Context management in Claude Code

Claude Code's responses are only good when you provide the right context. Give it the right files, and it will give excellent answers.

Context is everything.

In the previous lesson, we learned about core commands . Now, let's build on that foundation. Claude Code 's responses are only good when you provide the right context. Give it the right files, and it will give excellent answers. Give it irrelevant, messy stuff, and it will give confusing responses.

Context management isn't simply about adding files. It's about strategically selecting what Claude sees to get the results you need.

How context works

Imagine Claude's context as a desk. Surface area is limited. Everything on the desk is competing for attention.

1. A small, organized workspace: You can find everything you need and work efficiently.
2. A large, cluttered desk: Everything is buried. You lose focus.

Similarly with Claude. A few relevant files are better than dozens of indirectly related files.

The principle of minimum feasible context

Before adding anything, ask yourself: "Does Claude need this for the current mission?"

Yes, please add:

1. The files you are asking about or editing
2. The interface definition files are being used.
3. The files contain the template you want to follow.
4. The test files demonstrate the expected behavior.

No, skip this:

1. The entire "just in case" folder
2. Files indirectly related to the topic
3. The large configuration files that Claude won't need to change.

4. Documents that Claude doesn't need to reference.

When in doubt: Start without context, then add more if Claude seems to be missing information.

Context classification strategy

Build context layer by layer, from general to specific.

Level 1: Understanding the Project

```
/add README.md package.json
```

High-level project information. What is this? Which stack?

Level 2: Architectural Context

```
/add src/index.js src/routes.js
```

Access points and structure. How is this organized?

Layer 3: Feature Context

```
/add src/auth/*.js
```

The specific area where you are working.

Level 4: Task Context

```
/add src/auth/login.js src/auth/types.ts
```

The correct files for your current task.

You don't always need all the classes. For a simple error, skip straight to class 4.

Reading contextual signals

Claude lets you know when the context is wrong:

"I can't see it." Claude is missing a file. Please add it.

Claude is just guessing at the virtual code reference because it doesn't have actual code. Please add the source files.

The style or pattern is inconsistent. Claude hasn't seen your existing code yet. Please add examples of the patterns you want.

The response is very slow. The context might be too large. Try /compact.

The response refers to outdated work and is confusing with obsolete information. Please review `/clear`and revise it.

Task-based context strategy

Fix the error.

```
/clear /add [file có l?i] /add [các file liên quan có th? liên quan] > ?
ây là l?i: [thông báo l?i] > Tìm và s?a l?i.
```

Minimal context, focus only on what's necessary to understand and troubleshoot the problem.

Feature development

```
/clear /add src/similar-feature/* # M?u c?
n tuân theo /add src/types/*.ts # ??nh ngh?a ki?
u /add tests/similar-feature.test.js # Các m?u test > T?o m?t tính n?ng m?
i th?c hi?n X. Tuân theo các m?u hi?n có.
```

Context tells Claude which patterns to follow.

Check the code

```
/add $(git diff --name-only HEAD~1) # Ch? các file ?ã thay ??i > Xem l?
i các thay ??i này ?? tìm l?i, v?n ?? v? ki?u và nh?ng v?n ?? ti?m ?n.
```

Focus on what has actually changed.

Restructuring

```
/add [t?t c? các file liên quan ??n vi?c tái c?u trúc] > Tái c?
u trúc [thành ph?n] ?? s? d?ng [m?u m?i]. > C?p nh?t t?t c? các cách s? d?
ng trên nh?ng file này.
```

All affected files are checked beforehand to coordinate the changes.

Understanding unfamiliar code

```
/add src/mystery-module/* > Gi?i thích cách mô-?un này ho?t ??ng. ?i?
m vào chính là gì? Các ph?n k?t n?i v?i nhau nh? th? nào?
```

Let Claude explore and explain before you begin.

Break down the context for larger tasks.

Some tasks require more context than are appropriate. Solution: Break down the work.

Instead of:

```
/add src/**/* # T?t c? m?i th? - quá nhi?u > Tái c?u trúc toàn b? mã ngu?n
?? s? d?ng TypeScript
```

Please do the following:

```
# Phiên 1 /add src/utils/* > Chuy?n ??i utils sang TypeScript. ?
ây là tsconfig c?a chúng ta. # Phiên 2 /clear /add src/models/* > Chuy?n ??
i models sang TypeScript. ??i chi?u các m?u t?
 utils. # Phiên 3 /clear /add src/services/* > Chuy?n ??
i services sang TypeScript.
```

Each session has a specific context. The work is done step by step.

Practice exercises

Try this context management exercise:

1. Open Claude Code in a project.
2. Adding too many files: `/add **/*.js`
3. Ask a specific question about a function.
4. Note how the answer may reference unrelated code.
5. Now, use it `/clear`and just add the relevant files.
6. Ask the same question
7. Compare the quality of the answers.

Specific context often provides a significant advantage.

Context Management Checklist

Before starting a task:

1. [ ] `/clear`or `/compact`if the preceding context is irrelevant
2. [ ] Determine the minimum number of files required
3. [ ] Add files in a logical order (template first, then target file)
4. [ ] Verify by `/ls`checking if the context is correct.

While performing the task:

1. [ ] Add files as needed if Claude seems to be missing information
2. [ ] Use `/compact`if response slows down
3. [ ] Do not add the "just in case" file

Complete a task:

1. [ ] Consider using this `/compact`before performing the next task if the context will overlap.
2. [ ] `/clear`if switching to an unrelated job

Key points to remember

1. The quality of context is more important than the quantity of context.
2. Minimum feasible context: Only what is necessary for the current task.
3. Context classification ranges from general to specific based on task complexity.
4. Monitor for signals that indicate the context needs adjustment.
5. Break down large tasks into smaller, focused sessions.

1. Question 1:

   When should you add test files to the context?

   1. A. Never - tests are not real code.
   2. B. When writing new code, you need to follow existing test patterns.
   3. C. Always, for every task
   4. D. Only after the code is finished.

   EXPLAIN:

   Test files show Claude the expected test patterns and structures. Adding them when writing new code helps ensure consistency.

2. Question 2:

   What is the principle of 'least feasible context'?

   1. A. Always use the smallest AI model possible.
   2. B. Only add files directly related to the current task.
   3. C. Never add more than one file
   4. D. Delete all files before asking a question.

   EXPLAIN:

   Only add what is necessary for the current task. Relevant context is better than the broader overall context.

3. Question 3:

   What happens when the context is too large?

   1. A. Claude Code is faulty.
   2. B. Slower and less focused responses.
   3. C. Files that are automatically deleted
   4. D. Nothing happens - the more context the better.

   EXPLAIN:

   The large context slows down processing and can confuse the model. Claude has limited attention span for all that information, resulting in a less focused response.

Submit your work

Training results

You have completed **0** questions.

-- / --

Review the lesson

You finished reading the article "**Context management in Claude Code**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.