Claude Code - Best Practices para Usuarios Avanzados

Guía exhaustiva de patterns, strategies, y best practices para usar Claude Code efectivamente en producción.

📑 Tabla de Contenidos

1. Permission Strategies
2. Context Management
3. MCP Server Design
4. Agent Architecture
5. Skill Organization
6. Git Workflow Integration
7. CI/CD Automation
8. Team Collaboration
9. Security & Privacy
10. Performance Optimization
11. Error Handling & Recovery
12. Testing Strategies

Permission Strategies

Choosing the Right Permission Mode

Different scenarios require different permission strategies:

Development Mode (Rapid Iteration)

Scenario: Prototyping, exploring ideas, adding features quickly

Configuration:

```
{
  "permissions": {
    "defaultMode": "acceptEdits"
  }
}
```

When to use:

- Solo development
- Hackathons / prototyping
- Feature branches (not main)
- Experimental work

Risks:

- Potential for unwanted changes
- Less visibility into what's modified
- May need to review changes afterward

Mitigation:

- Use with git branches (easy rollback)
- Review changes with `git diff` before committing
- Set up post-edit hooks for validation

Review Mode (Safe Exploration)

Scenario: Reviewing code, planning refactors, understanding codebase

Configuration:

```
{
  "permissions": {
    "defaultMode": "plan"
  }
}
```

When to use:

- Onboarding to new codebases
- Code review sessions
- Pre-implementation planning
- Production codebase exploration
- Security audits

Benefits:

- Zero risk of accidental changes
- Focus on analysis and understanding
- Can explore freely

Default Mode (Balanced)

Scenario: Standard development with safety checks

Configuration:

```
{
  "permissions": {
    "defaultMode": "default"
  }
}
```

When to use:

- Regular development work
- Pair programming with Claude
- When you want approval control

Benefits:

- Visibility into every change
- Can reject unwanted modifications
- Balance between speed and safety

Path-Specific Rules

Fine-grained control based on file locations:

```
{
  "permissions": {
    "defaultMode": "default",
    "rules": [
      {
        "working_directory": "tests/**",
        "defaultMode": "acceptEdits",
        "reason": "Tests are safe to modify freely"
      },
      {
        "working_directory": "src/core/**",
        "tools": ["Read", "Grep", "Glob"],
        "deny": ["Write", "Edit"],
        "reason": "Core logic requires explicit approval"
      },
      {
        "working_directory": "**/*.config.js",
        "defaultMode": "default",
        "reason": "Config files need review"
      },
      {
        "working_directory": "production/**",
        "deny": ["Bash", "Write", "Edit"],
        "reason": "Production files are read-only"
      }
    ]
  }
}
```

Tool-Specific Restrictions

Limit specific tools in sensitive contexts:

```
{
  "permissions": {
    "rules": [
      {
        "working_directory": "**",
        "allowedTools": {
          "Bash": [
            "git status:*",
            "git diff:*",
            "git log:*",
            "npm test:*",
            "npm run lint:*"
          ]
        },
        "reason": "Only allow safe read-only and test commands"
      }
    ]
  }
}
```

Dynamic Permission Switching

Switch modes mid-session based on task:

```
# Start in plan mode for exploration
> [Shift+Tab until Plan Mode]
> Analyze the authentication system

# Switch to default for implementation
> [Shift+Tab to Default Mode]
> Now implement the OAuth2 flow we planned

# Switch to acceptEdits for quick fixes
> [Shift+Tab to Accept Edits]
> Fix all the linting issues quickly
```

Context Management

Understanding Context Limits

Claude Code has a large but finite context window. Effective context management is crucial for:

- Performance (faster responses)
- Cost (fewer tokens = lower cost)
- Quality (focused context = better results)

Monitoring Context Usage

```
# In Claude session
> /context
```

This shows a visual grid of context usage:

```
[████████████▒▒▒▒▒▒▒▒] 60% used
```

Strategies for Context Preservation

1. Use Subagents for High-Volume Operations

Problem: Reading 100 files fills context quickly

Solution:

```
> Use a subagent to search all test files for uses of the deprecated API,
and return only a summary of which files need updating
```

Why it works:

- Subagent's context is separate
- Only summary returned to main context
- Main context stays lean

2. Strategic File Reading

Bad:

```
> Read all files in src/ and summarize
```

Good:

```
> Use Glob to find TypeScript files in src/
> From those results, Read only the controller files
> Summarize just the API endpoints
```

3. Incremental Exploration

Bad:

```
> Analyze the entire codebase structure
```

Good:

```
> What's the overall directory structure?
[After seeing structure]
> Now tell me about the src/api/ directory
[After understanding API]
> Dig into src/api/auth/ specifically
```

4. Reference Files Instead of Reading

Use @ mentions when you don't need Claude to analyze:

```
> The authentication logic is in @src/auth.controller.js - can you add
rate limiting to it?
```

This includes the file content without explicitly reading it first.

5. Summarization Checkpoints

For long sessions:

```
> Before we continue, summarize what we've done so far and what's left to
do
```

This creates a compressed reference point.

Context-Efficient Prompts

Inefficient:

```
> Show me every function in the codebase that deals with users
```

Efficient:

```
> Find files that match **/users.* and tell me which ones have functions
for user creation or deletion (just list the function names and files)
```

When to Start a New Session

Signs you should start fresh:

- Context indicator shows >80% usage
- Responses become slow or less accurate
- Task switches to a different area of codebase
- Previous work is completed and committed

Tip: Name sessions for easy resumption:

```
> /rename authentication-refactor
```

Later:

```
claude --resume authentication-refactor
```

MCP Server Design

Choosing HTTP vs Stdio

HTTP Servers (Recommended for Cloud Services)

Use when:

- Connecting to cloud APIs (GitHub, Notion, Slack)
- Service has existing REST API
- Multiple clients might connect
- Server can handle concurrent requests

Pros:

- Easy to deploy and scale
- Standard HTTP auth patterns
- Can be shared across team
- Cloud-hosted (no local resources)

Cons:

- Network latency
- Requires internet connection
- Authentication complexity

Example:

```
claude mcp add --transport http github https://api.githubcopilot.com/mcp/
\
  --header "Authorization: Bearer $GITHUB_TOKEN"
```

Stdio Servers (For Local Tools)

Use when:

- Direct system access needed (databases, files)
- Low latency critical
- Tool runs locally (CLI tools, local DBs)
- Single user/process

Pros:

- No network latency
- Direct access to system resources
- Works offline
- Simpler auth (environment variables)

Cons:

- Must be installed locally
- One connection at a time
- Team members need individual setup

Example:

```
claude mcp add --transport stdio db -- npx -y @bytebase/dbhub \
  --dsn "postgresql://localhost:5432/mydb" \
  --env DATABASE_PASSWORD=$DB_PASS
```

MCP Server Scope Strategy

Project Scope (Team Shared)

Use for:

- Tools everyone on the team needs
- Project-specific integrations
- Development environment setup

Configuration (`.mcp.json` in project root):

```
{
  "mcpServers": {
    "project-db": {
      "type": "stdio",
      "command": "npx",
      "args": ["-y", "@bytebase/dbhub", "--dsn", "${DATABASE_URL}"],
      "env": {
        "DATABASE_URL": "${DATABASE_URL}"
      }
```

```
    },
    "github": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp/"
    }
  }
}
```

Benefits:

- Team consistency
- Version controlled
- Easy onboarding (clone repo = get MCP servers)
- Shared configuration

Setup:

```
# Add to project
claude mcp add --scope project github https://api.githubcopilot.com/mcp/

# Commit to repo
git add .mcp.json
git commit -m "feat: add GitHub MCP server for team"
```

User Scope (Personal Tools)

Use for:

- Personal productivity tools
- Cross-project utilities
- Your own API keys/credentials

Configuration (`~/.claude.json`):

```
{
  "mcpServers": {
    "personal-notes": {
      "type": "http",
      "url": "https://mcp.notion.com/mcp",
      "headers": {
        "Authorization": "Bearer YOUR_PERSONAL_TOKEN"
      }
    }
  }
}
```

Setup:

```
claude mcp add --scope user notion https://mcp.notion.com/mcp \
  --header "Authorization: Bearer $NOTION_TOKEN"
```

Local Scope (Experimental)

Use for:

- Testing new MCP servers
- One-off experiments
- Project-specific overrides

Storage: `~/.claude.json` under specific project path

Setup:

```
# Default scope (local)
claude mcp add test-server https://localhost:8080/mcp
```

Security Best Practices for MCP

1. Never Commit Secrets

Bad (`.mcp.json`):

```
{
  "mcpServers": {
    "github": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp/",
      "headers": {
        "Authorization": "Bearer ghp_hardcoded_token_here"
      }
    }
  }
}
```

Good:

```
{
  "mcpServers": {
    "github": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp/",
      "headers": {
        "Authorization": "Bearer ${GITHUB_TOKEN}"
```

```
      }
    }
  }
}
```

Then in `.env` (not committed):

```
GITHUB_TOKEN=ghp_your_token_here
```

2. Use Read-Only Tokens When Possible

For MCP servers that only need read access:

- GitHub: Use token with only `read:org`, `read:repo` scopes
- Database: Create read-only user
- APIs: Request minimum required permissions

3. Audit MCP Server Permissions

Regularly review what each MCP server can access:

```
# List all servers
claude mcp list

# Check specific server
claude mcp get github

# In Claude session
> /mcp
# Review what resources each server exposes
```

4. Separate Production and Development

Never point MCP servers at production resources directly:

Bad:

```
claude mcp add db -- npx -y @bytebase/dbhub \
  --dsn "postgresql://admin:pass@production.db:5432/prod"
```

Good:

```
# Use read-replica or development database
claude mcp add db -- npx -y @bytebase/dbhub \
```

```
  --dsn "postgresql://readonly:pass@dev-replica.db:5432/dev"
```

MCP Server Patterns

Pattern 1: Aggregation Server

Single MCP server that aggregates multiple data sources:

```
// custom-mcp-server.js
import { Server } from '@modelcontextprotocol/sdk/server/index.js';

const server = new Server({
  name: 'team-aggregator',
  version: '1.0.0'
});

// Aggregate GitHub + Jira + Database
server.setRequestHandler('resources/list', async () => {
  const [github, jira, db] = await Promise.all([
    fetchGitHubData(),
    fetchJiraData(),
    queryDatabase()
  ]);

  return { resources: [...github, ...jira, ...db] };
});
```

Pattern 2: Caching Proxy

MCP server that caches expensive operations:

```
const cache = new Map();

server.setRequestHandler('tools/call', async (request) => {
  const cacheKey = JSON.stringify(request);

  if (cache.has(cacheKey)) {
    return cache.get(cacheKey);
  }

  const result = await expensiveOperation(request);
  cache.set(cacheKey, result);

  return result;
});
```

Pattern 3: Rate-Limited Wrapper

Protect underlying APIs from abuse:

```
import { RateLimiter } from 'limiter';

const limiter = new RateLimiter({
  tokensPerInterval: 10,
  interval: 'minute'
});

server.setRequestHandler('tools/call', async (request) => {
  await limiter.removeTokens(1);
  return await callUnderlyingAPI(request);
});
```

Agent Architecture

When to Create a Custom Agent

Create a custom agent when:

1. **Task is specialized**: Security auditing, performance analysis, specific code reviews
2. **Tool restrictions needed**: Agent should only read, not write
3. **Specific model appropriate**: Use Haiku for fast searches, Opus for complex analysis
4. **Context isolation beneficial**: High-volume operations that would fill main context
5. **Proactive invocation desired**: Want Claude to automatically use agent when task matches

Don't create an agent for:

- One-off tasks
- General development
- Tasks that need frequent user interaction

Agent Design Patterns

Pattern 1: Read-Only Analyzer

Use case: Security audits, code quality checks, documentation generation

```
---
name: analyzer
description: Analyze code for issues. Use proactively after code changes.
tools: Read, Grep, Glob
model: sonnet
permissionMode: plan
---

You are a code analyzer. Examine code and report findings without making
changes.
```

```
Process:
1. Scan relevant files
2. Identify issues by category
3. Report with file:line references
4. Suggest fixes (but don't implement)
```

Benefits:

- Can't accidentally modify code
- Fast (plan mode has less overhead)
- Safe to run automatically

Pattern 2: Auto-Fixer

Use case: Linting, formatting, simple refactoring

```
---
name: auto-fixer
description: Automatically fix code style and common issues.
tools: Read, Edit, Grep, Glob
model: haiku
permissionMode: acceptEdits
---

You are an auto-fixer. Fix issues automatically according to these rules:

1. ESLint violations: Fix automatically
2. Import ordering: Alphabetize
3. Console.logs: Remove from non-test files
4. Unused variables: Remove if clearly unused

Be conservative. If unsure, skip the fix.
```

Benefits:

- Fast fixes without user intervention
- Haiku model keeps costs low
- AcceptEdits mode for speed

Pattern 3: Interactive Specialist

Use case: Complex debugging, architecture discussions

```
---
name: architect
description: Architectural guidance and complex decision making.
tools: Read, Grep, Glob, AskUserQuestion
```

```
model: opus
permissionMode: default
---

You are a senior architect. Help users make complex technical decisions.

Process:
1. Understand the requirements deeply (ask questions)
2. Analyze existing architecture
3. Present multiple options with tradeoffs
4. Recommend an approach
5. Create detailed implementation plan

Always ask clarifying questions. Never assume requirements.
```

Benefits:

- Opus model for complex reasoning
- AskUserQuestion for interactive refinement
- Default mode for user control

Pattern 4: Background Worker

Use case: Long-running tasks (test suites, migrations)

```
---
name: test-runner
description: Run tests and report results. Can run in background.
tools: Bash, Read, Grep
model: haiku
permissionMode: dontAsk
---

You are a test runner. Execute test suites and report results.

Process:
1. Run test command
2. Parse output
3. Identify failures
4. Report summary

For failures:
- File and test name
- Error message
- Stack trace (first few lines)

Keep output concise.
```

Benefits:

- Can run in background (Ctrl+B)
- dontAsk mode allows automation
- Haiku for cost efficiency

Agent Composition

Agents can invoke other agents:

```
---
name: full-review
description: Comprehensive code review using multiple specialized agents.
tools: Task, Read
model: sonnet
---

You orchestrate a full code review using specialized agents:

1. Use security-auditor agent for security scan
2. Use performance-analyzer agent for performance issues
3. Use test-coverage agent to check test coverage
4. Use docs-checker agent for documentation

Compile results into a unified report organized by priority.
```

Agent Hierarchies

```
main-agent (User interaction)
├── analyzer-agent (Read-only analysis)
│   ├── security-agent (OWASP checks)
│   └── performance-agent (Profiling)
├── fixer-agent (Automated fixes)
│   ├── lint-agent (Style fixes)
│   └── refactor-agent (Simple refactors)
└── reporter-agent (Compile results)
```

Skill Organization

Single-File vs Multi-File Skills

Single-File Skill (Simple)

Use when:

- Skill is straightforward
- Less than 200 lines of instruction
- No external scripts needed

Structure:

```
.claude/skills/
└── quick-check/
    └── SKILL.md
```

Example (`SKILL.md`):

```
---
name: quick-check
description: Quick sanity check before commits.
allowed-tools: Read, Grep, Bash
---

Run these checks:
1. No console.log in src/
2. No debugger statements
3. All tests pass
4. No TODO comments in committed code

Report any violations.
```

Multi-File Skill (Complex)

Use when:

- Detailed documentation needed
- Multiple scripts or resources
- Team needs comprehensive reference

Structure:

```
.claude/skills/
└── code-review/
    ├── SKILL.md              # Overview (what Claude reads)
    ├── STYLE_GUIDE.md        # Detailed style rules
    ├── SECURITY.md           # Security checklist
    ├── EXAMPLES.md           # Good/bad examples
    └── scripts/
        ├── check-coverage.sh
        └── run-linter.sh
```

SKILL.md (Progressive Disclosure):

```
---
name: code-review
```

```
description: Comprehensive code review using team standards.
allowed-tools: Read, Grep, Glob, Bash
---

Perform code review following our team standards.

## Quick Reference

- Style: See STYLE_GUIDE.md for details
- Security: See SECURITY.md for checklist
- Examples: See EXAMPLES.md for good/bad patterns

## Process

1. Check style (use scripts/run-linter.sh)
2. Security scan (follow SECURITY.md)
3. Test coverage (use scripts/check-coverage.sh)
4. Review logic and clarity

For detailed guidelines, refer to the other files in this skill.
```

Skill Scope Strategy

Personal Skills (`~/.claude/skills/`)

Use for:

- Your personal coding preferences
- Cross-project utilities
- Experimental workflows

Example:

```
~/.claude/skills/
├── my-debugging-flow/
│   └── SKILL.md
└── quick-docs/
    └── SKILL.md
```

Project Skills (`.claude/skills/`)

Use for:

- Team standards and conventions
- Project-specific workflows
- Shared processes

Example:

```
.claude/skills/
├── api-design/
│   ├── SKILL.md
│   └── STANDARDS.md
├── database-migrations/
│   ├── SKILL.md
│   └── checklist.md
└── deployment/
    ├── SKILL.md
    └── scripts/
        └── deploy.sh
```

Commit to version control:

```
git add .claude/skills/
git commit -m "docs: add team code review skill"
```

Skill vs Agent vs Command

When to use each:

Feature	Skill	Agent	Slash Command
Auto-invoke	✅ Yes	✅ Yes	❌ No (explicit)
Multiple files	✅ Yes	❌ No	❌ No
Isolated context	❌ No	✅ Yes	❌ No
Tool restrictions	✅ allowed-tools	✅ tools	✅ allowed-tools
Custom model	✅ Yes	✅ Yes	❌ No (inherits)
Scripts	✅ Yes	❌ No	✅ Yes (bash)
Progressive disclosure	✅ Yes	❌ No	❌ No

Choose Skill when:

- Knowledge encapsulation (team standards, processes)
- Multiple supporting documents needed
- Want auto-invocation based on task

Choose Agent when:

- Need context isolation
- Want specific tool/permission restrictions
- Background execution possible
- Task is specialized and recurring

Choose Command when:

- Simple prompt template
- Explicit invocation preferred
- User provides arguments
- One-liner suffices

Git Workflow Integration

Commit Workflow with Claude

Manual Approach

```
# 1. Make changes with Claude
> Implement user authentication

# 2. Review changes
> Show me what files changed
> git diff src/auth.controller.js

# 3. Stage selectively
git add src/auth.controller.js src/middleware/auth.middleware.js

# 4. Create commit
> Create a commit with these staged changes. Follow conventional commits
format.

# Claude analyzes diff and creates:
git commit -m "feat(auth): implement JWT-based authentication

- Add login and register endpoints
- Create auth middleware for protected routes
- Add JWT token generation and validation

Closes #123"
```

Automated Approach

```
# One command
> Implement user authentication, test it, and create a commit when done
```

Claude will:

1. Implement the feature
2. Run tests
3. If tests pass, stage changes and commit
4. If tests fail, fix and retry

Pre-commit Hooks

Option 1: npm Script

`package.json`:

```
{
  "scripts": {
    "pre-commit": "claude -p 'Review staged changes. Block commit if
critical issues found. Output: PASS or FAIL with reasons.'"
  }
}
```

Use with:

```
npm run pre-commit && git commit -m "message"
```

Option 2: Husky

Install Husky:

```
npm install --save-dev husky
npx husky install
```

Create hook (`.husky/pre-commit`):

```
#!/bin/sh
. "$(dirname "$0")/_/husky.sh"

# Run Claude Code review
claude -p "Review staged changes for:
1. Security issues (critical)
2. Code style violations (warnings)
3. Missing tests for new functions (warnings)

Output format:
PASS - if no critical issues
FAIL - if critical issues found (list them)
" > /tmp/claude-review.txt

if grep -q "FAIL" /tmp/claude-review.txt; then
  cat /tmp/claude-review.txt
  echo "\n❌ Commit blocked by Claude Code review"
  exit 1
fi
```

```
echo "✅ Pre-commit check passed"
```

Pull Request Workflow

Creating PRs

```
# Option 1: Simple
> Create a PR for these changes

# Option 2: Detailed
> Create a PR with:
- Title: "feat: Add user authentication"
- Target branch: main
- Description should include:
  * Summary of changes
  * Testing done
  * Breaking changes (if any)
  * Screenshots (if UI changes)
```

PR Reviews

```
# Review someone else's PR
> Review PR #456 from user-auth branch

# Claude will:
# 1. Fetch PR diff
# 2. Analyze changes
# 3. Check for issues (security, style, logic)
# 4. Suggest improvements
# 5. Comment on PR if you approve
```

Branch Strategies

Feature Branch Workflow

```
# 1. Create branch
git checkout -b feat/user-auth

# 2. Implement with Claude
claude "Implement user authentication"

# 3. Review and commit
> Review my changes and create commits (split into logical commits)

# 4. Create PR
```

```
> Create a PR to main

# 5. Claude can also handle updates
> Update the PR based on review feedback in PR #123
```

Git Worktrees for Parallel Work

Scenario: Need to work on multiple features simultaneously

```
# Main project in ~/project
cd ~/project

# Create worktree for feature A
git worktree add ../project-feature-a -b feat/feature-a

# Create worktree for feature B
git worktree add ../project-feature-b -b feat/feature-b

# Work on feature A
cd ../project-feature-a
claude "Implement feature A"

# In another terminal, work on feature B
cd ~/project-feature-b
claude "Fix urgent bug B"

# Both sessions are isolated
# Changes don't interfere

# When done, merge and remove worktrees
cd ~/project
git merge feat/feature-a
git worktree remove ../project-feature-a
```

Benefits:

- No branch switching (context preserved)
- Independent Claude sessions
- Parallel development
- No stashing needed

CI/CD Automation

GitHub Actions Integration

Basic PR Review

`.github/workflows/claude-review.yml`:

```yaml
name: Claude PR Review

on:
  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Setup Claude Code
        run: |
          curl -fsSL https://claude.ai/install.sh | bash
          export PATH="$HOME/.claude/bin:$PATH"
          claude config set apiKey ${{ secrets.ANTHROPIC_API_KEY }}

      - name: Review Changes
        run: |
          export PATH="$HOME/.claude/bin:$PATH"
          claude -p "Review PR changes focusing on security and
correctness. Output markdown." > review.md

      - name: Comment on PR
        uses: actions/github-script@v7
        with:
          script: |
            const fs = require('fs');
            const review = fs.readFileSync('review.md', 'utf8');
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: `## 🤖 Claude Code Review\n\n${review}`
            });
```

Advanced: Block PR on Issues

```yaml
- name: Review and Block if Critical
  id: review
  run: |
    export PATH="$HOME/.claude/bin:$PATH"
    REVIEW=$(claude -p "Review changes. Output JSON: {\"status\":
\"pass|fail\", \"critical\": [...], \"warnings\": [...]}" --output-format
json)
    echo "review=$REVIEW" >> $GITHUB_OUTPUT
```

```
    STATUS=$(echo $REVIEW | jq -r '.status')
    if [ "$STATUS" = "fail" ]; then
      echo "❌ Critical issues found"
      exit 1
    fi

- name: Require fixes
  if: failure()
  run: |
    echo "::error::Critical issues found. PR blocked."
```

Jenkins Integration

```
pipeline {
  agent any

  environment {
    ANTHROPIC_API_KEY = credentials('anthropic-api-key')
  }

  stages {
    stage('Setup') {
      steps {
        sh 'curl -fsSL https://claude.ai/install.sh | bash'
        sh 'export PATH="$HOME/.claude/bin:$PATH"'
        sh 'claude config set apiKey $ANTHROPIC_API_KEY'
      }
    }

    stage('Code Review') {
      steps {
        script {
          def review = sh(
            script: 'claude -p "Review changes for security and quality"',
            returnStdout: true
          )
          echo "Review: ${review}"

          // Parse and fail if critical
          if (review.contains('CRITICAL')) {
            error('Critical issues found')
          }
        }
      }
    }

    stage('Auto-fix') {
      when {
        expression { env.BRANCH_NAME != 'main' }
      }
      steps {
```

```
          sh 'claude -p "Fix any linting issues automatically and commit"'
        }
      }
    }
  }
```

GitLab CI Integration

`.gitlab-ci.yml`:

```
claude-review:
  stage: test
  image: ubuntu:latest
  before_script:
    - apt-get update && apt-get install -y curl
    - curl -fsSL https://claude.ai/install.sh | bash
    - export PATH="$HOME/.claude/bin:$PATH"
    - claude config set apiKey $ANTHROPIC_API_KEY
  script:
    - |
      claude -p "Review MR changes. Focus on:
      1. Security vulnerabilities
      2. Performance issues
      3. Test coverage
      Output findings in markdown." > review.md
    - cat review.md
  artifacts:
    reports:
      codequality: review.md
  only:
    - merge_requests
```

Continuous Translation

Auto-translate locale files on changes:

`.github/workflows/translate.yml`:

```
name: Auto-translate

on:
  push:
    paths:
      - 'src/locales/en.json'
    branches:
      - main

jobs:
  translate:
```

```
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Claude
        run: |
          curl -fsSL https://claude.ai/install.sh | bash
          export PATH="$HOME/.claude/bin:$PATH"
          claude config set apiKey ${{ secrets.ANTHROPIC_API_KEY }}

      - name: Translate
        run: |
          export PATH="$HOME/.claude/bin:$PATH"
          claude -p "Find new keys in src/locales/en.json that don't exist
in es.json, fr.json, de.json. Translate them accurately and update the
files. Preserve JSON formatting."

      - name: Create PR
        uses: peter-evans/create-pull-request@v5
        with:
          commit-message: "chore(i18n): auto-translate new strings"
          title: "🌍 Auto-translated new locale strings"
          body: "Automated translation of new strings from en.json"
          branch: auto-translate
          labels: i18n, automated
```

Team Collaboration

Project Setup for Teams

1. Initialize Project

```
cd your-project
claude
> /init
```

This creates `CLAUDE.md` with project context.

2. Define Team Configuration

`.claude/settings.json`:

```
{
  "permissions": {
    "defaultMode": "default",
    "rules": [
      {
```

```
        "working_directory": "src/core/**",
        "defaultMode": "plan",
        "reason": "Core logic needs careful review"
      },
      {
        "working_directory": "tests/**",
        "defaultMode": "acceptEdits",
        "reason": "Tests are safe to modify"
      }
    ]
  },
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "npm run lint:fix $FILE_PATH",
            "continueOnError": true
          }
        ]
      }
    ]
  }
}
```

3. Add Team MCP Servers

`.mcp.json`:

```
{
  "mcpServers": {
    "github": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp/",
      "description": "GitHub integration for issues and PRs"
    },
    "dev-db": {
      "type": "stdio",
      "command": "npx",
      "args": ["-y", "@bytebase/dbhub", "--dsn", "${DEV_DATABASE_URL}"],
      "description": "Development database (read-only)"
    }
  }
}
```

4. Create Team Agents

`.claude/agents/code-reviewer.md`:

```
---
name: code-reviewer
description: Reviews code following team standards. Use for PR reviews.
tools: Read, Grep, Glob
model: sonnet
permissionMode: plan
---

[Team-specific review guidelines]
```

5. Add Team Skills

`.claude/skills/api-design/SKILL.md`:

```
---
name: api-design
description: Design REST APIs following team conventions.
allowed-tools: Read, Write, Edit
---

[Team API design standards]
```

6. Document in CLAUDE.md

```
# Project: E-Commerce Platform

## Architecture
- Microservices with REST APIs
- PostgreSQL database
- React frontend

## Team Conventions
- Feature branches (feat/*, fix/*)
- Conventional commits
- Min 80% test coverage
- Code review required for merge

## Common Tasks

### Add new API endpoint
1. Define route in src/routes/
2. Create controller in src/controllers/
3. Add service logic in src/services/
4. Write tests in tests/integration/
5. Update API docs in docs/api/

### Database migrations
```

```bash
npm run migration:create <name>
# Edit migration file
npm run migration:up
```

Agents Available

- code-reviewer: PR reviews following our standards
- security-auditor: OWASP security scans
- test-coverage: Analyze test coverage gaps

Skills Available

- api-design: REST API design standards
- database-migrations: Safe migration process
- deployment-checklist: Pre-deploy verification

```
#### 7. Commit Configuration

```bash
git add .claude/ .mcp.json CLAUDE.md
git commit -m "feat: add Claude Code team configuration"
git push
```
```

Onboarding New Team Members

New developer joins:

```
git clone team-repo
cd team-repo
npm install
claude
```

Claude automatically loads:

- Team settings (`.claude/settings.json`)
- MCP servers (`.mcp.json`)
- Agents (`.claude/agents/`)
- Skills (`.claude/skills/`)
- Project knowledge (`CLAUDE.md`)

First session:

```
> Help me understand this codebase
```

```
Claude:
I see you're working on the E-Commerce Platform. Based on CLAUDE.md:
- Microservices architecture
- REST APIs with Node.js
- PostgreSQL database
- React frontend

Key areas:
- src/api/ - API endpoints
- src/services/ - Business logic
- src/database/ - Data layer

We have these team agents available:
- code-reviewer for PR reviews
- security-auditor for security checks

And these skills:
- api-design for REST standards
- database-migrations for DB changes

What would you like to explore first?
```

Maintaining Team Configuration

Versioning

Treat `.claude/` as code:

```
# Feature branch for config changes
git checkout -b feat/add-security-agent

# Make changes
mkdir -p .claude/agents
# ... create security-auditor.md

# Commit with message
git add .claude/agents/security-auditor.md
git commit -m "feat(config): add security auditor agent for OWASP checks"

# PR for review
gh pr create --title "Add security auditor agent"
```

Updates

```
# Weekly/monthly review
> /agents
# Review all agents, update as needed
```

```
> /skills
# Review all skills, update documentation

# Update CLAUDE.md with new patterns
# Document new common tasks
```

Communication

When updating team config:

```
# Slack/Teams message
Hey team! 👋

I've added a new security-auditor agent to help catch OWASP issues.

📍 PR: https://github.com/team/repo/pull/456
📖 Usage: Just ask Claude to "run a security audit"

It's in plan mode (read-only) so it's safe to run anytime.

Pull latest main to get it!
```

Security & Privacy

API Key Management

Never Hardcode Keys

Bad:

```
claude config set apiKey sk-ant-123456789
```

Good:

```
export ANTHROPIC_API_KEY=sk-ant-123456789
# Claude automatically reads from environment
```

Team API Keys

For CI/CD or shared environments:

GitHub Secrets:

```
# .github/workflows/review.yml
env:
  ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
```

Setup:

1. Go to repo → Settings → Secrets
2. Add `ANTHROPIC_API_KEY`
3. Reference in workflows

Key Rotation:

```
# Generate new key from dashboard
# Update in all secret stores
# Revoke old key

# Test
claude -p "test" # Should work with new key
```

Sensitive Data Handling

1. Never Commit Secrets

`.gitignore`:

```
.env
.env.local
.claude.json
secrets/
*.pem
*.key
```

2. Sanitize Before Sharing

When sharing Claude sessions:

```
> Summarize our conversation, but remove any API keys, passwords, or
sensitive data
```

3. Use Read-Only Access

For production databases:

```
# Create read-only user
CREATE USER claude_readonly WITH PASSWORD 'secure_pass';
GRANT CONNECT ON DATABASE prod TO claude_readonly;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO claude_readonly;

# Use in MCP
claude mcp add db -- npx -y @bytebase/dbhub \
  --dsn "postgresql://claude_readonly:secure_pass@prod:5432/db"
```

Audit Logging

Track what Claude does:

1. Enable Session Logging

`settings.json`:

```
{
  "logging": {
    "enabled": true,
    "level": "info",
    "directory": ".claude/logs/"
  }
}
```

2. Review Logs Periodically

```
# View recent activity
cat .claude/logs/$(date +%Y-%m-%d).log

# Search for specific actions
grep "Write" .claude/logs/*.log
grep "Bash" .claude/logs/*.log
```

3. Hook-Based Auditing

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Write|Edit|Bash",
        "hooks": [
          {
            "type": "command",
            "command": "echo \"[$(date)] $TOOL_NAME: $FILE_PATH\" >>
```

```
.claude/audit.log"
          }
        ]
      }
    ]
  }
}
```

Enterprise Deployment

Self-Hosted Options

AWS Bedrock:

```
# Configure
claude config set provider bedrock
claude config set region us-east-1

# Use
claude -p "query"
```

Google Vertex AI:

```
claude config set provider vertex
claude config set project your-project-id
```

Azure (via API):

```
export ANTHROPIC_API_URL=https://your-azure-endpoint.com/v1
claude -p "query"
```

Network Isolation

Run Claude Code in restricted network:

```
# Only allow Claude API
firewall-cmd --add-rich-rule='rule family="ipv4" destination
address="api.anthropic.com" accept'

# Block other internet
firewall-cmd --set-default-zone=drop
```

Compliance

For regulated industries:

1. **Data Residency**: Use AWS Bedrock in required region
2. **Audit Logs**: Enable comprehensive logging
3. **Access Control**: Limit who can use API keys
4. **Data Handling**: Never send PII/PHI to Claude without approval
5. **Review**: Periodic audits of Claude usage

Performance Optimization

Response Time Optimization

1. Choose Right Model

Task	Model	Reason
Quick search	Haiku	Fast, cheap, good for simple tasks
Code review	Sonnet	Balanced speed and quality
Architecture	Opus	Best reasoning for complex decisions
Bulk operations	Haiku	Cost-effective for repetitive tasks

Configure per agent:

```
---
name: quick-searcher
model: haiku
---
```

2. Use Subagents for High-Volume

Slow:

```
> Read all 100 test files and find uses of deprecated API
```

Fast:

```
> Use a fast subagent to grep all test files for 'deprecatedAPI' and
return just the file names
```

3. Limit Thinking Budget

For CI/CD where speed matters:

```
export MAX_THINKING_TOKENS=5000
claude -p "quick review of PR"
```

Cost Optimization

1. Monitor Costs

```
# In session
> /cost

# See token usage
# Input tokens: 1,234
# Output tokens: 567
# Estimated cost: $0.05
```

2. Use Cheaper Models When Possible

```
# Expensive (Opus for simple task)
---
name: file-finder
model: opus
---

# Optimized (Haiku sufficient)
---
name: file-finder
model: haiku
---
```

3. Batch Operations

Inefficient:

```
> Fix linting in file1.js
> Fix linting in file2.js
> Fix linting in file3.js
```

Efficient:

```
> Fix linting in all JS files in src/
```

4. Reduce Context Bloat

Use @ mentions sparingly:

```
# Includes entire file (expensive if large)
> Analyze @src/large-file.js

# Better: Specific question
> Does src/large-file.js export a function named 'processData'?
```

Token Usage Patterns

High token usage scenarios:

- Reading many large files
- Generating extensive code
- Extended back-and-forth conversations
- Using Opus model

Low token usage scenarios:

- Targeted questions
- Using Haiku model
- File searches (Grep/Glob)
- Short generation tasks

Error Handling & Recovery

Common Errors and Solutions

API Rate Limits

Error:

```
Error: Rate limit exceeded. Please try again later.
```

Solutions:

```
# 1. Use personal API key (higher limits)
claude config set apiKey YOUR_KEY

# 2. Reduce thinking budget
export MAX_THINKING_TOKENS=5000

# 3. Wait and retry
sleep 60 && claude -p "query"
```

```
# 4. Use Haiku (fewer tokens)
> /model haiku
```

Context Window Full

Error:

```
Error: Context window exceeded
```

Solutions:

```
# 1. Start new session
> /rename current-work
# Exit and start fresh
claude

# 2. Use subagent for high-volume task
> Use a subagent to search all files and return summary

# 3. Be more specific
# Instead of: "Read all files"
# Do: "Search for 'TODO' in src/ and list locations"
```

MCP Connection Failures

Error:

```
Error: Could not connect to MCP server 'github'
```

Solutions:

```
# 1. Check server status
> /mcp

# 2. Verify credentials
claude mcp get github

# 3. Remove and re-add
claude mcp remove github
claude mcp add --transport http github https://api.githubcopilot.com/mcp/

# 4. Check network
curl https://api.githubcopilot.com/health
```

Permission Denied

Error:

```
Error: Permission denied for tool 'Write'
```

Solutions:

```
# 1. Check current mode
> /permissions

# 2. Switch mode
# Press Shift+Tab to cycle modes

# 3. Check path-specific rules
# Review .claude/settings.json

# 4. Grant temporary permission
> [When prompted, select "Allow"]
```

Recovery Strategies

Save Progress Before Risky Operations

```
# Create checkpoint
git add .
git commit -m "wip: checkpoint before refactor"

# Do risky operation with Claude
> Refactor the entire authentication system

# If goes wrong
git reset --hard HEAD
```

Incremental Approach

Instead of:

```
> Refactor the entire codebase to use async/await
```

Do:

```
> List all files using callbacks
[Review list]
> Refactor src/auth.js to use async/await
[Review, test]
> Refactor src/users.js to use async/await
[Review, test]
# ... continue one file at a time
```

Rollback Hooks

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "cp $FILE_PATH $FILE_PATH.claude-backup"
          }
        ]
      }
    ]
  }
}
```

Now every edit is backed up:

```
# Restore if needed
mv src/auth.js.claude-backup src/auth.js
```

Testing Strategies

Testing Custom Agents

Unit Testing Agents

Create test cases:

```
# Test agent with known input
claude --agents "$(cat .claude/agents/security-auditor.md)" \
  -p "Analyze this code: const sql = 'SELECT * FROM users WHERE id = ' +
userId;"
```

Expected output: Should detect SQL injection

Integration Testing

`.claude/tests/agent-tests.sh`:

```
#!/bin/bash

# Test security-auditor
echo "Testing security-auditor..."
result=$(claude -p "Use security-auditor to scan test-
files/vulnerable.js")

if echo "$result" | grep -q "SQL Injection"; then
  echo "✅ Security auditor detected SQL injection"
else
  echo "❌ Security auditor failed to detect issue"
  exit 1
fi

echo "All agent tests passed!"
```

Testing MCP Servers

Local MCP Server Testing

```
# Start server locally
node mcp-server.js &
SERVER_PID=$!

# Test connection
claude mcp add --transport http test-server http://localhost:8080/mcp

# Run tests
claude -p "Test MCP server by listing resources"

# Cleanup
kill $SERVER_PID
claude mcp remove test-server
```

Mock MCP Responses

For CI/CD testing without real APIs:

`mock-mcp-server.js`:

```
import { Server } from '@modelcontextprotocol/sdk/server/index.js';

const server = new Server({ name: 'mock', version: '1.0.0' });

server.setRequestHandler('resources/list', async () => {
  return {
    resources: [
      { uri: 'mock://test', name: 'Test Resource' }
    ]
  };
});

server.setRequestHandler('resources/read', async () => {
  return { contents: [{ uri: 'mock://test', text: 'Mock data' }] };
});
```

Regression Testing

Track Claude's outputs over time:

```
# Create baseline
claude -p "Analyze src/auth.js for security issues" > baseline-auth-
analysis.txt

# After changes, compare
claude -p "Analyze src/auth.js for security issues" > current-auth-
analysis.txt
diff baseline-auth-analysis.txt current-auth-analysis.txt
```

Troubleshooting Guide

General Debugging Process

1. **Check status**: `> /permissions`, `> /mcp`, `> /agents`, `> /skills`
2. **Enable verbose**: `Ctrl+O` to see thinking
3. **Review logs**: Check `.claude/logs/`
4. **Simplify**: Break complex task into smaller parts
5. **Fresh start**: Start new session if context is corrupted

When to Contact Support

- Persistent API errors
- Billing issues
- Feature requests
- Bug reports

How to report:

```
# Include context
claude --version
# Describe issue
# Share logs (sanitize sensitive data)
```

This guide covers advanced patterns for production use of Claude Code. Refer back to sections as you encounter specific scenarios in your workflow.