Basic principles of creating prompts for programming.

Master the techniques for creating few-shot prompts, inference sequences, system prompts, and specific model patterns to generate reliable code in OpenAI, Claude, and Gemini.

Master the techniques for creating few-shot prompts, inference sequences, system prompts, and specific model patterns to generate reliable code in OpenAI, Claude, and Gemini.

Each grand programming language (LLM) has its own characteristics. GPT-5 handles complex multi-step prompts well. Claude understands instructions literally and prefers XML tags. Gemini excels at multimodal tasks. Understanding these differences will transform generic prompts into reliable ones.

Let's explore the most important techniques for developers.

System Prompt: Platform Setup

The system prompt defines who the AI ??is, what it does, and how it behaves. For developer use cases, a good system prompt has four parts:

1. **Role** : What is AI? "You are a senior Python developer specializing in data paths."

2. **Task** : What it does. "Create Python functions based on specifications."

3. **Constraints** : What it must do and what it must not do. "Always include type prompts. Never use global variables. Handle errors explicitly."

4. **Output format** : How the response is structured. "Return only Python code. No explanation unless requested."

```
system_prompt = """B?n là m?t l?p trình viên Python cao c?p. NHI?M V?: T?
o các hàm Python t? các thông s? k? thu?t. RÀNG BU?C: - Luôn bao g?
m prompt ki?u d? li?u - S? d?ng tên bi?n mô t? (2-3 t?, vi?t hoa ch? cái ??
u) - X? lý các tr??ng h?p ngo?i l? m?t cách rõ ràng (??u vào tr?ng, giá tr?
 None) - Tuân theo PEP 8 ??U RA: Ch? tr? v? code hàm trong m?t kh?
i code Python. Không gi?i thích tr? khi ???c yêu c?u."""
```

? **Quick check** : Your system prompt says "Write clean code." An engineer on your team asks why the AI ?? output varies so much between requests. What's wrong?

Answer : "Clean code" is subjective. One run might interpret it as minimalist code, another as well-commented code, and yet another as code that uses many design patterns. Replace vague guidelines with specific ones: "Follow PEP 8. Functions under 20 lines. One return statement per function. No comments unless the logic is unclear." Specificity helps reduce the discrepancies.

Few-Shot prompt creation technique: Teaching by example.

Concise instructions are the most ROI-effective technique for developers. Instead of describing what you want, show them what you want.

Sample:

```
D??i ?ây là các ví d? v? ??nh d?ng ??u vào-??u ra tôi c?n: Input: Chuy?n ??
i nhi?t ?? t? ?? C sang ??
 F Output: def celsius_to_fahrenheit(celsius: float) -> float: return celsius * 9
?m tra xem m?t chu?i có ph?i là email h?p l?
 hay không Output: import re def is_valid_email(email: str) -> bool: pattern = r
? hãy t?o: Input: {{user_specification}}
```

Rules for creating effective few-shot examples:

1. 3-5 examples is the optimal number — fewer and the model will guess; the greater the contextual pressure, the more harmful it is.
2. Diverse examples—including varying levels of complexity, exceptions, and patterns.
3. Typical examples — if you want to show error handling, demonstrate how to handle errors in examples.
4. Placed within tags — XML tags () help Claude parse the structure; blocks of back quotes work with GPT

Chain of reasoning: Logic plays a crucial role.

The CoT prompt technique requires the model to display its reasoning before responding. Research shows this improves the Pass@1 code generation rate by up to 16%.

When should you use CoT?

1. Complex logic with multiple steps
2. Data transformation with business rules
3. Implement the algorithm
4. Debugging and reviewing the code

When should you skip CoT?

1. Simple CRUD operations
2. Formatting/conversion tasks
3. Tasks that require speed over accuracy.

Prompt CoT has a structure for the code:

```
Hãy suy ngh? theo t?ng b??c sau: 1. Hi?u các yêu c?u 2. Xác ??nh các tr??
ng h?p ngo?i l? 3. Ch?n thu?t toán/ph??ng pháp 4. Tri?n khai t?ng b??
c 5. Xác minh vi?c tri?n khai x? lý t?t c? các tr??ng h?p ngo?i l?
 Task: {{task_description}}
```

? **Quick test** : You use string inference for a simple function that converts a string to capitalize the first letter of each word. The model generates 30 lines of reasoning for a function that is only 3 lines long. Is CoT always better?

Answer : No. For simple tasks, CoT increases latency and token cost without improving quality. The model knows how to capitalize the first letter of the chain. CoT is for tasks where the model might go wrong for no apparent reason. Reserve CoT for complex logic, multi-step transformations, and algorithmic problems.

Specific patterns of the model

OpenAI (GPT-4.1, GPT-5)

1. GPT-5 handles merged multi-step prompts — give it the entire task at once instead of splitting it up.
2. Use it `response_format`with a JSON schema to get structured output (Lesson 3 covers this).
3. Pin to specific model versions in production: gpt-4.1-2025-04-14
4. Supports natural function calls — defining tools as JSON schema.

Anthropic (Claude IV)

1. Claude took instructions literally — be clear about what you want AND what you don't want.
2. XML tags effectively structure prompt data: , , ,
3. Expanded thinking (parameters `thinking`) allows Claude to reason internally before responding.
4. Cache prompt reduces costs by up to 90% for repetitive static content.

Google (Gemini)

1. Powerful multimodal capabilities - send images, PDFs, and text prompts.
2. Structured output via`response_mime_type: "application/json"`
3. Suitable for tasks that combine text with visual analysis (examining screenshots, understanding diagrams).

CRISP Framework

A simple structure for any developer prompt:

1. Context: Background information, source code context, expertise.
2. Role: Who should AI be (senior developer, security reviewer, etc.)
3. Instructions: Specific tasks with constraints
4. Technical specifications: Output format, language, style requirements
5. Complete: For example, exceptions, verification steps

Key points to remember

1. The system prompt needs four parts: Role, task, constraints, and output format - vague instructions produce inconsistent output.
2. The technique of creating few-shot prompts (3-5 diverse examples) is the highest ROI technique for achieving consistent output quality.
3. The inference chain improves complex tasks by 16% but adds unnecessary overhead to simple tasks.
4. Understand your model: Claude is literal (let's be clear), GPT-5 handles merge prompts, Gemini excels at multimodals.
5. Pin specific models to the versions currently in production — updating the model may break working prompts.

1. Question 1:

   Claude literally interpreted the system instruction 'Use descriptive variable names' and created variables like `the_total_sum_of_all_prices_after_tax`. What should you change?

   1. A. Completely disregard this guidance – the confusion between correlation and causality here leads to ineffective strategies.
   2. B. Convert to GPT
   3. C. More specifically: 'Use descriptive but concise variable names (e.g., total_price, not x, but not the_total_sum_of_all_prices_after_tax). Follow Python's naming conventions.' Claude took the instruction literally – specificity prevents over-compliance.

   EXPLAIN:

   Anthropic's documentation also states that Claude takes instructions literally. If you say 'describe,' it will be a maximum description. The solution: Limit it with examples and scope. 'Describe but concisely (maximum 2-3 words)' accompanied by examples will give Claude a clear objective. This literal understanding is actually a strength when you're learning how to write accurate instructions.

2. Question 2:

   You are using a prompt string inference for a complex data transformation. The model's reasoning is correct, but the final result contains errors. What is the most likely scenario?

   1. A. String inference methods are inefficient for programming code - a common misunderstanding leading to suboptimal results.
   2. B. The model made the correct inference but encountered errors during implementation.
   3. C. Use a different model.

   EXPLAIN:

   Chain reasoning (CoT) improves reasoning ability by up to 16% (scoring a 1 on code tests), but there is still a difference between correct reasoning and correct implementation. The solution: require the model to self-check its code based on its own reasoning. This self-checking step will detect implementation errors that might slip through even if the logic is correct. Chain reasoning improves reasoning ability but does

not guarantee correct code. Add clear guidance: 'After thoroughly reasoning the method, implement it step by step. Then, verify your implementation against each step of your reasoning.'

3. Question 3:

   You add 3 few-shot examples to the code generation prompt. The output quality improves significantly. You add 15 more examples. The quality plateaus and sometimes even deteriorates. Why?

   1. A. More examples will consume context window tokens, leaving the model with less space to reason and generate code.
   2. B. The model cannot count beyond 3 – this sounds reasonable but overlooks a crucial differentiating factor.
   3. C. You need exactly 10 examples.

   EXPLAIN:

   Prompt few-shot examples follow a quality curve. 0 examples: model guesses. 3-5 diverse examples: model understands the pattern and generalizes well. 10+ examples: contextual window pressure, and the model may over-fit the examples instead of the instructions. Ongoing research shows that 3-5 diverse, carefully selected examples perform better than larger sets. Beyond 3-5 diverse examples, you get diminishing efficiency—the model starts matching patterns to the examples instead of understanding the task. Few-shot examples are of higher quality than quantity.

Submit your work

Training results

You have completed **0** questions.

-- / --

Review the lesson