Contents

Prompt Engineering for AI Assisted Coding

When using Cursor AI, prompt engineering is slightly different from normal ChatGPT prompting because:

- It works inside your codebase
- It has file awareness
- It can refactor, edit, and generate code directly
- Context window management matters more

Cursor works best when you give it:

a. Clear objective
b. Scope (which files?)
c. Constraints (framework, version, style)
d. Expected output (refactor? explanation? patch?)

Weak Prompt (Common Mistake)

Fix this code.

Why is this weak:

- What kind of fix?
- Refactor or debug?
- Keep architecture?
- Improve performance?
- Follow which standard?

Strong Prompt Structure for Cursor

Use this template:

Context:
This is a Spring Boot 3 REST API project using Java 21.

Goal:
Refactor this controller to follow best practices.

Constraints:
- Keep current endpoint paths unchanged
- Use constructor injection
- Apply proper exception handling
- Follow clean code principles

Output:
Modify the file directly and explain major improvements.

Techniques for Cursor Prompting

Tell Cursor the Architecture First

Cursor performs better when you declare architecture.

Example:

This project follows:
- Layered architecture
- Controller → Service → Repository pattern
- JPA with PostgreSQL
- DTO-based request/response

Update the service layer to improve transaction handling.

Without this, it may invent structure.

Use "Act As" Roles

Example:

Act as a senior backend engineer reviewing this code.

Identify:
- Security risks
- Performance bottlenecks
- Violations of SOLID principles

This improves analysis depth dramatically.

Use Multi-Step Prompts (Very Powerful)

Instead of:

Add authentication.

Do this:

Step 1: Analyze current security setup.
Step 2: Suggest best authentication approach for this project.
Step 3: Implement JWT-based authentication.
Step 4: Add necessary configuration classes.

You get controlled evolution instead of chaotic generation.

Use Diff-Based Instructions

Cursor is excellent at controlled modifications.

Instead of:

Improve this function.

Try:

Refactor this function but: *- Do not change its signature* *- Do not modify database schema* *- Keep backward compatibility* *- Reduce cognitive complexity*

This prevents over-engineering.

Use "Explain Before Changing"

Very powerful for learning:

Before modifying the code: *1. Explain what is wrong* *2. Propose 2 solutions* *3. Then implement the best one*

This turns Cursor into a mentor instead of just a generator.

The RTF Framework

To get professional results, you must treat the AI like a specialized colleague. Using the **RTF (Role, Task, Format)** framework prevents generic or "lazy" code.

- **Role:** Define the expertise level (e.g., "Senior Python Developer," "Security Auditor").
- **Task:** Define the specific action (e.g., "Refactor this loop," "Write unit tests").
- **Format:** Define how you want the answer (e.g., "A Git Diff," "A Markdown table," "Commented code").

Example Comparison:

- **Weak Prompt:** "Clean up this code."
- **RTF Prompt:** "Act as a **Senior React Developer**. **Refactor** the following component to use the useReducer hook instead of multiple useState calls. Output the result as a **Side-by-Side Diff**."

Scenario 1: A messy, "junior-level" JavaScript function that needs a professional refactor.

calculator.js:

```
function calc(a, b, type) {
  if (type == "add") {
    return a + b;
  } else if (type == "sub") {
    return a - b;
  } else {
    return "error";
  }
}
```

The Prompt to Demo:
"Act as a **Senior TypeScript Developer**. **Refactor** this function to use an enum for the operation types and ensure the function is strictly typed. **Output** the result as a modern TypeScript file."

Scenario 2: The "Logic & Security" Lab
Practice **RTF Prompting** to fix security flaws and logic gaps in a banking context.

Open this code and use Ctrl + K. Act as a **Senior Security Auditor** and ask the AI to fix the vulnerabilities.

```
// File: bankTransfer.js
function transferFunds(amount, accountId) {
  // BUG: No validation if amount is negative
  // BUG: No check if accountId exists
  console.log("Transferring " + amount + " to " + accountId);

  let balance = 1000;
  balance = balance - amount;

  return "Success! New balance: " + balance;
}
```

Recommended Prompt:
"Act as a **Senior Security Auditor**. **Refactor** this transfer function to: 1. Prevent negative transfers, 2. Ensure the accountId is a 10-digit string, and 3. Check for sufficient balance. **Output** the result as clean, production-ready code."

Chain-of-Thought Prompting

AI models often fail when they try to write 100 lines of code at once. **Chain-of-Thought** prompting forces the AI to "think out loud" and create a logical plan before it touches the keyboard. This significantly reduces "hallucinated" libraries or broken logic.

Step-by-Step Demo: The "Plan-First" Approach

1. Open **Chat (Ctrl + L)**.
2. Input a complex request but end it with a planning instruction:

*"I need a Node.js script that connects to a SQL database and exports transactions to a CSV. **Before writing any code, list the steps and logic you will follow. Wait for my approval.**"*

3. Review the AI's plan. If a step is wrong (e.g., it suggests the wrong library), correct it.
4. Type **"Proceed"** to have it generate the code based on the approved plan.

Scenario 1: Building a complex logic flow for a banking application (matching your interest in the financial sector).

The Prompt to Demo (In Ctrl + L Chat):
*"I need to build a 'Surprise Assumption of Duties' tracker for a bank. It needs to check if a manager has been on mandatory leave for 5 days, and if so, trigger a notification for a 'Surprise Assumption' by a relief officer. **Before writing any code, explain the logic flow and the database schema I would need. Do not write the code yet.**"*

Scenario 2: The "API & Async" Lab
Practice **Chain-of-Thought (CoT)** to handle asynchronous data fetching properly.

This code crashes because it doesn't wait for the data. Use Ctrl + L to ask the AI to **plan** the fix before writing it.

```
// File: userData.js
function getUserProfile(userId) {
  const data = fetch(`https://api.bank.com/users/${userId}`);
  // BUG: 'fetch' is async, but this code treats it as sync
  console.log(data.name);
  return data;
}
```

Recommended Prompt:
*"This code is failing because of asynchronous timing issues. **Before fixing it, explain the steps** needed to implement a proper async/await pattern with a try/catch block for error handling. Once I approve the plan, write the code."*

Zero-shot vs. Few-shot Prompting

Sometimes you want the AI to follow your specific coding "vibe"—like how you name variables or how you handle errors.

- **Zero-shot:** You ask for code with no examples. The AI uses its default style.
- **Few-shot:** You provide 1 or 2 examples of your preferred style first, then ask for the task.

Few-shot Demonstration:

1. Open **Inline Edit (Ctrl + K)**.
2. Provide context by pointing to your existing style:

"Here is how I handle errors in this project: try { … } catch (e) { logError(e); }. Now, using that same style, write a function that fetches user data from @api.js."

3. Observe how the AI mirrors your exact error-handling pattern instead of using a generic console.log.

Scenario 1: Teaching the AI to follow a specific, custom error-handling pattern used in your classroom or company.

Copy this "Pattern Style" into a file named style_guide.js:

```
// OUR PROTECTED PATTERN:
// We always wrap in try-catch and use the custom 'BankLogger' class.
try {
  // logic here
} catch (err) {
  BankLogger.error("Process Failed", { timestamp: Date.now(), details: err });
}
```

The Prompt to Demo (In Ctrl + K while highlighting the code):
*"Using the **same pattern** shown here, write a new function called processLoanApplication that takes a user object and sends it to an API endpoint /v1/apply."*

Scenario 2: The "Style & Consistency" Lab
Practice **Few-shot Prompting** to match a specific UI component style.

Look at the "Pattern" component below. Use Ctrl + K to generate a NEW component (a NotificationCard) that looks exactly like it.

```
<div class="card shadow-sm border-blue">
  <h3 class="text-bold">Transaction Alert</h3>
  <p class="text-muted">Please review your recent activity.</p>
  <button class="btn-primary">View Details</button>
</div>
```

Recommended Prompt:
*"Using the **same HTML structure and CSS classes** (shadow-sm, text-bold, btn-primary) as the Transaction Alert card above, create a new NotificationCard. It should have a title for 'System Update' and a button that says 'Update Now'."*

Cursor-Specific Tips

Be explicit about file scope

Instead of vague requests:

Improve the project.

Say:

Focus only on: *- UserController.java* *- UserService.java* *Ignore other modules.*

Limit hallucinations

If you want strict edits:

- Do not create new classes unless absolutely necessary.
- Do not introduce new dependencies.

Control verbosity

Keep explanations short. *Only explain architectural changes.*

or

Explain in detail because I am learning.

Prompt Patterns for Developers

Project Context:
[Framework, version, database, architecture]

Current Problem:
[Describe clearly]

Objective:
[What you want improved]

Constraints:
- No breaking changes
- Maintain API contract
- Use best practices for [framework]

Quality Requirements:
- Clean code
- SOLID principles
- Production-ready

Output:
- Show diff-style changes
- Explain reasoning

This works incredibly well in Cursor.

Common Cursor Mistakes

- Giving vague commands
- Asking for too many changes at once
- Not specifying constraints
- Not defining architecture
- Letting it rewrite too much