Claude Code - Complete Official Documentation

> ***Source***: *This documentation is compiled directly from the official Anthropic Claude Code documentation at https://docs.anthropic.com/en/docs/claude-code/*
>
> ***Last Updated***: *Based on official documentation as of January 2025*

Table of Contents

Overview

Claude Code is an agentic coding tool that lives in your terminal, understands your codebase, and helps you code faster through natural language commands. By integrating directly with your development environment, Claude Code streamlines your workflow without requiring additional servers or complex setup.

Key Capabilities

Claude Code's comprehensive capabilities include:

Core Development Tasks

- **File Editing & Bug Fixing**: Edit files and fix bugs across your codebase
- **Code Understanding**: Answer questions about your code's architecture and logic
- **Testing & Commands**: Execute and fix tests, linting, and other commands
- **Git Operations**: Search through git history, resolve merge conflicts, and create commits and PRs

Advanced Features

- **Comprehensive Tool Set**: File operations, code search, web browsing, and more
- **Enterprise Integration**: Works with Amazon Bedrock and Google Vertex AI
- **Security First**: Direct API connection without intermediate servers
- **Context Awareness**: Maintains understanding of your entire project

---

Installation & Setup

System Requirements

- **Operating Systems**:
  - macOS 10.15+
  - Ubuntu 20.04+/Debian 10+
  - Windows via WSL

Installation Steps

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

# Launch Claude Code
claude
```

Important: Do NOT use `sudo npm install -g` as this can lead to permission issues and security risks.

Windows (WSL) Setup

Currently, Claude Code does not run directly in Windows and requires WSL:

```
# If you encounter OS/platform detection issues
# WSL may be using Windows npm. Check with:
which npm
which node
# These should point to Linux paths starting with /usr/ not /mnt/c/
```

Terminal Configuration

Theme Matching

Claude cannot control your terminal theme. Match Claude Code's theme to your terminal:

- During onboarding
- Any time via the `/config` command

Line Breaks

You have several options for entering linebreaks:

- **Quick escape**: Type `\` followed by Enter
- **Keyboard shortcut**: Press Option+Enter (Meta+Enter)
- **iTerm2/VSCode users**: Run `/terminal-setup` for Shift+Enter configuration

Notifications

Configure notifications to never miss when Claude completes a task:

- **macOS**: Enable notification permissions in System Settings → Notifications → [Your Terminal App]
- **Note**: Notifications are specific to iTerm2 and not available in default macOS Terminal

Long Content Handling

When working with extensive code or long instructions:

- **Avoid direct pasting**: Claude Code may struggle with very long pasted content
- **Use file-based workflows**: Write content to a file and ask Claude to read it
- **VS Code limitations**: The VS Code terminal is particularly prone to truncating long pastes

Vim Keybindings

Claude Code supports a subset of Vim keybindings:

- Enable with `/vim` or configure via `/config`
- Supported editing commands: `x`, `dw/de/db/dd/D`, `cw/ce/cb/cc/C`, `.`

Authentication Methods

Claude Code supports multiple authentication methods:

1. Anthropic Console (Default)

- Connect through the Anthropic Console
- Complete the OAuth process
- Requires active billing at console.anthropic.com

2. Claude App (Pro or Max Plan)

- Subscribe to Claude's Pro or Max plan
- Unified subscription for both Claude Code and web interface
- Log in with your Claude.ai account

3. Enterprise Platforms

Configure Claude Code for enterprise deployments:

- **Amazon Bedrock**: See Bedrock configuration
- **Google Vertex AI**: See Vertex AI configuration

4. API Key Authentication

For SDK usage:

```
# Create an Anthropic API key in the Console
# Set environment variable
export ANTHROPIC_API_KEY=your-api-key
```

Getting Started

Quick Start

1. **Launch Claude Code**:

```
claude
```

2. **Understand your codebase**:

```
> What does this project do?
> Explain the architecture of this application
```

3. **Make changes**:

```
> Add error handling to the API endpoints
> Create a new component for user authentication
```

Claude Code reads your files as needed - you don't have to manually add context.

Permission System

Claude Code always asks for permission before modifying files. You can:

- Approve individual operations
- Set up allowed tools for automatic approval
- Configure permissions via `/permissions` command

Core Features & Tools

Claude Code provides a comprehensive set of tools:

File Operations

- **Read**: View file contents
- **Write**: Create or modify files
- **List**: Browse directory structures
- **Search**: Find code across your project

Code Execution

- **Bash**: Run shell commands
- **Test execution**: Run and analyze test results
- **Linting**: Execute code quality checks

Version Control

- **Git operations**: Commits, branches, merges
- **History analysis**: Search through git history
- **PR creation**: Generate pull requests

Web Integration

- **Web search**: Search for documentation and solutions
- **Web fetch**: Retrieve content from URLs

Commands Reference

Interactive REPL Commands

Based on the official documentation, the following commands are available:

Command	Description	Source
`/config`	Configure Claude Code settings and preferences	Multiple mentions in docs
`/permissions`	View and manage Claude Code's tool permissions	"You can view & manage Claude Code's tool permissions with /permissions"
`/allowed-tools`	Configure allowed tools	"Permission rules can be configured using /allowed-tools"
`/hooks`	Manage hooks for automating actions	"Run the /hooks slash command"
`/mcp`	Manage Model Context Protocol servers	"Check MCP server status any time using the /mcp command"
`/vim`	Toggle Vim mode keybindings	"Enable vim-style editing with /vim command"
`/terminal-setup`	Configure terminal shortcuts	"Run /terminal-setup within Claude Code"
`/bug`	Report bugs	"Report bugs directly with the /bug command"
`/ide`	Connect to IDE from external terminal	"use the /ide command in any external terminal to connect to the IDE"
`/cost`	View current session usage	"Use /cost to see current session usage"
`/status`	Verify configuration	"Use the /status slash command to verify your configuration"
`/install-github-app`	Set up GitHub app	"Just open claude and run /install-github-app"

Note: The `--continue` and `--resume` are CLI flags for launching Claude Code, not interactive slash commands within the REPL.

Slash Commands

Control Claude's behavior during an interactive session with slash commands.

Built-in System Commands

See Interactive REPL Commands above for system commands like `/config`, `/permissions`, etc.

Custom Slash Commands

Create custom commands as Markdown files that Claude Code can execute:

Project Commands

Commands stored in your repository and shared with your team:

```
# Create a project command
mkdir -p .claude/commands
echo "Analyze this code for performance issues and suggest optimizations:" >
```

```
.claude/commands/optimize.md

# Use in Claude
> /project:optimize
```

Personal Commands

Commands available across all your projects:

```
# Create a personal command
mkdir -p ~/.claude/commands
echo "Review this code for security vulnerabilities:" > ~/.claude/commands/security-review.md

# Use in Claude
> /user:security-review
```

Namespaced Commands

Organize commands in subdirectories:

```
# Create namespaced command
mkdir -p .claude/commands/frontend
echo "Create a new React component" > .claude/commands/frontend/component.md

# Use in Claude
> /project:frontend:component
```

Dynamic Commands with Arguments

Pass dynamic values using the `$ARGUMENTS` placeholder:

```
# Command definition
echo 'Fix issue #$ARGUMENTS following our coding standards' > .claude/commands/fix-issue.md

# Usage
> /project:fix-issue 123
```

MCP Server Commands

MCP servers can expose prompts as slash commands:

```
# Without arguments
> /mcp__github__list_prs

# With arguments
> /mcp__github__pr_review 456
> /mcp__jira__create_issue "Bug title" high
```

Advanced Features

Execute Bash Commands Before Slash Command

Use the `!` prefix to execute bash commands and include output: ```markdown

allowed-tools: Bash(git add:*), Bash(git status:*), Bash(git commit:*) description: Create a git commit

Context

- Current git status: ! `git status`
- Current git diff (staged and unstaged changes): ! `git diff HEAD`
- Current branch: ! `git branch --show-current`
- Recent commits: ! `git log --oneline -10`

Your task

Based on the above changes, create a single git commit.

```
##### Include File Contents
Use the `@` prefix to reference files:
```markdown
# Reference a specific file
Review the implementation in @src/utils/helpers.js

# Reference multiple files
Compare @src/old-version.js with @src/new-version.js
```

Command Metadata

Add metadata to commands using frontmatter: ```markdown

allowed-tools: Bash(npm test), Write(src/*) description: Run tests and fix any failures

Your command content here...

```
---

## CLI Reference

### Basic Usage

```bash
# Run a single prompt and exit (print mode)
claude -p "Write a function to calculate Fibonacci numbers"

# Using a pipe to provide stdin
echo "Explain this code" | claude -p
```

```
# Output in JSON format with metadata
claude -p "Generate a hello world function" --output-format json

# Stream JSON output as it arrives
claude -p "Build a React component" --output-format stream-json
```

CLI Flags and Options

Based on the official documentation:

Flag	Description
`-p, --print`	Run in non-interactive print mode
`--output-format`	Set output format: `text` (default), `json`, `stream-json`
`--continue`	Continue the most recent conversation
`--resume`	Resume a specific conversation by ID or show picker
`--system-prompt`	Override system prompt (only works with --print)
`--append-system-prompt`	Append to default system prompt
`--mcp-config`	Load MCP servers from configuration file
`--allowedTools`	Specify allowed tools for the session
`--permission-prompt-tool`	MCP tool for handling permission prompts
`--debug`	Enable debug output

Conversation Management

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

# Continue with a specific prompt
claude --continue "Now refactor this for better performance"

# Resume a specific conversation by session ID
claude --resume 550e8400-e29b-41d4-a716-446655440000

# Continue in print mode (non-interactive)
claude -p --continue "Add error handling"
```

Output Formats

Format	Description
`text`	Plain text response (default)
`json`	JSON object with metadata

`stream-json`	Real-time JSON objects

Custom System Prompts

```
# Override system prompt
claude -p "Build a REST API" --system-prompt "You are a senior backend engineer. Focus on security, performance, and maintainability."

# Append to default prompt
claude -p "Create a database schema" --append-system-prompt "Always use PostgreSQL best practices"
```

Configuration & Settings

Settings Hierarchy

Claude Code uses hierarchical settings:

1. **Enterprise Managed Policy** (highest priority)
   - macOS: `/Library/Application Support/ClaudeCode/managed-settings.json`
   - Linux/WSL: `/etc/claude-code/managed-settings.json`
2. **User Settings**
   - `~/.claude/settings.json` - Applies to all projects
3. **Project Settings**
   - `.claude/settings.json` - Shared with team (in source control)
   - `.claude/settings.local.json` - Personal preferences (git ignored)

Configuration Example

```
{
  "permissions": {
    "allow": [
      "Bash(npm run lint)",
      "Bash(npm run test:*)",
      "Read(~/.zshrc)"
    ],
    "deny": [
      "Bash(curl:*)"
    ]
  },
  "env": {
    "CLAUDE_CODE_ENABLE_TELEMETRY": "1",
    "OTEL_METRICS_EXPORTER": "otlp"
  }
}
```

Environment Variables

All environment variables can be configured in settings.json:

- Automatically set for each session
- Can be rolled out team-wide
- Override system environment variables

Permissions & Security

Security Safeguards

Claude Code includes multiple layers of security protection:

Prompt Injection Protection

- **Permission system**: Sensitive operations require explicit approval
- **Context-aware analysis**: Detects potentially harmful instructions by analyzing the full request
- **Input sanitization**: Prevents command injection by processing user inputs
- **Command blocklist**: Blocks risky commands that fetch arbitrary content from the web like `curl` and `wget`
- **Network request approval**: Tools that make network requests require user approval by default
- **Isolated context windows**: Web fetch uses a separate context window to avoid injecting potentially malicious prompts
- **Trust verification**: First-time codebase runs and new MCP servers require trust verification
- **Command injection detection**: Suspicious bash commands require manual approval even if previously allowlisted

Access Restrictions

- **Folder access restriction**: Claude Code can only access the folder where it was started and its subfolders—it cannot go upstream to parent directories
- **Prompt fatigue mitigation**: Support for allowlisting frequently used safe commands per-user, per-codebase, or per-organization

Permission System

Claude Code uses a tiered permission system to balance power and safety:

```
# View & manage permissions
/permissions

# Configure in settings.json
{
  "permissions": {
    "allow": [
      "Bash(npm run lint)",
      "Bash(npm run test:*)",
      "Read(~/.zshrc)"
    ],
    "deny": [
      "Bash(curl:*)",
      "Write(/etc/*)"
    ]
  }
}
```

Identity and Access Management (IAM)

Authentication Methods

Claude Code supports authentication via:

- Claude.ai credentials
- Anthropic API credentials
- Bedrock Auth
- Vertex Auth

On macOS, API keys, OAuth tokens, and other credentials are stored on encrypted macOS Keychain.

API Key Helper

The setting `apiKeyHelper` can be set to a shell script which returns an API key:

- By default, this helper is called after 5 minutes or on HTTP 401 response
- Specify `CLAUDE_CODE_API_KEY_HELPER_TTL_MS` for custom refresh interval

Permission Hierarchy

When multiple settings sources exist, they are applied in the following order (highest to lowest precedence):

1. Enterprise Managed Policy Settings
2. Project Settings (`.claude/settings.json` and `.claude/settings.local.json`)
3. User Settings (`~/.claude/settings.json`)

This hierarchy ensures organizational policies are always enforced while allowing flexibility at project and user levels.

Enterprise Policy Management

For enterprise deployments, managed policy settings take precedence over all other settings:

- **macOS**: `/Library/Application Support/ClaudeCode/managed-settings.json`
- **Linux/WSL**: `/etc/claude-code/managed-settings.json`

These policy files follow the same format as regular settings files but cannot be overridden.

MCP Server Security

- Claude Code allows configuration of Model Context Protocol (MCP) servers
- List of allowed MCP servers is configured in source code as part of Claude Code settings
- Encourage writing your own MCP servers or using servers from trusted providers
- Anthropic does not manage or audit any MCP servers

Security Architecture

- **Direct API Connection**: Queries go straight to Anthropic's API without intermediate servers
- **Local Execution**: All operations happen in your environment
- **Transparent Operations**: Require approval for git commands before executing
- **Configurable Permissions**: Users and organizations can configure permissions directly

---

Hooks System

Hooks are user-defined shell commands that execute at various points in Claude Code's lifecycle.

Hook Events

Event	Description	Matcher Support
`PreToolUse`	Before tool calls	Yes
`PostToolUse`	After successful tool completion	Yes
`OnNotify`	When notifications are sent	No
`OnAgentFinish`	When main agent completes	No
`OnSubAgentFinish`	When subagent completes	No

Hook Configuration

Quick Start Example

```
# Run /hooks and select PreToolUse
# Add matcher for Bash
# Enter command:
jq -r '"\(.tool_input.command) - \(.tool_input.description // "No description")"' >>
~/.claude/bash-command-log.txt
```

Configuration Structure

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "your-command-here",
            "timeout": 60
          }
        ]
      }
    ]
  }
}
```

Hook Security Best Practices

- **Validate inputs**: Never trust input data blindly
- **Quote variables**: Use `"$VAR"` not `$VAR`
- **Block path traversal**: Check for `..` in paths
- **Use absolute paths**: Specify full paths for scripts
- **Skip sensitive files**: Avoid `.env`, `.git/`, keys

Hook Data Schema

Hooks receive JSON via stdin:

```
{
  "session_id": "string",
  "transcript_path": "string",
  "tool_name": "string",
  "tool_input": {},
  "tool_response": {}
}
```

Model Context Protocol (MCP)

MCP enables Claude Code to connect to external tools and data sources.

Adding MCP Servers

Local Server

```
# Basic syntax
claude mcp add <name> <command> [args...]

# Example
claude mcp add my-server -e API_KEY=123 -- /path/to/server arg1 arg2
```

SSE/HTTP Server

```
# Basic SSE server
claude mcp add --transport sse sse-server https://example.com/sse-endpoint

# With authentication headers
claude mcp add --transport sse api-server https://api.example.com/mcp --header "X-API-Key: your-key"
```

MCP Scopes

1. **Local Scope** (default)
   - Project-specific user settings
   - Only available to you in current project
2. **Project Scope**
   - Stored in `.mcp.json` at project root
   - Shared with team via source control
   - Requires approval on first use
3. **User Scope**
   - Available across all projects
   - Personal utilities and tools

Example: PostgreSQL Integration

```
claude mcp add postgres-server /path/to/postgres-mcp-server \
  --connection-string "postgresql://user:pass@localhost:5432/mydb"

# In Claude:
> describe the schema of our users table
> what are the most recent orders in the system?
```

MCP Resources

Reference MCP resources using `@` mentions:

```
@server:resource
```

OAuth Authentication

Many remote MCP servers support OAuth 2.0:

```
claude mcp add --transport sse github-server https://api.github.com/mcp
# Follow OAuth flow when prompted
```

SDK Usage

The Claude Code SDK enables programmatic integration.

Command Line SDK

```
# Basic usage
claude -p "Write a function"

# With error handling
if ! claude -p "$prompt" 2>error.log; then
    echo "Error occurred:" >&2
    cat error.log >&2
    exit 1
fi

# Parse JSON output
result=$(claude -p "Generate code" --output-format json)
code=$(echo "$result" | jq -r '.result')
cost=$(echo "$result" | jq -r '.cost_usd')
```

TypeScript SDK

```
import { ClaudeCode } from '@anthropic-ai/claude-code-sdk';

const claude = new ClaudeCode({
  apiKey: process.env.ANTHROPIC_API_KEY
});
```

```
const result = await claude.run({
  prompt: "Write a function to calculate Fibonacci numbers",
  outputFormat: 'json'
});
```

Python SDK

```
from claude_code_sdk import ClaudeCode

claude = ClaudeCode(api_key=os.environ['ANTHROPIC_API_KEY'])

result = claude.run(
    prompt="Write a function to calculate Fibonacci numbers",
    output_format='json'
)
```

Response Format

```
{
  "type": "result",
  "subtype": "success",
  "total_cost_usd": 0.003,
  "is_error": false,
  "duration_ms": 1234,
  "duration_api_ms": 800,
  "num_turns": 6,
  "result": "The response text here...",
  "session_id": "abc123"
}
```

IDE Integration

VS Code / Cursor / Windsurf

1. **Install Extension**:
   - Search for "Claude Code" in marketplace
   - Install and restart IDE
2. **Connect Claude**:

   ```
   # In IDE's integrated terminal
   claude
   ```

3. **Features**:
   - Selection context automatically shared
   - File reference shortcuts: Cmd+Option+K (Mac) or Alt+Ctrl+K (Linux/Windows)
   - Diagnostic errors shared automatically

JetBrains IDEs

1. **Install Plugin**:
   - Install from JetBrains marketplace
   - Restart IDE
2. **Connect**:
   - Run `claude` in built-in terminal
   - Or use `/ide` command from external terminal

External Terminal Connection

Connect from any terminal:

```
# Launch Claude Code
claude

# Connect to IDE
/ide
```

Common Workflows

Understanding a New Codebase

```
> What does this project do?
> Explain the main components and how they interact
> What are the key dependencies?
```

Finding Code

```
> Where is the authentication logic implemented?
> Show me all the API endpoints
> Find the database connection code
```

Fixing Bugs

```
> I'm getting error "undefined is not a function" - help me fix it
> The login endpoint returns 500 - can you debug this?
```

Modernizing Code

```
> Update this code to use modern JavaScript syntax
> Refactor this class to use hooks instead
> Convert callbacks to async/await
```

Adding Tests

```
> Write unit tests for the UserService class
> Add integration tests for the API endpoints
> Create test cases for edge scenarios
```

Creating PRs

```
> Create a commit with these changes
> Generate a PR description
> What should I include in the changelog?
```

Using @ References

```
# Include file content
> Review the code in @src/components/Header.js

# Include directory listing
> What's in @src/utils/?

# MCP resources
> @postgres:users_table schema
```

Extended Thinking

For complex tasks requiring deep reasoning:

Activation

Use phrases that trigger extended thinking:

- "think deeply about..."
- "think harder about edge cases"
- "think more about the implications"
- "think a lot about the best approach"

Example

```
> I need to implement a new authentication system using OAuth2 for our API.
> Think deeply about the best approach for implementing this in our codebase.
```

Claude will:

1. Gather relevant information from your codebase
2. Display thinking process in italic gray text
3. Provide a thorough, well-reasoned response

Best Use Cases

- Complex architectural decisions
- Challenging bug investigations
- Multi-step implementation planning
- Security vulnerability analysis
- Performance optimization strategies

Development Containers

Claude Code provides a preconfigured devcontainer setup that works seamlessly with VS Code's Remote - Containers extension and similar tools.

Features

- **Production-ready Node.js**: Built on Node.js 20 with essential development dependencies
- **Security by design**: Custom firewall restricting network access to only necessary services
- **Developer-friendly tools**: Includes git, ZSH with productivity enhancements, fzf, and more
- **Startup verification**: Validates firewall rules when the container initializes
- **Isolation**: Creates a secure development environment separated from your main system

Security Approach

The container implements a multi-layered security approach:

- **Precise access control**: Restricts outbound connections to whitelisted domains only (npm registry, GitHub, Anthropic API, etc.)
- **Default-deny policy**: Blocks all other external network access
- **Enhanced security measures**: Allow running `claude --dangerously-skip-permissions` for unattended operation

Setup

1. Install VS Code and the Remote - Containers extension
2. Clone the Claude Code reference implementation repository
3. When prompted, click "Reopen in Container" (or use Command Palette: Cmd+Shift+P → "Remote-Containers: Reopen in Container")

Configuration Components

- **devcontainer.json**: Controls container settings, extensions, and volume mounts
- **Dockerfile**: Defines the container image and installed tools
- **init-firewall.sh**: Establishes network security rules

Use Cases

- Isolate different client projects
- Ensure code and credentials never mix between environments
- Provide new team members with fully configured environments in minutes

Monitoring

Claude Code supports OpenTelemetry (OTel) metrics and events for monitoring and observability.

Configuration

```
# 1. Enable telemetry export
export CLAUDE_CODE_ENABLE_TELEMETRY=1

# 2. Choose exporters (both optional)
export OTEL_METRICS_EXPORTER=otlp  # Options: otlp, prometheus, console
export OTEL_LOGS_EXPORTER=otlp     # Options: otlp, console
```

```
# 3. Configure OTLP endpoint (for OTLP exporter)
export OTEL_EXPORTER_OTLP_PROTOCOL=grpc
export OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317

# 4. Optional: Set service name
export OTEL_SERVICE_NAME=claude-code
```

Available Metrics

Code Metrics

- **claude_code.code_lines_diff**: Incremented when code is added or removed
- **claude_code.pull_requests_created**: Incremented when creating PRs
- **claude_code.git_commits_created**: Incremented when creating commits

API Metrics

- **claude_code.input_tokens**: Incremented after each API request
- **claude_code.output_tokens**: Incremented after each API request

User Interaction Metrics

- **claude_code.edits.decisions**: Incremented when user accepts/rejects edits
  - Attributes: `decision` (accept/reject), `language` (file type)

Events (via OpenTelemetry logs)

- **prompt_submitted**: Logged when user submits a prompt
- **tool_result**: Logged when a tool completes execution
- **api_request**: Logged for each API request to Claude
- **api_request_error**: Logged when API request fails
- **permission_decision**: Logged when tool permission decision is made

Analysis Capabilities

All metrics can be segmented by:

- `user.account_uuid`
- `organization.id`
- `session.id`
- `model`
- `app.version`

Note: OpenTelemetry support is currently in beta and details are subject to change.

Cost Management

Usage Overview

- **Average cost**: $6 per developer per day
- **90th percentile**: Daily costs remain below $12
- **Monthly average**: ~$50-60/developer per month with Sonnet 4

Cost Tracking

In-Session Tracking

```
# View current session usage
/cost
```

Anthropic API

- Configure workspace spend limits
- View cost and usage reporting in the Anthropic Console

Bedrock and Vertex

Claude Code does not send metrics from your cloud. For cost tracking:

- Several enterprises use LiteLLM (open-source, unaffiliated with Anthropic)
- Provides spend tracking by key
- **Note**: Anthropic has not audited its security

Cost Factors

- Number of instances running
- Usage in automation
- Model selection
- Token consumption per interaction

Memory Management (CLAUDE.md)

Overview

CLAUDE.md is a special file that stores important project information, conventions, and frequently used commands. Claude Code automatically reads this file to understand your project context.

Creating CLAUDE.md

Create a `CLAUDE.md` file in your project root:

```
# Project Context

## Architecture Overview
- Frontend: React with TypeScript
- Backend: Node.js with Express
- Database: PostgreSQL with Prisma ORM
- Testing: Jest for unit tests, Cypress for E2E

## Development Setup
1. Install dependencies: `npm install`
2. Set up database: `npm run db:setup`
3. Run migrations: `npm run db:migrate`
4. Start dev server: `npm run dev`

## Coding Conventions
- Use ESLint configuration for code style
- Follow conventional commits for messages
- All new features need tests (min 80% coverage)
- Use TypeScript strict mode

## Common Commands
```

```
- `npm test` - Run all tests
- `npm run lint` - Check code style
- `npm run build` - Build for production
- `npm run db:seed` - Seed development database

## Project Structure
- `/src/api` - API endpoints
- `/src/components` - React components
- `/src/services` - Business logic
- `/src/utils` - Helper functions

## Important Notes
- Never commit .env files
- Always run tests before pushing
- Use feature branches for new work
```

Benefits

- **Persistent Knowledge**: Claude remembers project-specific information
- **Team Alignment**: Share conventions and standards
- **Reduced Repetition**: No need to explain context repeatedly
- **Quick Reference**: Common commands at Claude's fingertips

Best Practices

1. **Check into source control**: Share with your team
2. **Keep it updated**: Maintain as project evolves
3. **Be concise**: Focus on essential information
4. **Include examples**: Show preferred patterns
5. **Document gotchas**: Note common pitfalls

Enterprise Adoption

At Anthropic, CLAUDE.md is standard in every codebase to ensure Claude Code understands:

- System architecture
- How to run tests
- Best practices for contributing
- Project-specific conventions

GitHub Actions Integration

Quick Setup

1. **Automatic Setup** (Anthropic API only):

   ```
   claude /install-github-app
   ```

2. **Manual Setup**:
   - Install Claude Code GitHub App
   - Add `ANTHROPIC_API_KEY` secret
   - Create workflow files

Example Workflows

Turn Issues into PRs

```yaml
name: Claude Code Issue to PR
on:
  issue_comment:
    types: [created]

jobs:
  claude-code:
    if: contains(github.event.comment.body, '@claude')
    runs-on: ubuntu-latest
    steps:
      - uses: anthropics/claude-code-action@v1
        with:
          github-token: ${{ secrets.GITHUB_TOKEN }}
          anthropic-api-key: ${{ secrets.ANTHROPIC_API_KEY }}
```

Automated Code Review

```yaml
name: Claude Code Review
on:
  pull_request:
    types: [opened, synchronize]

jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: anthropics/claude-code-action@v1
        with:
          command: review
          github-token: ${{ secrets.GITHUB_TOKEN }}
          anthropic-api-key: ${{ secrets.ANTHROPIC_API_KEY }}
```

Cost Considerations

- Claude API costs apply
- GitHub Actions compute time
- Consider using on specific events only

Enterprise Integration

Deployment Overview

Claude Code supports flexible configuration options for enterprise deployments:

- **Direct Provider**: AWS Bedrock or Google Vertex AI for model access
- **LLM Gateway**: A service that handles authentication and provides provider-compatible endpoints
- Both configurations can be used in tandem

Amazon Bedrock

Prerequisites

- AWS account with Bedrock access enabled
- Appropriate IAM permissions
- AWS CLI installed and configured (optional)

Configuration

```
# Enable Bedrock integration
export CLAUDE_CODE_USE_BEDROCK=1
export AWS_REGION=us-east-1

# Authentication methods:
# 1. Environment variables
export AWS_ACCESS_KEY_ID=your-access-key-id
export AWS_SECRET_ACCESS_KEY=your-secret-access-key
export AWS_SESSION_TOKEN=your-session-token

# 2. AWS Profile
export AWS_PROFILE=your-profile

# 3. IAM Role (when running on EC2/ECS/Lambda)
# Credentials are automatically provided
```

Note: Claude Code does not currently support dynamic credential management (such as automatically calling `aws sts assume-role`).

Google Vertex AI

Prerequisites

- Google Cloud Platform (GCP) account with billing enabled
- Vertex AI API enabled
- Appropriate IAM roles

Configuration

```
# Enable Vertex AI
export CLAUDE_CODE_USE_VERTEX=1
export CLOUD_ML_REGION=us-east5
export ANTHROPIC_VERTEX_PROJECT_ID=your-project-id

# Authenticate
gcloud auth application-default login

# Enable APIs
gcloud services enable aiplatform.googleapis.com

# Optional: Disable prompt caching if needed
export DISABLE_PROMPT_CACHING=1
```

Model Configuration

```
# Override default models
export ANTHROPIC_MODEL='claude-opus-4@20250514'
export ANTHROPIC_SMALL_FAST_MODEL='claude-3-5-haiku@20241022'
```

Note: Vertex AI may not support Claude Code default models on non-us-east5 regions.

Corporate Proxy Configuration

Configure Claude Code to work with your organization's proxy servers:

```
# Basic proxy configuration
export HTTPS_PROXY='https://proxy.example.com:8080'

# With authentication
export HTTPS_PROXY='https://username:password@proxy.example.com:8080'
```

LLM Gateway Integration

Deploy centralized model access with usage tracking, budgeting, and audit logging:

Basic Configuration

```
# Set base URL for your LLM gateway
export ANTHROPIC_BASE_URL='https://your-llm-gateway.com'

# For provider-specific endpoints
export ANTHROPIC_BEDROCK_BASE_URL='https://your-llm-gateway.com/bedrock'
export ANTHROPIC_VERTEX_BASE_URL='https://your-llm-gateway.com/vertex'
```

Combined Configurations

Example: Google Vertex AI with LLM Gateway

```
# Enable Vertex
export CLAUDE_CODE_USE_VERTEX=1

# Configure LLM gateway
export ANTHROPIC_VERTEX_BASE_URL='https://your-llm-gateway.com/vertex'
export CLAUDE_CODE_SKIP_VERTEX_AUTH=1  # If gateway handles GCP auth
```

The `SKIP_AUTH` flags (`CLAUDE_CODE_SKIP_BEDROCK_AUTH` , `CLAUDE_CODE_SKIP_VERTEX_AUTH`) are used when the gateway handles provider authentication.

Enterprise Best Practices

1. **Start Small**: Encourage new users to try Claude Code for codebase Q&A or smaller bug fixes
2. **Plan First**: Ask Claude Code to make a plan and check suggestions
3. **Managed Permissions**: Security teams can configure managed permissions that cannot be overwritten
4. **MCP Configuration**: Central team should configure MCP servers and check `.mcp.json` into codebase

5. **Cost Tracking**: Create dedicated accounts/projects for Claude Code to simplify cost tracking

Status Verification

Use the `/status` slash command to verify your configuration.

Troubleshooting

Installation Issues

Permission Errors

```
# Don't use sudo!
# Instead, configure npm prefix:
mkdir ~/.npm-global
npm config set prefix '~/.npm-global'
echo 'export PATH=~/.npm-global/bin:$PATH' >> ~/.bashrc
source ~/.bashrc
```

Auto-Update Issues

- Check npm permissions (see above)
- Or disable: `export DISABLE_AUTOUPDATER=1`

Authentication Problems

```
# Clear stored auth
rm -rf ~/.claude/auth

# Re-authenticate
claude
```

Performance Issues

For large codebases:

1. Limit scope of operations
2. Use specific file paths
3. Consider `.claudeignore` file
4. Close unnecessary applications

Terminal-Specific Issues

JetBrains ESC Key

In PyCharm/IntelliJ:

1. Settings → Keymap
2. Search "Switch focus to Editor"
3. Remove ESC binding

VS Code Integration

- Use integrated terminal
- Ensure `code` CLI is installed

- Check extension permissions

Repetitive Approvals

```
# Allow specific commands
/permissions

# Or configure in settings.json
{
  "permissions": {
    "allow": ["Bash(npm test)", "Write(src/*)"]
  }
}
```

Best Practices

Project Setup

1. **Create CLAUDE.md** for project context
2. **Configure .claudeignore** to skip large/irrelevant files
3. **Set up team permissions** in `.claude/settings.json`
4. **Create common slash commands** in `.claude/commands/`

Effective Prompting

1. **Be specific** about requirements
2. **Reference files** with @ syntax
3. **Use extended thinking** for complex tasks
4. **Provide examples** when possible

Security

1. **Review permissions** regularly
2. **Use project-scoped MCP servers** carefully
3. **Validate hook commands** before adding
4. **Keep sensitive data** out of prompts

Performance

1. **Work incrementally** on large changes
2. **Use file references** instead of pasting
3. **Limit search scope** when possible
4. **Close unused MCP connections**

Team Collaboration

1. **Share settings** via source control
2. **Document conventions** in CLAUDE.md
3. **Create team slash commands**
4. **Use GitHub Actions** for automation

License & Data Usage

License

Claude Code is provided as a Beta research preview under Anthropic's Commercial Terms of Service.

Feedback Transcripts

If you send feedback (via `/bug` or transcripts):

- Used to debug issues and improve functionality
- Helps reduce similar bugs in the future
- **NOT used for model training**

For full details:

- Commercial Terms of Service
- Privacy Policy