Claude CLI

Custom Commands Guide

Complete Documentation for Creating and Using
Custom Slash Commands

Comprehensive Tutorial with Best Practices

October 23, 2025

Contents

1 Introduction **5**
- 1.1 What is Claude CLI? . . . 5
- 1.2 Purpose of This Guide . . . 5
- 1.3 Key Features . . . 5

2 Understanding Custom Commands **5**
- 2.1 What Are Custom Commands? . . . 5
- 2.2 Command Structure . . . 5
- 2.3 Command Locations and Namespaces . . . 6
- 2.4 Using Parameters with $ARGUMENTS . . . 6

3 Setting Up Your Environment **6**
- 3.1 Prerequisites . . . 6
- 3.2 Initial Setup . . . 7
  - 3.2.1 Step 1: Create the Commands Directory . . . 7
  - 3.2.2 Step 2: Verify Directory Creation . . . 7

4 Creating the Save Progress Command **7**
- 4.1 Overview . . . 7
- 4.2 Creating the Command File . . . 7
  - 4.2.1 Step 1: Open Your Text Editor . . . 7
  - 4.2.2 Step 2: Add the Command Content . . . 8
  - 4.2.3 Step 3: Save and Exit . . . 9
- 4.3 Understanding the Command Components . . . 9
  - 4.3.1 Component 1: CLAUDE.md Creation . . . 9
  - 4.3.2 Component 2: Git Integration . . . 9
  - 4.3.3 Component 3: User Feedback . . . 10

5 Creating the Resume Command **10**
- 5.1 Purpose . . . 10
- 5.2 Creating the Command File . . . 10
  - 5.2.1 Step 1: Open Your Text Editor . . . 10
  - 5.2.2 Step 2: Add the Command Content . . . 10
  - 5.2.3 Step 3: Save and Exit . . . 11
- 5.3 How Resume Works . . . 11

6 Using the Commands **11**
- 6.1 Your Complete Workflow . . . 11
  - 6.1.1 Starting a Work Session . . . 11
  - 6.1.2 Working on Your Project . . . 12
  - 6.1.3 Ending a Work Session . . . 12
  - 6.1.4 Exiting Claude CLI . . . 13
- 6.2 Verifying Commands Are Loaded . . . 13

7 Understanding CLAUDE.md **13**
- 7.1 What is CLAUDE.md? . . . 13
- 7.2 File Structure Explained . . . 13
- 7.3 Best Practices for CLAUDE.md . . . 14
- 7.4 Example CLAUDE.md . . . 14

8 Additional Useful Commands **15**
8.1 Quick Commit Command . . . 15
8.2 Code Review Command . . . 16
8.3 Test Command . . . 17
8.4 Fix Command . . . 18

9 Best Practices **18**
9.1 Command Organization . . . 19
9.2 Writing Effective Commands . . . 19
9.2.1 Be Clear and Specific . . . 19
9.2.2 Structure Instructions Logically . . . 19
9.2.3 Provide Context . . . 19
9.3 Version Control Best Practices . . . 20
9.3.1 Commit Project Commands . . . 20
9.3.2 Don't Commit Global Commands . . . 20
9.4 Maintenance and Updates . . . 20
9.4.1 Regular Review . . . 20
9.4.2 Command Documentation . . . 20

10 Troubleshooting **21**
10.1 Common Issues and Solutions . . . 21
10.1.1 Command Not Found . . . 21
10.1.2 Permission Denied Errors . . . 21
10.1.3 CLAUDE.md Not Created . . . 22
10.1.4 Commands Not Showing in /help . . . 22
10.2 Getting Help . . . 23
10.2.1 Check Claude Documentation . . . 23
10.2.2 Community Resources . . . 23

11 Advanced Topics **23**
11.1 Chaining Commands . . . 23
11.2 Using Commands with File References . . . 23
11.3 Creating Command Aliases . . . 24
11.4 Integrating with Git Hooks . . . 24
11.5 Using MCP (Model Context Protocol) . . . 24

12 Complete Workflow Example **24**
12.1 Scenario: Building a New Feature . . . 24
12.1.1 Day 1: Starting the Feature . . . 24
12.1.2 Day 2: Continuing Development . . . 25
12.1.3 Day 3: Completing the Feature . . . 25
12.2 Benefits of This Workflow . . . 26

13 Quick Reference **26**
13.1 Command Syntax . . . 26
13.2 File Locations . . . 26
13.3 Git Commands Used . . . 27
13.4 Keyboard Shortcuts . . . 27

14 Conclusion **27**

14.1 What You've Learned . . . 27
14.2 Next Steps . . . 27
14.2.1 Immediate Actions . . . 27
14.2.2 Future Enhancements . . . 28
14.3 Additional Resources . . . 28
14.4 Final Thoughts . . . 28

A Complete Command Templates **28**
A.1 Documentation Generator . . . 28
A.2 Refactor Command . . . 29
A.3 Security Audit Command . . . 30

B Version History **32**

C Glossary **32**

1 Introduction

1.1 What is Claude CLI?

Claude CLI (Command Line Interface), also known as Claude Code, is Anthropic's agentic coding tool that operates directly in your terminal. It uses the latest Claude models (Sonnet 4.5 and Opus 4.1) to help you write, debug, refactor, and manage code through natural language instructions.

1.2 Purpose of This Guide

This comprehensive guide will teach you how to:

- Create custom slash commands for Claude CLI
- Implement a progress-saving workflow
- Resume work seamlessly between sessions
- Understand command namespaces and organization
- Follow best practices for command development

1.3 Key Features

Why Custom Commands Matter

Custom commands allow you to:

- **Automate repetitive workflows** – Save time on common tasks
- **Maintain consistency** – Use standardized processes across projects
- **Share best practices** – Team members can use the same commands
- **Reduce cognitive load** – No need to remember complex instructions

2 Understanding Custom Commands

2.1 What Are Custom Commands?

Custom commands in Claude CLI are simple Markdown files that contain structured instructions. When you invoke a command, Claude reads the Markdown file and executes the instructions.

2.2 Command Structure

Every custom command follows this basic structure:

```
# Command Title
Brief description of what the command does

## Instructions
1. **Step One**
   - Detailed action
```

```
   - Additional context

2. **Step Two**
   - Next action
   - More details

## Notes
- Additional information
- Best practices
```

Listing 1: Basic Command Structure

2.3 Command Locations and Namespaces

Commands can be stored in two locations, each with its own namespace:

Location	Scope	Usage Format
`.claude/commands/`	Project-specific (only in current project)	`/project:command-name`
`~/.claude/commands/`	Global (all projects)	`/command-name`

Recommendation

For commands you'll use across multiple projects (like `save-progress`), create them in `~/.claude/commands/` as global commands. This makes them easier to remember and use.

2.4 Using Parameters with $ARGUMENTS

Commands can accept parameters using the special `$ARGUMENTS` variable:

```
# Fix Issue
Debug and fix: $ARGUMENTS

## Instructions
1. Understand the problem: $ARGUMENTS
2. Search for relevant files
3. Implement the fix
4. Test the solution
```

Listing 2: Parameterized Command Example

Usage:

```
> /fix login button not responding
```

3 Setting Up Your Environment

3.1 Prerequisites

Before creating custom commands, ensure you have:

- Claude CLI installed and configured

- A Claude Pro, Max subscription, or API access
- Git installed on your system
- Basic familiarity with command-line operations

3.2 Initial Setup

3.2.1 Step 1: Create the Commands Directory

Choose one of the following based on your needs:

```
# For commands available in all projects (recommended)
mkdir -p ~/.claude/commands
```

Listing 3: Create Global Commands Directory

```
# For project-specific commands only
mkdir -p .claude/commands
```

Listing 4: Create Project Commands Directory

3.2.2 Step 2: Verify Directory Creation

```
# Check global directory
ls -la ~/.claude/commands/

# Check project directory (if created)
ls -la .claude/commands/
```

Success Indicator

If the directory listing shows the folder without errors, you're ready to create commands!

4 Creating the Save Progress Command

4.1 Overview

The `save-progress` command is designed to save your current work session with complete context, making it easy to resume later. This command will:

1. Create or update a `CLAUDE.md` context file
2. Commit all changes to Git with a descriptive message
3. Display a formatted summary of your session
4. Provide clear instructions for resuming work

4.2 Creating the Command File

4.2.1 Step 1: Open Your Text Editor

```
nano ~/.claude/commands/save-progress.md
```

4.2.2 Step 2: Add the Command Content

Copy and paste the following complete command definition:

````
# Save Progress

Save current state for easy resume later.

## Instructions

1. **Create/Update CLAUDE.md**
   - Add timestamp
   - Summarize work completed this session
   - List files modified
   - Document next steps clearly
   - Note any blockers or decisions needed

2. **Git Commit**
   - Stage changes: `git add .`
   - Commit with message: "Progress: [session summary]"
   - Include WIP tag if incomplete: "WIP: [description]"

3. **Session Summary**
   Display to user:
   ```
   [checkmark] Progress Saved!

   [memo] Summary: [what was accomplished]
   [folder] Files Modified: [list]
   [next] Next Steps: [clear action items]
   [bookmark] Commit: [hash]

   To resume: Start new session with "/resume"
   or "@CLAUDE.md resume work"
   ```

## CLAUDE.md Format

When creating or updating CLAUDE.md, use this structure:

```
# Project: [Name]
Last Updated: [Timestamp]

## Current Sprint
- Goal: [what we're building]
- Progress: [X]%

## This Session ([Date])
[checkmark] Completed:
- [task 1]
- [task 2]

[construction] In Progress:
- [current task with details]

[next] Next:
````
````

```
1. [immediate next step]
2. [following step]

[warning] Blockers:
- [any issues]

## Tech Stack
[relevant technologies]
```

```

## Notes
- Use clear, descriptive commit messages
- Keep progress reports concise but informative
- Ensure all changes are tracked before committing
- CLAUDE.md is the primary source for resuming work
```

Listing 5: save-progress.md Complete Content

4.2.3 Step 3: Save and Exit

- Press `Ctrl + X`
- Press `Y` to confirm
- Press `Enter` to save

4.3 Understanding the Command Components

4.3.1 Component 1: CLAUDE.md Creation

The command creates a persistent context file that contains:

- **Project information** – Name and current sprint details
- **Session history** – What was accomplished in this session
- **Current status** – Work in progress
- **Next steps** – Clear action items for resuming
- **Blockers** – Any issues or decisions needed
- **Tech stack** – Technologies being used

4.3.2 Component 2: Git Integration

The command automatically:

- Stages all modified files
- Creates a descriptive commit message
- Uses "WIP:" prefix for incomplete work
- Records the commit hash for reference

4.3.3 Component 3: User Feedback

Provides clear visual feedback including:

- Summary of accomplishments
- List of modified files
- Next steps to take
- Commit reference
- Instructions for resuming

5 Creating the Resume Command

5.1 Purpose

The `resume` command reads your saved context and helps you pick up exactly where you left off. It intelligently analyzes multiple sources to give you a complete picture of your project state.

5.2 Creating the Command File

5.2.1 Step 1: Open Your Text Editor

```
nano ~/.claude/commands/resume.md
```

5.2.2 Step 2: Add the Command Content

```
# Resume Work

Resume from the last saved progress.

## Instructions

1. **Check Project Context**
   - Read CLAUDE.md if it exists
   - Review recent git commits: `git log --oneline -5`
   - Check current branch: `git branch --show-current`
   - See uncommitted changes: `git status`

2. **Analyze Current State**
   - Identify the last feature being worked on
   - Check for TODO comments in recent files
   - Look for failing tests
   - Review any error logs

3. **Provide Summary**
   - Summarize what was last being worked on
   - List completed tasks from CLAUDE.md
   - Highlight pending work
   - Suggest next immediate action based on "Next Steps"

4. **Ask for Direction**
```

```
   - "Should I continue with [last task from CLAUDE.md]?"
   - "Or would you like to work on something else?"

## Context Priority

Analyze sources in this order:
1. CLAUDE.md (primary source)
2. Recent git commits
3. Git branch names
4. TODO/FIXME comments
5. Uncommitted changes
```

Listing 6: resume.md Complete Content

5.2.3 Step 3: Save and Exit

Follow the same save process as before (Ctrl+X, Y, Enter).

5.3 How Resume Works

The resume command follows a systematic approach:

1. **Reads CLAUDE.md** – Gets the most recent session context
2. **Checks Git history** – Verifies what was actually committed
3. **Analyzes current branch** – Understands the feature context
4. **Scans for TODOs** – Finds explicit next steps in code
5. **Synthesizes information** – Creates a coherent summary
6. **Suggests actions** – Recommends what to do next

6 Using the Commands

6.1 Your Complete Workflow

6.1.1 Starting a Work Session

```
# Navigate to your project directory
cd /path/to/your/project

# Start Claude CLI
claude

# Resume from previous session
> /resume
```

Listing 7: Starting Claude CLI

Expected Output:

```
[clipboard] Resuming from last session (2025-10-23)

Last worked on: User Authentication Module (feature/auth branch)

[checkmark] Completed:
- Login endpoint with JWT
- Basic authentication middleware
- Unit tests for login

[construction] Current Status:
- 3 uncommitted changes in src/auth/
- Feature is 90% complete

[next] Next Steps:
1. Implement password reset email flow (highest priority)
2. Add rate limiting to auth endpoints
3. Write integration tests

Should I continue with the password reset flow, or would you
like to work on something else?
```

6.1.2 Working on Your Project

```
# Give Claude instructions
> Implement the password reset email flow

# Claude will start working...
# You can monitor progress or continue with other tasks
```

6.1.3 Ending a Work Session

```
# When you're ready to stop
> /save-progress
```

Listing 8: Saving Your Progress

Expected Output:

```
[checkmark] Progress Saved!

[memo] Summary: Implemented password reset email flow with token generation
[folder] Files Modified:
  - src/auth/password-reset.js
  - src/email/templates/reset-password.html
  - tests/auth/reset.test.js
[next] Next Steps:
  1. Add rate limiting to reset endpoint
  2. Test email delivery in staging
  3. Update API documentation
[bookmark] Commit: f3a9d2c
```

```
To resume: Start new session with "/resume" or "@CLAUDE.md resume work"
```

6.1.4 Exiting Claude CLI

```
# Exit the CLI
> exit

# Or press Ctrl+D
```

6.2 Verifying Commands Are Loaded

```
# Inside Claude CLI
> /help
```

Listing 9: Check Available Commands

You should see your custom commands listed:

```
Available Commands:
  /save-progress    Save current state for easy resume later
  /resume           Resume from the last saved progress
  /help             Show this help message
  ...
```

7 Understanding CLAUDE.md

7.1 What is CLAUDE.md?

`CLAUDE.md` is a persistent context file that lives in your project root. It serves as the "memory" between Claude CLI sessions, storing critical information about your project state.

7.2 File Structure Explained

```
# Project: [Your Project Name]
# Top-level identifier for the project

Last Updated: [Timestamp]
# Automatic timestamp from last save-progress

## Current Sprint
# High-level goals and progress tracking
- Goal: [What feature or milestone you're building]
- Progress: [Percentage complete]

## This Session ([Date])
# Detailed session information

[checkmark] Completed:
# Everything finished in this session
- [Specific task 1 with details]
- [Specific task 2 with details]
```

```

[construction] In Progress:
# Current work that's not yet complete
- [Task being worked on right now]
- [Include relevant details, blockers, or context]

[next] Next:
# Clear, actionable steps for the next session
1. [Immediate next action - most important]
2. [Following action]
3. [Additional steps if applicable]

[warning] Blockers:
# Anything preventing progress
- [Technical blocker or decision needed]
- [External dependency]

## Tech Stack
# Technologies used in this project
[List of frameworks, libraries, languages]
```

Listing 10: Annotated CLAUDE.md Structure

7.3 Best Practices for CLAUDE.md

Keeping CLAUDE.md Effective

- **Be specific** – "Added login endpoint" is better than "Worked on auth"
- **Update regularly** – Use `/save-progress` at natural stopping points
- **Keep it current** – Archive old sessions to a separate file if needed
- **Include context** – Note WHY decisions were made, not just WHAT was done
- **Track blockers** – Explicitly note what needs external input

7.4 Example CLAUDE.md

```
# Project: E-Commerce Platform
Last Updated: 2025-10-23 15:30:00

## Current Sprint
- Goal: Implement complete user authentication system
- Progress: 75%

## This Session (2025-10-23)
[checkmark] Completed:
- Password reset email flow with JWT tokens (30min expiry)
- Email template using company branding guidelines
- Integration tests covering happy path and error cases
- Updated API documentation with new endpoint

[construction] In Progress:
- Rate limiting for password reset endpoint
```

```
- Currently implementing token bucket algorithm
- Need to decide on limits: considering 3 requests per hour

[next] Next:
1. Complete rate limiting implementation
2. Add monitoring/alerting for abuse attempts
3. Deploy to staging and test email delivery
4. Schedule security review with team

[warning] Blockers:
- Need SendGrid API credentials for staging environment
- Waiting on design team for final email template approval

## Tech Stack
Node.js, Express.js, PostgreSQL, JWT, SendGrid, Jest, Supertest
```

Listing 11: Real-World CLAUDE.md Example

8 Additional Useful Commands

8.1 Quick Commit Command

Save time with automated commit messages:

```
nano ~/.claude/commands/commit.md
```

Listing 12: Create commit.md

```
# Quick Commit

Commit changes with a smart, auto-generated message: $ARGUMENTS

## Instructions
1. Run `git status` to see changes
2. Stage all changes with `git add .`
3. Generate a clear, conventional commit message based on:
   - Files modified
   - Nature of changes (fix, feature, refactor, etc.)
   - Additional context from $ARGUMENTS if provided
4. Commit with the generated message
5. Show the commit hash and summary

## Commit Message Format
Follow Conventional Commits:
- feat: New feature
- fix: Bug fix
- refactor: Code restructuring
- docs: Documentation changes
- test: Test additions or changes
- style: Formatting, no code change

## Notes
- If $ARGUMENTS provided, incorporate it into commit message
- Keep subject line under 50 characters
- Add detailed body if changes are significant
```

Listing 13: commit.md Content

Usage:

```
> /commit
# or
> /commit fixed authentication bug
```

8.2 Code Review Command

Perform self-review before committing:

```
nano ~/.claude/commands/review.md
```

Listing 14: Create review.md

```
# Code Review

Perform a thorough code review of recent changes.

## Instructions
1. **Identify Changes**
   - Run `git diff` for unstaged changes
   - Or review specific files if mentioned

2. **Analyze Code Quality**
   - Check adherence to coding standards
   - Look for common anti-patterns
   - Verify error handling
   - Check for code duplication

3. **Security Review**
   - Look for SQL injection vulnerabilities
   - Check for XSS vulnerabilities
   - Verify input validation
   - Check for exposed secrets

4. **Performance Analysis**
   - Identify potential bottlenecks
   - Check for N+1 query problems
   - Look for unnecessary computations
   - Review algorithm efficiency

5. **Testing Coverage**
   - Verify tests exist for new code
   - Check edge cases are covered
   - Ensure error cases are tested

6. **Provide Feedback**
   - Rate: Approved / Needs Work / Critical Issues
   - Give specific, actionable suggestions
   - Provide code examples where helpful
   - Prioritize issues by severity

## Review Checklist
```

```
- [ ] Code follows project style guide
- [ ] No security vulnerabilities
- [ ] Adequate test coverage
- [ ] No performance issues
- [ ] Documentation updated
- [ ] No TODO comments without tickets
```

Listing 15: review.md Content

8.3 Test Command

Run tests intelligently:

```
nano ~/.claude/commands/test.md
```

Listing 16: Create test.md

```
# Run Tests

Run project tests intelligently: $ARGUMENTS

## Instructions
1. **Detect Test Framework**
   - Check for Jest (package.json)
   - Check for pytest (Python)
   - Check for Go test (Go)
   - Check for other frameworks

2. **Determine Test Scope**
   - If $ARGUMENTS provided: run tests matching that pattern
   - If no args: detect recently changed files
   - Run only tests related to changed code

3. **Execute Tests**
   - Run appropriate test command
   - Capture output clearly
   - Show progress if long-running

4. **Analyze Results**
   - Display pass/fail summary
   - If failures: analyze error messages
   - Suggest fixes for common failure patterns

5. **Coverage Report**
   - Show test coverage if available
   - Highlight uncovered code
   - Suggest additional tests if needed

## Examples
- `/test` - Run relevant tests
- `/test user-auth` - Run auth-related tests
- `/test --all` - Run entire test suite
```

Listing 17: test.md Content

8.4 Fix Command

Debug and fix issues quickly:

```
nano ~/.claude/commands/fix.md
```

Listing 18: Create fix.md

```
# Fix Issue

Debug and fix the issue: $ARGUMENTS

## Instructions
1. **Understand the Problem**
   - Parse $ARGUMENTS for issue description
   - Ask clarifying questions if needed
   - Reproduce the issue if possible

2. **Investigate**
   - Search codebase for relevant files using grep
   - Check error logs
   - Review recent changes that might be related
   - Look for similar past issues

3. **Identify Root Cause**
   - Analyze the code path
   - Check for edge cases
   - Verify assumptions
   - Consider environment factors

4. **Implement Fix**
   - Make minimal necessary changes
   - Follow the principle of least surprise
   - Maintain code style consistency
   - Add comments explaining the fix

5. **Verify Fix**
   - Run related tests
   - Manually test the scenario
   - Check for regression
   - Verify edge cases

6. **Document**
   - Explain what was wrong
   - Describe how it was fixed
   - Note any future considerations
   - Update relevant documentation

## Notes
- Always verify the fix doesn't break existing functionality
- Consider adding a test to prevent regression
```

Listing 19: fix.md Content

9 Best Practices

9.1 Command Organization

Organizing Your Commands

Global Commands (`~/.claude/commands/`)
Use for:

- Workflow commands (save-progress, resume, commit)
- General development tasks (review, test, fix)
- Documentation generation

Project Commands (`.claude/commands/`)
Use for:

- Project-specific build processes
- Deployment workflows
- Custom project conventions

9.2 Writing Effective Commands

9.2.1 Be Clear and Specific

Less Effective	More Effective
"Check the code"	"Perform security audit checking for SQL injection, XSS, and exposed secrets"
"Fix bugs"	"Debug issue: $ARGUMENTS, identify root cause, implement minimal fix, verify with tests"
"Update docs"	"Generate API documentation with usage examples for modified functions"

9.2.2 Structure Instructions Logically

Always follow this pattern:

1. **Understand** – Gather context and requirements
2. **Analyze** – Investigate and identify issues
3. **Act** – Implement the solution
4. **Verify** – Test and validate
5. **Communicate** – Report results clearly

9.2.3 Provide Context

Include relevant information:

- Project conventions to follow
- Tools available in the environment

- Expected output format
- Error handling requirements

9.3 Version Control Best Practices

9.3.1 Commit Project Commands

```
# Add project commands to version control
git add .claude/commands/
git commit -m "feat: add project-specific Claude commands"
```

Why This Matters

When team members clone the repository, they automatically get your custom commands. This ensures consistent workflows across the entire team.

9.3.2 Don't Commit Global Commands

Global commands are personal and should not be added to version control:

```
# Add to .gitignore
echo "~/.claude/commands/" >> .gitignore
```

9.4 Maintenance and Updates

9.4.1 Regular Review

Periodically review your commands:

- Are they still relevant?
- Do they reflect current best practices?
- Are there repetitive patterns that could be automated?

9.4.2 Command Documentation

Create a README for your project commands:

```
nano .claude/commands/README.md
```

Document:

- Purpose of each command
- When to use it
- Any prerequisites
- Example usage

10 Troubleshooting

10.1 Common Issues and Solutions

10.1.1 Command Not Found

Problem

When you type `/save-progress`, Claude responds: "Unknown command: /save-progress"

Causes and Solutions:

1. **File doesn't exist**

```
# Check if file exists
ls -la ~/.claude/commands/save-progress.md

# If not found, create it
nano ~/.claude/commands/save-progress.md

```

2. **Wrong file extension**

```
# Check file extension
ls ~/.claude/commands/

# Rename if wrong
mv ~/.claude/commands/save-progress.txt \
   ~/.claude/commands/save-progress.md

```

3. **Claude CLI needs restart**

```
# Exit Claude
> exit

# Restart
claude

# Verify commands loaded
> /help

```

10.1.2 Permission Denied Errors

Problem

Error: "bash: permission denied" when creating files

Solution:

```
# Fix directory permissions
chmod 755 ~/.claude
chmod 755 ~/.claude/commands

```

```
# Fix file permissions
chmod 644 ~/.claude/commands/*.md

# Verify
ls -la ~/.claude/commands/
```

10.1.3 CLAUDE.md Not Created

Problem

After running `/save-progress`, no CLAUDE.md file appears

Possible Causes:

1. **Not in project root**

```
# Make sure you're in the project root directory
pwd
cd /path/to/your/project

```

2. **Write permissions issue**

```
# Test file creation
touch test.md
rm test.md

# If fails, check permissions
ls -la .

```

3. **Git not initialized**

```
# Initialize git if needed
git init

```

10.1.4 Commands Not Showing in /help

Problem

Custom commands don't appear when you run `/help`

Checklist:

Verify files are in correct directory

Check file extensions are `.md`

Restart Claude CLI

Ensure files have proper headers

```
# Debug command loading
cd ~/.claude/commands
ls -la
cat save-progress.md | head -5

# Restart Claude
exit
claude
> /help
```

10.2 Getting Help

10.2.1 Check Claude Documentation

```
# Official documentation
https://docs.claude.com/en/docs/claude-code
```

10.2.2 Community Resources

- GitHub: `hesreallyhim/awesome-claude-code`
- Anthropic's Best Practices Guide
- Community forums and Discord

11 Advanced Topics

11.1 Chaining Commands

You can execute multiple commands in sequence:

```
> /review then /test then /commit
```

Claude will:

1. Perform code review
2. Run tests
3. Commit changes if review passes and tests succeed

11.2 Using Commands with File References

Reference specific files while using commands:

```
> @src/auth/login.js /review
```

Claude will:

- Load the specified file into context
- Execute the review command on that file

11.3 Creating Command Aliases

For frequently used command combinations:

```
# Deploy Preparation

Complete pre-deployment checklist

## Instructions
1. Run code review: execute /review command
2. Run full test suite: execute /test --all
3. Check for security issues: execute /security
4. Update CHANGELOG
5. Bump version number
6. Create git tag
7. Generate deployment summary

## Notes
This is a meta-command that runs multiple other commands
```

Listing 20: Create deploy-prep.md

11.4 Integrating with Git Hooks

Commands can work with Git hooks for automation:

```
# .git/hooks/pre-commit
#!/bin/bash
claude -p "/review" || exit 1
```

11.5 Using MCP (Model Context Protocol)

Claude CLI can connect to MCP servers for extended functionality:

```
# Add GitHub MCP server
claude mcp add github

# Create command that uses GitHub API
nano ~/.claude/commands/create-pr.md
```

12 Complete Workflow Example

12.1 Scenario: Building a New Feature

Let's walk through a complete development workflow using our custom commands.

12.1.1 Day 1: Starting the Feature

```
# Start Claude CLI
claude

# Resume from last session (if applicable)
> /resume

```

```
# Start working on new feature
> Create a new user profile page with avatar upload

# ... Claude generates code ...

# Review the changes
> /review

# Run tests
> /test profile

# Save progress at end of day
> /save-progress
```

Result: CLAUDE.md created with session details, code committed.

12.1.2 Day 2: Continuing Development

```
# Start Claude
claude

# Resume work
> /resume

# Output shows:
# "Last session: User profile page with avatar upload (80% complete)
# Next: Add image validation and upload to S3"

# Continue with suggested next step
> Add image validation for avatar uploads - check file type,
  size, and dimensions

# ... work continues ...

# When bug appears
> /fix avatar upload fails with large images

# ... Claude debugs and fixes ...

# Verify fix
> /test avatar-upload

# Save progress
> /save-progress
```

12.1.3 Day 3: Completing the Feature

```
claude

> /resume

# Complete remaining work
> Finish the profile page implementation

```

```
# Final review
> /review

# Run all tests
> /test

# Prepare for deployment
> Update documentation for the new profile page feature

# Save final state
> /save-progress

# Create pull request (if using GitHub MCP)
> Create a pull request for the user profile feature
```

12.2 Benefits of This Workflow

Workflow Advantages

- **Context preservation** – Never lose track of where you were
- **Systematic approach** – Follow consistent quality processes
- **Documentation** – Automatic progress tracking
- **Team collaboration** – Easy handoff between developers
- **Quality assurance** – Built-in review and testing steps

13 Quick Reference

13.1 Command Syntax

Command	Usage
`/save-progress`	Save current session state
`/resume`	Resume from last saved state
`/commit`	Quick commit with auto-message
`/review`	Perform code review
`/test [pattern]`	Run tests (optionally filtered)
`/fix <description>`	Debug and fix an issue
`/help`	List all available commands
`/clear`	Clear conversation history

13.2 File Locations

Path	Purpose
`~/.claude/commands/`	Global commands (all projects)
`.claude/commands/`	Project-specific commands
`CLAUDE.md`	Project context file (root)

Path	Purpose
`.claude/settings.json`	Claude configuration

13.3 Git Commands Used

Command	Purpose
`git add .`	Stage all changes
`git commit -m "..."`	Commit with message
`git status`	Check working directory status
`git log -oneline -5`	Show recent commits
`git branch -show-current`	Display current branch

13.4 Keyboard Shortcuts

Shortcut	Action
`Ctrl + C`	Exit Claude CLI
`Ctrl + D`	Exit Claude CLI (alternative)
`Esc`	Stop current Claude operation
`Esc Esc`	Show message history
`Up Arrow`	Navigate to previous message
`Shift + Drag`	Reference file in Claude
`Ctrl + V`	Paste image from clipboard

14 Conclusion

14.1 What You've Learned

By following this guide, you now know how to:

- Create custom slash commands for Claude CLI
- Implement a complete progress-saving workflow
- Resume work seamlessly between sessions
- Organize commands for personal and team use
- Troubleshoot common issues
- Apply best practices for command development

14.2 Next Steps

14.2.1 Immediate Actions

1. Create the `save-progress` and `resume` commands
2. Test the workflow on a real project
3. Customize CLAUDE.md format to your needs
4. Share project commands with your team

14.2.2 Future Enhancements

- Add more commands from the examples provided
- Integrate with your CI/CD pipeline
- Create project-specific commands for your workflow
- Explore MCP servers for extended functionality
- Set up Git hooks for automation

14.3 Additional Resources

- Official documentation: `https://docs.claude.com`
- Best practices: `https://www.anthropic.com/engineering/claude-code-best-practices`
- Community commands: `https://github.com/hesreallyhim/awesome-claude-code`
- Support: `https://support.claude.com`

14.4 Final Thoughts

Custom commands transform Claude CLI from a simple coding assistant into a powerful, personalized development environment. By automating repetitive tasks and maintaining context between sessions, you can focus on what matters most: solving problems and building great software.

The key to success is starting simple, iterating based on your actual workflow, and continuously refining your commands as you discover new patterns and needs.

You're Ready!

You now have everything you need to create a professional, efficient workflow with Claude CLI. Start with the core commands (save-progress and resume), practice the workflow, and gradually expand your command library as you identify opportunities for automation. Happy coding!

A Complete Command Templates

This appendix contains complete, copy-paste-ready command templates for common development tasks.

A.1 Documentation Generator

```
# Generate Documentation

Create or update documentation for: $ARGUMENTS

## Instructions
1. **Identify Target**
   - If $ARGUMENTS specifies file(s), document those
   - Otherwise, scan recent git changes for undocumented code

```

```
2. **Generate Documentation**
   For functions/methods:
   - Parameter descriptions with types
   - Return value description
   - Usage examples
   - Error conditions

   For files/modules:
   - Purpose and responsibility
   - Public API overview
   - Usage examples
   - Dependencies

3. **Update Project Docs**
   - Update README.md if applicable
   - Add to API documentation
   - Update CHANGELOG.md
   - Create examples/ directory if needed

4. **Verify Completeness**
   - All public functions documented
   - Examples are runnable
   - Links are valid
   - Formatting is consistent

## Documentation Standards
- Use JSDoc, docstrings, or project standard
- Include at least one usage example
- Explain complex parameters
- Note any side effects
- Reference related functions

## Output
Show summary of:
- Files documented
- Functions/methods documented
- Examples added
- Documentation files updated
```

Listing 21: docs.md

A.2 Refactor Command

```
# Refactor Code

Improve code quality for: $ARGUMENTS

## Instructions
1. **Analyze Current Code**
   - Load target file from $ARGUMENTS
   - Identify code smells:
     * Long functions (>50 lines)
     * Duplicate code
     * Deep nesting (>3 levels)
     * Magic numbers/strings
```

```
     * Poor naming
     * Tight coupling

2. **Plan Refactoring**
   - List specific improvements
   - Prioritize by impact
   - Ensure changes maintain functionality

3. **Implement Improvements**
   Apply appropriate patterns:
   - Extract method for long functions
   - Remove duplication (DRY)
   - Simplify conditionals
   - Improve naming
   - Add constants for magic values
   - Reduce coupling

4. **Verify Correctness**
   - Run existing tests
   - Add new tests if needed
   - Check edge cases still work
   - Verify no regression

5. **Document Changes**
   - Explain what was refactored
   - Note any behavior changes
   - Update relevant documentation

## Refactoring Principles
- Make small, incremental changes
- Keep tests passing
- Don't change behavior unnecessarily
- Improve readability
- Reduce complexity

## Output Format
Before/After comparison
List of improvements made
Test results
Recommendation for next refactoring
```

Listing 22: refactor.md

A.3 Security Audit Command

```
# Security Audit

Perform security review of: $ARGUMENTS

## Instructions
1. **Code Analysis**
   Check for vulnerabilities:

   **Injection Attacks:**
   - SQL injection (raw queries)
```

```
   - NoSQL injection
   - Command injection
   - XSS (Cross-Site Scripting)
   - LDAP injection

   **Authentication/Authorization:**
   - Weak password policies
   - Missing authentication
   - Broken access control
   - Session management issues
   - JWT vulnerabilities

   **Data Exposure:**
   - Sensitive data in logs
   - Exposed API keys/secrets
   - Unencrypted sensitive data
   - Information disclosure

   **Configuration:**
   - Default credentials
   - Debug mode in production
   - Unnecessary services enabled
   - Improper error handling

2. **Dependency Check**
   - Scan package.json/requirements.txt
   - Check for known vulnerabilities
   - Identify outdated dependencies
   - Review license compliance

3. **Secret Scanning**
   - Search for hardcoded passwords
   - Look for API keys in code
   - Check for exposed tokens
   - Verify .env files in .gitignore

4. **Input Validation**
   - Check all user inputs
   - Verify sanitization
   - Test boundary conditions
   - Validate file uploads

5. **Report Generation**
   Format:
   - Severity: Critical/High/Medium/Low
   - Description of vulnerability
   - Affected code location
   - Remediation steps
   - References (CVE, OWASP)

## Severity Definitions
Critical: Immediate exploitation possible
High: Significant security risk
Medium: Moderate risk, requires conditions
Low: Minor risk or best practice

```

```
## Output
Prioritized list of findings
Remediation recommendations
Compliance checklist (OWASP Top 10)
```

Listing 23: security.md

B Version History

Version	Date	Changes
1.0	2025-10-23	Initial release with core commands (save-progress, resume)

C Glossary

Claude CLI Command-line interface for Claude Code, Anthropic's agentic coding tool

Slash Command
Custom commands invoked with forward slash (/) prefix

CLAUDE.md Context file storing project state and session information

Namespace Organizational prefix for commands (e.g., /project:, /dev:)

$ARGUMENTS
Special variable in commands that accepts user-provided parameters

MCP Model Context Protocol - standard for connecting AI to external tools

Agentic Coding
AI-driven development where the assistant takes autonomous actions

Context Window
Amount of conversation history Claude can reference

Session A single continuous period of work in Claude CLI