(u) Closely

This is your primary indicator. Take action before it reaches 70%.

Pro Tip: The 20% Rule

Reserve ~20% of context for end-of-session operations like multi-file changes, final corrections, or generating summaries.

03 Context Consumption & Symptoms

Token Costs by Action

Action	Context Cost	Example
Small file read	Low (~500)	config.json
Large file read	High (~5K)	bundle.js
Command execution	Medium (~1K)	npm test
Multi-file search	High (~3K+)	grep across repo
Long conversations	Accumulates	Extended debugging

Warning Signs

Symptom	Severity	Action
Shorter responses	🟡 Warning	Continue cautiously
Forgetting instructions	🟠 Serious	Document state
Inconsistencies	🔴 Critical	New session
"Can't access file" (already read)	🔴 Critical	Clear immediately

Context Recovery Commands

/compact

Summarizes conversation, preserves key context, ~50% reduction

Use when 70-85% full

/clear

Fresh start, loses all context, 100% reduction

Use when changing topics

/rewind

Undo recent changes, preserves context

Use for mistakes

03

Context Recovery Strategies

Three approaches for different scenarios

1 Compact

/compact

Summarizes conversation history

- ✓ Preserves key context
- ✓ -50% reduction in usage
- Fast operation

Use when: 70-85% context

2 Clear

/clear

Complete fresh start

- 100% context freed
- ✗ Loses all context
- Immediate relief

Use when: Changing topics

3 Targeted

Be Specific

Precision over brute force

- Specific queries
- Avoid "read entire file"
- Use symbol references

Use when: Precision needed

Decision Flowchart

Context >70%?
Check /status
→
Same task?
Continuing work
→
Use /compact
Preserve context
→
Use /clear
Fresh start

03

The 20% Reserve Rule & Context Poisoning

The 20% Reserve Rule

Always reserve ~20% of context for end-of-session operations:

- ✓ Multi-file operations at session end
- ✓ Last-minute corrections or fixes
- ✓ Generating summaries or checkpoints
- ✓ Error recovery or debugging

Practical Guideline

If context hits 80%, take action immediately

Context Poisoning (Bleeding)

When information from one task contaminates another:

Pattern 1: Style Bleeding

Task 1: "Create blue button" → Task 2: "Create form" → All buttons blue!

Pattern 2: Instruction Contamination

Conflicting instructions paralyze Claude

Pattern 3: Temporal Confusion

File renamed, but Claude uses old name

Context Hygiene Checklist

- ☑ New tasks = explicit markdown boundaries
- ☑ Structural changes = inform Claude explicitly
- ☑ Contradictory instructions = clarify priority
- ☑ Long session (>2h) = consider /clear

03 Plan Mode Deep Dive

"Look but don't touch" - safe exploration mode

What Plan Mode Allows

- Reading files and code
- Searching the codebase
- Analyzing architecture
- Proposing approaches
- Writing to plan files

What Plan Mode Prevents

- Editing existing files
- Running state-changing commands
- Creating new files
- Making commits

When to Use Plan Mode

Situation	Use Plan Mode?	Reason
Exploring unfamiliar codebase	✓ Yes	Safe exploration without risk
Investigating a bug	✓ Yes	Understand before modifying
Planning a new feature	✓ Yes	Design before implementing
Fixing a typo	✗ No	Direct edit is faster
Quick edit to known file	✗ No	Unnecessary overhead

03

Think Levels & Rewind Feature

Think Levels

Level	Flag	Tokens	Use Case
Standard	`--think`	~4K	Multi-component analysis
Deep	`--think-hard`	~10K	Architectural analysis
Maximum	`--ultrathink`	~32K	Critical system redesign

When to Use Each:

- **Think:** Features affecting 3+ files
- **Think Hard:** System-wide dependencies
- **Ultrathink:** Legacy modernization, complex debugging

Rewind Feature

What It Does

Undo mechanism for Claude's changes. Reverts file changes and restores previous state.

How to Use

```
/rewind
```

Or: "Undo the last change"

Limitations

- Only works on Claude's changes (not manual edits)
- Current session only
- Git commits NOT automatically reverted

Best Practice: Checkpoint Before Risk

Before risky operations, create a git checkpoint: `git commit -m "Checkpoint before experiment"`

04

Configuration System

Customizing Claude Code for your project and team

Hierarchy

Three-level configuration system

CLAUDE.md

Project instructions and conventions

.claude/ Folder

Complete configuration structure

Permissions

Security and access control

04

Configuration Hierarchy

Three levels with clear precedence rules

1

Local

/project/.claude/CLAUDE.md

Personal overrides, gitignored

2

Project

/project/CLAUDE.md

Team conventions, committed to git

3

Global

-/.claude/CLAUDE.md

Personal preferences, all projects

Precedence Rule: Local > Project > Global

Settings from higher levels override lower levels. Local configuration always wins.

04

CLAUDE.md Files in Practice

Global Level

~/.claude/CLAUDE.md

```
# Global Claude Code Settings
## Communication Style
- Be concise in responses
- Use code examples over explanations
## Preferred Tools
- TypeScript over JavaScript
- Prefer pnpm over npm
```

Personal preferences across all projects

Project Level

/project/CLAUDE.md

```
# Project: MyApp
## Tech Stack
- Next.js 14 with App Router
- TypeScript 5.3
- PostgreSQL with Prisma
## Code Conventions
- Use functional components
- File naming: kebab-case
```

Team conventions, committed to git

Local Level

/project/.claude/CLAUDE.md

```
# My Local Preferences
## Overrides
- Skip pre-commit hooks
- Use verbose logging
- Disable auto-formatting
```

Personal overrides, gitignored

Best Practices

- ✓ Keep it concise and focused
- ✓ Include concrete examples
- ✓ Update when conventions change
- ✓ Reference external documentation

Common Mistakes

- ✗ Writing long essays instead of concise rules
- ✗ Being vague about conventions
- ✗ Letting it go stale as project evolves
- ✗ Duplicating external documentation

04

The .claude/ Folder Structure

Complete configuration directory layout

Directory Tree

```
.claude/
├── CLAUDE.md # Personal instructions
├── settings.json # Hooks (committed)
├── settings.local.json # Permissions (gitignored)
├── agents/ # Custom agents
│   ├── backend-architect.md
│   └── code-reviewer.md
├── commands/ # Slash commands
│   ├── commit.md
│   └── pr.md
├── hooks/ # Event scripts
│   └── security-check.sh
├── rules/ # Auto-loaded conventions
└── skills/ # Knowledge modules
    └── security-guardian/
```

What to Commit vs Gitignore

Content Type	Location	Shared?
Team conventions	rules/	✓ Commit
Reusable agents	agents/	✓ Commit
Team commands	commands/	✓ Commit
Automation hooks	hooks/	✓ Commit
Knowledge modules	skills/	✓ Commit
Personal preferences	CLAUDE.md	✗ Gitignore
Personal permissions	settings.local.json	✗ Gitignore

04

Settings & Permissions Configuration

settings.local.json

```
{
  "permissions":{
    "allow":[
      "Bash(git *)",
      "Bash(pnpm *)",
      "Edit",
      "Write"
    ],
    "deny":[
      "Bash(rm -rf *)",
      "Bash(sudo *)"
    ],
    "ask":[
      "Bash(npm publish)"
    ]
  }
}
```

Personal permission overrides (gitignored)

allowedTools Alternative

```
{
  "allowedTools": [
    "Read(*)",
    "Bash(git status:*)",
    "Bash(pnpm *:*)",
    "TodoRead",
    "TodoWrite"
  ]
}
```

Granular tool control in ~/.claude.json

Progressive Permission Levels

Level 1 - Beginner

```
{
  "allowedTools": [
    "Read(*)",
    "Grep(*)",
    "Glob(*)"
  ]
}
```

Level 2 - Intermediate

```
{
  "allowedTools": [
    "Read(*)",
    "Grep(*)",
    "Bash(git:*)",
    "Bash(pnpm:*)",
    "TodoRead",
    "TodoWrite"
  ]
}
```

Level 3 - Advanced

```
{
  "allowedTools": [
    "Read(*)",
    "Edit(*)",
    "Write(*)",
    "Bash(git:*)",
    "Bash(pnpm:*)",
    "Task(*)",
    "TodoRead"
  ]
}
```

04

Permission Best Practices & Security

Critical Security Rules

1. Never use `--dangerously-skip-permissions` in production
2. Configure granular permissions with allowedTools
3. Use PreToolUse hooks for dangerous operations
4. Review all file changes before accepting
5. Create git checkpoints before risky operations
6. Check for secrets in .env files before committing

Horror Stories (Learn from Others)

The Great Deletion

rm -rf node_modules followed by rm -rf . (path error)

Database Destruction

DROP TABLE users in poorly generated migration

Force Push Disaster

git push --force to main unintentionally

Secret Leakage

Deletion of .env files with credentials

Managed Settings

- env: Environment variables
- cleanupPeriodDays: Transcript retention
- disableAllHooks: Block all hooks

Security Philosophy

- Treat Claude like untrusted intern
- Minimum necessary permissions
- Sandbox and audit regularly

05

Agents & Specialization

Creating specialized AI assistants for specific domains

What Are Agents?

Specialized AI sub-processes

File Structure

YAML frontmatter + markdown

Best Practices

Specialization over generalization

Advanced Patterns

Parallelization and composition

Key Takeaways

The 10 most important lessons for mastering Claude Code

1. Always review diffs before accepting changes
2. Use /compact before context gets critical (>70%)
3. Be specific in prompts (WHAT-WHERE-HOW-VERIFY)
4. Start with Plan Mode for complex/risky tasks
5. Create CLAUDE.md for every project
6. Configure granular permissions
7. Use agents for specialized tasks
8. Leverage hooks for automation
9. Commit frequently after each completed task
10. Keep learning and experimenting

Your Action Plan

Today

Install Claude Code, complete Day 1 checklist

This Week

Create project CLAUDE.md, practice context management

Week 2-3

Create custom agents, set up hooks and skills