CLAUDE CODE

Slash Commands

A Progressive Tutorial for Data Scientists & AI Engineers

From built-in essentials to custom ML/AI workflows

Version 1

Feb 2026

Created by Ajit Jaokar

(using Claude Opus 4.6)

February 2026

Built for Claude Code CLI, VS Code Extension, Desktop App & Web

1. Why Slash Commands Matter

Claude Code is Anthropic's agentic coding tool that runs in your terminal, IDE, browser, and as a desktop app. It reads your codebase, edits files, runs commands, and integrates with your development tools. At the heart of efficient interaction with Claude Code are slash commands—prefixed with a forward slash (/) and typed directly into your session to trigger specific actions.

1.1 What Are Slash Commands?

Slash commands are shortcuts you type during a Claude Code session to control behavior, manage context, automate workflows, and invoke custom routines. They fall into two categories:

Built-in commands are provided by Claude Code out of the box—things like /clear, /compact, /model, and /help. These handle session management, context optimization, and navigation.

Custom commands (Skills) are markdown files you create that become slash commands. A file at .claude/skills/review/SKILL.md creates the /review command. These encode your team's workflows, coding standards, and domain-specific procedures into reusable, shareable routines.

1.2 Why They're Significant for DS/AI Engineers

Data scientists and AI engineers face unique workflow challenges: long-running training jobs, complex data pipelines, model evaluation loops, and constant iteration between notebooks and production code. Slash commands address these by providing:

Context efficiency. ML projects involve massive codebases and long conversations. Commands like /compact and /clear prevent context window overflow, keeping Claude's responses sharp.

Workflow automation. Repetitive tasks—running experiment sweeps, generating evaluation reports, formatting model cards—become single commands.

Team standardization. Project-level commands in .claude/skills/ enforce consistent practices across your ML team: code review standards, documentation templates, and deployment checklists.

Model-aware automation. Skills can specify which Claude model to use. Route quick data cleaning to Haiku for speed, and complex architecture decisions to Opus for depth.

Key Insight:
Slash commands are user-invoked (you decide when to run them). Skills can also be model-invoked—Claude automatically uses them when the task matches the skill's description. This distinction is critical for building intelligent workflows.

2. Built-in Slash Commands—The Essentials

These commands come with every Claude Code installation. Master them first before building custom commands.

Command	What It Does
`/help`	Shows all available slash commands including your custom ones and MCP server commands
`/clear`	Wipes conversation history for a fresh start. Use often—every new task deserves clean context
`/compact`	Compresses conversation history while preserving key details. Add instructions: /compact focus on data pipeline logic
`/model`	Switch between models mid-session (Sonnet 4.5, Haiku 4.5, Opus 4.5). Use Haiku for quick tasks, Opus for complex reasoning
`/context`	Visualizes context window usage as a colored grid—your “fuel gauge” for token consumption
`/cost`	Shows token usage and cost for the current session
`/status`	Displays version info and connectivity status for troubleshooting
`/memory`	Opens your CLAUDE.md memory files for editing project-specific instructions
`/init`	Walks you through creating a CLAUDE.md for your project—essential for new repos
`/config`	Opens settings for customization (prompt suggestions, model defaults, etc.)
`/permissions`	Manage tool approval settings—allow trusted commands to run without prompts
`/resume`	Opens a conversation picker to continue previous sessions by name or content
`/login`	Switch between accounts (useful for personal vs. work API keys)
`/review`	Triggers a code review workflow on your recent changes
`/hooks`	Interactive interface for defining lifecycle hooks (pre-edit, post-edit, etc.)
`/terminal-setup`	Installs Shift+Enter key binding for your terminal (VS Code, Alacritty, etc.)
`/doctor`	Checks installation health and diagnoses common issues
`/agents`	Manage subagents—specialized Claude instances for domain-specific tasks

Tip:
Use /clear between every distinct task. You don’t need old context eating your tokens, and you avoid Claude running compaction calls to summarize stale conversations.

3. Essential Keyboard Shortcuts

Slash commands are only half the story. These shortcuts complement them and dramatically speed up your workflow:

Shortcut	Action
`Shift+Tab`	Cycle through permission modes: Normal → Auto-Accept → Plan Mode
`Ctrl+C / Escape`	Ctrl+C exits Claude Code entirely. Use Escape to stop the current action
`Escape × 2`	Double-press Escape to see a list of all previous messages you can jump back to
`Tab`	Accept a prompt suggestion
`Ctrl+R`	Reverse search through command history
`Ctrl+O`	Toggle verbose mode (shows extended thinking)
`Up Arrow`	Navigate back through past prompts, even from previous sessions
`! command`	Run a shell command directly (e.g., !nvidia-smi) without Claude's conversational overhead
`# text`	Quickly add a memory to CLAUDE.md without opening the editor
`@ path`	Reference a file or directory in your prompt for targeted context

4. Creating Custom Slash Commands

Custom commands are where slash commands become truly powerful. They're markdown files that encode instructions Claude follows when you invoke them.

4.1 Where Commands Live

Scope	Location	Visibility
`Personal`	~/.claude/skills/	Available across all your projects
`Project`	.claude/skills/	Shared via git with your team
`Plugin`	Via /plugin install	Community-shared packages

Legacy .claude/commands/ files still work, but skills in .claude/skills/ are recommended as they support additional features like bundled scripts, frontmatter for invocation control, and auto-discovery by Claude.

4.2 Anatomy of a Skill File

Every skill needs a SKILL.md file with YAML frontmatter and markdown instructions:

```
# File: ~/.claude/skills/my-command/SKILL.md
---
name: my-command
description: What this command does (helps Claude auto-discover it)
allowed-tools: Bash(python *), Read, Write
model: sonnet                    # optional: route to specific model
disable-model-invocation: true   # optional: only YOU can invoke
argument-hint: [arg1] [arg2]     # optional: shows usage hint
---

# Instructions for Claude
When invoked, do the following:
1. Step one...
2. Step two...

Use $ARGUMENTS to access user-provided arguments.
```

4.3 Key Frontmatter Fields

Field	Purpose
`name`	Becomes the /slash-command name
`description`	Helps Claude decide when to auto-invoke the skill
`allowed-tools`	Tools Claude can use without per-use approval (e.g., Bash, Read, Write, Grep)
`model`	Route to a specific model: sonnet, haiku, or opus
`disable-model-invocation`	Set true to prevent Claude from auto-invoking (user-only)
`user-invocable`	Set false to hide from menu (model-only, background knowledge)
`argument-hint`	Usage hint shown in help (e.g., [file-path] [--verbose])

4.4 Shell Injection with !

You can embed shell commands in your skill files. Claude runs them first, captures the output, and injects it into the prompt context before processing:

```
# In your SKILL.md:
Current git status:
!git status

Recent changes:
!git diff --stat HEAD~3

Now analyze these changes and...
```

This is how commands like /commit know what you changed—they run !git diff automatically before generating a commit message.

5. Slash Commands for Data Scientists

Now let's build commands specifically designed for data science and ML workflows. These progress from simple utilities to sophisticated multi-step automations.

5.1 Data Profiling Command

Purpose: Quick EDA on any dataset

```
# .claude/skills/data-profile/SKILL.md
---
name: data-profile
description: Profile a dataset with summary statistics, missing values,
  distributions, and data quality checks
allowed-tools: Bash(python *), Read, Write
model: sonnet
argument-hint: [file-path]
disable-model-invocation: true
---

# Data Profiling

Profile the dataset at $ARGUMENTS:

1. Load the data (CSV, Parquet, or JSON)
2. Report: row count, column count, dtypes, memory usage
3. For each column: nulls, unique values, min/max/mean/median/std
4. Flag columns with >20% missing values
5. Flag potential data leakage (features highly correlated with target)
6. Show distribution shape (skewness) for numeric columns
7. Suggest preprocessing steps based on findings
8. Save a profile report as data_profile.md
```

Usage: `/data-profile data/train.csv`

5.2 Experiment Tracker Command

Purpose: Log ML experiments consistently

```
# .claude/skills/log-experiment/SKILL.md
---
name: log-experiment
description: Log an ML experiment with hyperparameters, metrics, and notes
allowed-tools: Bash(python *), Read, Write
```

```
disable-model-invocation: true
argument-hint: [experiment-name]
---

# Log Experiment

Current experiments log:
!cat experiments/log.jsonl 2>/dev/null || echo 'No log yet'

Create or append to experiments/log.jsonl with:
- experiment_name: $ARGUMENTS
- timestamp: current UTC time
- git_commit: current HEAD sha
- Ask user for: model_type, key hyperparameters, metrics
- Save as a JSON line
- Print a summary table of all experiments so far
```

Usage: `/log-experiment baseline-xgboost-v2`

5.3 Model Evaluation Report

Purpose: Generate standardized evaluation reports

```
# .claude/skills/eval-report/SKILL.md
---
name: eval-report
description: Generate a model evaluation report with metrics, plots,
  and error analysis
allowed-tools: Bash(python *), Read, Write
model: opus
disable-model-invocation: true
argument-hint: [predictions.csv] [ground-truth.csv]
---

# Model Evaluation Report

Generate a comprehensive evaluation report:

1. Load predictions and ground truth from $ARGUMENTS
2. Compute: accuracy, precision, recall, F1, AUC-ROC (classification)
   or MSE, RMSE, MAE, R-squared (regression)
3. Generate confusion matrix (classification) or residual plot (regression)
4. Identify worst-performing segments/slices
5. Compare against baseline if experiments/log.jsonl exists
6. Output eval_report.md with all findings
7. Flag any metric degradation vs. previous best
```

5.4 Notebook-to-Script Converter

Purpose: Convert Jupyter notebooks to production code

```
# .claude/skills/nb2script/SKILL.md
---
name: nb2script
description: Convert a Jupyter notebook to a clean, production-ready
  Python script with proper structure
allowed-tools: Bash(python *), Read, Write, Grep
model: sonnet
disable-model-invocation: true
argument-hint: [notebook.ipynb]
---

# Notebook to Production Script

Convert $ARGUMENTS to production Python:

1. Parse the .ipynb JSON
2. Remove cells that are purely exploratory (plots, prints, EDA)
3. Extract imports, consolidate at top
4. Wrap logic in functions with type hints and docstrings
5. Add argparse for configurable parameters
6. Add logging instead of print statements
7. Add if __name__ == '__main__' block
8. Run a lint check with ruff
9. Save as src/{notebook_name}.py
```

6. Slash Commands for AI Engineers

These commands target the unique needs of AI/ML engineers working on model training, deployment, and infrastructure.

6.1 Training Monitor

Purpose: Parse and summarize training logs

```
# .claude/skills/training-monitor/SKILL.md
---
name: training-monitor
description: Analyze training logs for convergence, anomalies, and
  suggestions. Use when reviewing training run output.
allowed-tools: Bash(python *), Read, Grep
model: sonnet
argument-hint: [log-file-or-directory]
---

# Training Monitor
```

```
Analyze training logs at $ARGUMENTS:

1. Parse loss curves (train and validation)
2. Detect: overfitting, underfitting, loss plateaus, NaN/Inf values
3. Report learning rate schedule effectiveness
4. Flag if validation loss hasn't improved in N epochs
5. Suggest: early stopping point, LR adjustments, regularization
6. Estimate time to convergence based on current trajectory
7. Output a concise training_analysis.md
```

6.2 Model Card Generator

Purpose: Auto-generate ML model cards for documentation

```
# .claude/skills/model-card/SKILL.md
---
name: model-card
description: Generate a model card following industry standards
allowed-tools: Bash(python *), Read, Write, Grep, Glob
model: opus
disable-model-invocation: true
argument-hint: [model-directory]
---

# Model Card Generator

Scan $ARGUMENTS and generate a model card:

1. Identify model architecture from code/config files
2. Extract training data details, preprocessing steps
3. Pull metrics from evaluation logs or experiment tracker
4. Document: intended use, limitations, ethical considerations
5. List dependencies and hardware requirements
6. Follow the format from Mitchell et al. (2019)
7. Save as MODEL_CARD.md in the model directory
```

6.3 GPU/Resource Monitor

Purpose: Quick infrastructure health check

```
# .claude/skills/gpu-check/SKILL.md
---
name: gpu-check
description: Check GPU utilization, memory, and running processes
allowed-tools: Bash(nvidia-smi *), Bash(ps *), Bash(df *), Read
model: haiku
---

# GPU & Resource Check

!nvidia-smi --query-gpu=index,name,utilization.gpu,memory.used,
```

```
  memory.total,temperature.gpu --format=csv,noheader
!df -h /data /models 2>/dev/null
!ps aux | grep -E 'python|train' | head -10

Summarize:
- GPU utilization and memory for each device
- Disk space warnings if any mount is >85% full
- Running training/inference processes
- Recommendations if resources are underutilized
```

Tip:
This command uses model: haiku for near-instant responses. Not every command needs Opus—match the model to the complexity.

6.4 Security & Dependency Audit

Purpose: Scan ML projects for common vulnerabilities

```
# .claude/skills/ml-security-scan/SKILL.md
---
name: ml-security-scan
description: Scan ML project for security issues: pickle exploits,
  exposed credentials, unsafe deserialization, dependency CVEs
allowed-tools: Bash(python *), Read, Grep, Glob
model: opus
disable-model-invocation: true
---

# ML Security Scan

Scan the current project for ML-specific security issues:

1. Check for pickle/joblib usage (deserialization attacks)
2. Scan for hardcoded API keys, tokens, credentials
3. Review model loading: torch.load() without weights_only=True
4. Check requirements.txt/pyproject.toml for known CVEs
5. Verify .gitignore excludes: .env, model weights, data files
6. Check for eval() or exec() in data processing pipelines
7. Report findings with severity levels and fix suggestions
```

6.5 Pipeline Debugger

Purpose: Diagnose failing ML pipelines

```
# .claude/skills/debug-pipeline/SKILL.md
---
name: debug-pipeline
description: Debug a failing ML pipeline by tracing data flow,
```

```
  checking shapes, and identifying breaking changes
allowed-tools: Bash(python *), Read, Grep, Glob
model: opus
argument-hint: [error-message-or-file]
---

# Pipeline Debugger

Debug the ML pipeline issue described in $ARGUMENTS:

1. Trace the data flow from source to failure point
2. Check tensor/array shape mismatches at each stage
3. Verify dtype compatibility (float32 vs float16 vs bfloat16)
4. Check for NaN/Inf propagation in intermediate outputs
5. Review recent git changes that might have caused the break
6. Suggest targeted fixes ranked by likelihood
7. Provide a minimal reproduction script if possible
```

7. Advanced Patterns & Best Practices

7.1 Combining Skills with Subagents

For complex workflows, skills can delegate to specialized subagents. Define subagents in .claude/agents/ and reference them in your skills:

```
# .claude/agents/data-validator.yaml
name: data-validator
description: Expert data validation agent
model: sonnet
prompt: |
  You are a data validation specialist. Check schemas,
  distributions, and integrity constraints.
tools: [Read, Bash, Grep]
```

Then in your skill, instruct Claude to delegate validation work to the data-validator agent. Claude will spin up the subagent with its own context window, preserving your main session's tokens.

7.2 Hooks for Automated Quality

Hooks run shell commands before or after Claude Code actions. Combine them with slash commands for automated quality gates:

```
# .claude/settings.json (excerpt)
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write(*.py)",
```

```
        "hooks": [
          { "type": "command", "command": "ruff check $file" },
          { "type": "command", "command": "mypy $file" }
        ]
      }
    ]
  }
}
```

Every time Claude writes a Python file, ruff and mypy run automatically. No slash command needed—it's baked into the workflow.

7.3 MCP Servers for External Data

Model Context Protocol (MCP) servers connect Claude to external tools. For data scientists, this means direct access to databases, cloud storage, and experiment tracking platforms:

```
# Add an MCP server for database access
claude mcp add --transport http postgres-mcp https://your-mcp-server.com

# Now Claude can query your database directly
# MCP servers can also provide their own slash commands
```

7.4 CLAUDE.md for ML Projects

Your CLAUDE.md file is loaded at the start of every session. For ML projects, include critical context so Claude doesn't waste tokens rediscovering your stack:

```
# CLAUDE.md

## Project: fraud-detection-v3
- Framework: PyTorch 2.2 + Lightning
- Data: /data/processed/ (Parquet, ~50M rows)
- Models: /models/ (checkpoints in SafeTensors format)
- Experiments: tracked in experiments/log.jsonl

## Commands
- Train: python src/train.py --config configs/base.yaml
- Evaluate: python src/evaluate.py --checkpoint models/latest.pt
- Test: pytest tests/ -x -q

## Conventions
- All tensors use bfloat16 on GPU, float32 on CPU
- Feature names follow: {source}_{transform}_{name}
- Commits follow conventional commits format
```

7.5 Scaling: Plugins and Marketplaces

Once you've built a collection of useful skills, package them as a plugin to share with your team or the community:

1. Structure your skills as a plugin repository on GitHub
2. Others install via /plugin marketplace add <your-repo-url>
3. Plugin skills are namespaced (plugin-name:skill-name) to avoid conflicts

8. Quick Reference Card

A concise summary of the most useful commands and patterns for daily work:

Session Management

Command	When to Use
`/clear`	Fresh start—use between every distinct task
`/compact`	Compress context when running long. Add focus: /compact keep training logic
`/resume`	Pick up a previous session by name or content search
`/model haiku`	Switch to Haiku for quick tasks. /model opus for complex reasoning
`/context`	Check how full your context window is

Workflow Commands (Create These)

Command	Purpose
`/data-profile`	Quick EDA on any dataset
`/log-experiment`	Consistent experiment logging
`/eval-report`	Standardized model evaluation
`/nb2script`	Convert notebooks to production code
`/training-monitor`	Analyze training logs for issues
`/model-card`	Generate model documentation
`/gpu-check`	Infrastructure health check
`/ml-security-scan`	Security audit for ML projects
`/debug-pipeline`	Diagnose failing ML pipelines

9. Getting Started Checklist

1. Install Claude Code: npm install -g @anthropic-ai/claude-code
2. Run /init in your project to create CLAUDE.md with your ML stack details

3. Run /help to see all available built-in commands
4. Create your first custom skill: mkdir -p .claude/skills/data-profile/ and add SKILL.md
5. Test it: type /data-profile path/to/dataset.csv in your Claude Code session
6. Commit .claude/skills/ to your repo so your team gets the same commands
7. Iterate: start with one command you use daily, then build your library over time

Note:
For the latest documentation, visit code.claude.com/docs. Slash commands have recently been merged into the broader “skills” system—your existing .claude/commands/ files still work, but new skills should use .claude/skills/ for the full feature set.