ChatGPT

Enhancing Small LLM Performance with Prompt Engineering

Introduction: Small language models (like Devstral Small 2) often give minimal or shallow answers by default. Unlike giant models that can infer a lot from short prompts, smaller LLMs need extra guidance to produce thorough, useful results [1]. The good news is that you can **boost a small model's output quality through clever prompting techniques**, all without any fine-tuning. Below are several prompting strategies – especially useful for coding tasks – that you can add to the system prompt to encourage the model to do more work, think deeper, and handle complex requests.

1. Use a Strong Expert Role in the System Prompt

Begin by **assigning the model a role or persona** that implies expertise and diligence. This sets the tone for all its responses. For example, a system prompt could say: *"You are a highly skilled software engineer AI who thoroughly analyzes problems and produces complete, step-by-step solutions."* This kind of role prompt has been shown to improve the depth of answers. In community experiments, simply starting with **"You are an expert AI assistant"** followed by detailed instructions led a 27B-parameter local model to give *"excellent answers"* on difficult questions [2]. Setting an expert or teacher persona guides the LLM to respond in a more detailed and helpful manner [3]. It implicitly tells the model that *"doing more work"* is expected.

Tips for role prompts: Describe the desired expertise (e.g. *"expert Python developer"*), work ethic (e.g. *"never gives up on a complex task"*), and style (e.g. *"explains solutions with clear comments"*). This encourages the model to adopt those qualities in its output.

2. Encourage Step-by-Step Reasoning (Chain-of-Thought)

One of the most effective ways to get a small LLM to tackle complex problems is to explicitly **prompt it to think step by step**. This is known as *chain-of-thought (CoT) prompting*. Instead of expecting the model to output a direct answer, **instruct it to explain its reasoning and break the problem into sub-tasks**. For example, you might add to the prompt: *"Let's solve this step by step. First, analyze the problem and outline a plan, then execute the plan one step at a time, explaining each step."*

Large models naturally exhibit this behavior, but **even smaller models can do better if guided to "think out loud"** [4]. Research has found that prompting a model to *"explain your answer step-by-step"* significantly improves accuracy on multi-step reasoning tasks [4]. OpenAI's guidelines likewise suggest **splitting complex tasks into simpler subtasks and giving the model time to 'think'** [5] – essentially telling the model to show its work.

In practice, this means your system prompt can include directives like:

- *"Break the problem into clear sub-problems and solve them one by one."*

- *“Show your thought process before giving the final answer.”*
- *“Use a numbered list to reason through the task logically.”*

By **walking the model through the reasoning process**, you prevent it from lazily jumping to a quick (and possibly incorrect or incomplete) answer. For coding, this might involve having it outline the program structure or logic first. In one coding example, an engineer’s prompt explicitly asked ChatGPT **to outline an implementation plan rather than writing code immediately** [6]. This chain-of-thought strategy helped the model produce a well-thought-out solution plan before coding. Similarly, Google’s small 2B model *Gemma* greatly improved its output when the prompt was expanded to include reasoning steps (an “analysis” before the final answer) – the **more elaborate, step-by-step prompt yielded far more accurate results** than a minimal prompt [7]. The takeaway: **tell the model to think step by step**, and it will work through the problem more diligently.

3. Break Down Tasks and Plan Before Coding

When giving a coding assignment to a small LLM, it helps to **have the model first produce a plan or pseudocode** as an intermediate step. This is a specific application of the step-by-step approach. Instead of directly saying “Write code to do X,” you can structure the prompt in phases: e.g. *“First, outline your plan for the program (what functions or steps will you need?). Then write the full code according to that plan.”*

This accomplishes two things: (1) it forces the model to **consider the entire task and all its components** before diving into code, and (2) it gives you a chance to see the model’s approach (which you or the model itself can refine) before final output. In one example, a developer **used a prompt that explicitly said “Don’t generate code yet.”** Instead, the model had to describe the solution and break it down as a task list (the “master plan”) [8] [9]. Only after this outline was the model prompted to generate the actual code. This technique, sometimes called *“Generated Knowledge prompting,”* resulted in more organized and correct code, because the model had a strong scaffold to follow [10] [11].

How to apply this: Add instructions in the system prompt like: *“Before writing any code, describe your solution approach in detail (e.g. list modules, functions, or steps). Only then proceed to write the code implementation.”* By having the model **think and plan first**, you increase its “work throughput” on the task – it will likely cover edge cases and necessary components in the plan, leading to a more complete final answer.

4. Incorporate Self-Checking and Reflection

Small LLMs can be prone to mistakes or to stopping once they’ve produced *something* that looks like an answer. You can counteract this by instructing the model to **review and critique its own solution before finalizing it**. In other words, build a *self-reflection* phase into the prompt. For example, you might say: *“After solving the problem, double-check your solution. Look for errors or missing pieces and fix them if found before giving the final answer.”*

A more elaborate approach is to ask for a brief **reflection after each major step** in its reasoning. In one shared prompt (designed for local 7B–30B models), the system message told the model to include a `<reflection>` after each idea where it *“Review[s] your reasoning, check[s] for potential errors or oversights, and confirm[s] or adjust[s] your conclusion if necessary”* [12]. Only then would it provide the final answer. This kind of self-critique instruction can **push the model to be more thorough** and not settle for the first draft of the answer. In fact, prompting LLMs to critique their own outputs is a proven technique for improving answer quality – studies have shown it *“has shown great success in helping models refine and improve their responses.”* [13] .

For coding tasks, you can leverage this by asking the model to, for instance, **inspect its code for errors or test it mentally**. A system prompt addition could be: *“After writing the code, go through it step by step to verify it meets all requirements and has no bugs or syntax errors. Only then provide the final code.”* This encourages the model not only to write more complete code, but also to catch mistakes or omissions, essentially increasing the effort it invests in the task.

5. Provide Examples or Templates (Few-Shot Prompting)

If the model still isn’t as effective as you’d like, consider **demonstrating the desired thorough output via examples**. Small models benefit from examples of how to do a task correctly [14] . For instance, you can include a short example of an input and an exemplary output in the prompt (this is called *few-shot prompting*). In coding, that might be a tiny sample function and a well-commented solution, or a pseudo Q&A where a “user” asks for a code solution and the “assistant” provides a detailed answer. This shows the model the format and level of detail you expect.

Make sure the examples **highlight the behavior you want**: e.g. comprehensive reasoning, edge-case handling, comments, etc. Google’s AI guide on smaller LLMs found that giving a few examples plus using chain-of-thought significantly improved a tiny model’s performance [14] [7] . In their case (rating product reviews), the small model initially gave incorrect or incomplete answers, but after adding **several examples and an analysis-before-answer format**, the outputs became accurate [7] . Essentially, the model learned from the prompt how to do the task better.

When providing examples, also **be explicit about output format**. If you expect a certain structure (like a JSON, or in our case maybe a full code block followed by an explanation), say so clearly. Smaller models are less likely to *infer* what format you need – you must tell them. Clarity and specificity in instructions are key: *“Write the final answer as a Python code block, and then explain the code in one paragraph.”* Such precise guidance focuses the model on doing all parts of the job.

6. Emphasize Thoroughness and Persistence

Finally, it can help to directly **tell the model to be thorough and not skip hard parts**. Don’t hesitate to reinforce this mindset in the system prompt. For example: *“The assistant never refuses a legitimate request and never gives a partial answer. It always tries its best to produce a complete solution, even if the problem is complex.”* While the model might still have limits, this nudge can reduce the tendency of smaller LLMs to do the bare minimum. Also encourage it to **consider edge cases or additional aspects** of the task: *“Think of any edge cases or additional improvements and include them in your solution.”*

Such instructions, combined with the techniques above, will make the model more “eager” to tackle large, complicated tasks rather than stopping early. In summary, **be as explicit as possible about expecting a detailed, extensive answer**. As one set of prompting principles puts it: *“Write clear instructions... Split complex tasks into simpler subtasks... Give the model time to 'think'.”* [5] All of these boil down to guiding the model to put in more effort. With a well-crafted system prompt, even a small LLM can exhibit surprisingly robust performance on coding problems by reasoning systematically and not giving up until the task is done.

Sources:

- Google Developers, *Practical Prompt Engineering for Smaller LLMs* [1] [14] [7]
- Reddit – r/LocalLLaMA community prompt for small models [2] [12]
- Martin Fowler, *LLM Prompting for Programming* (Xu Hao's ChatGPT strategy) [6] [9]
- Learn Prompting, *Self-Criticism Techniques* [13]
- OpenAI/Medium, *Prompt Engineering Best Practices* [5] [3]
- IBM, *Chain-of-Thought Prompting Explanation* [4]

---

[1] [7] [14] Practical prompt engineering for smaller LLMs | Articles | web.dev
https://web.dev/articles/practical-prompt-engineering

[2] [12] Ingenious prompts for smaller models: reaching PhD level with local models? : r/LocalLLaMA
https://www.reddit.com/r/LocalLLaMA/comments/1fhyj7e/ingenious_prompts_for_smaller_models_reaching_phd/

[3] [5] Best Prompt Techniques for Best LLM Responses | by Jules S. Damji | The Modern Scientist | Medium
https://medium.com/the-modern-scientist/best-prompt-techniques-for-best-llm-responses-24d2ff4f6bca

[4] What is chain of thought (CoT) prompting? | IBM
https://www.ibm.com/think/topics/chain-of-thoughts

[6] [8] [9] [10] [11] An example of LLM prompting for programming
https://martinfowler.com/articles/2023-chatgpt-xu-hao.html

[13] Introduction to Self-Criticism Prompting Techniques for LLMs
https://learnprompting.org/docs/advanced/self_criticism/introduction?srsltid=AfmBOopm2ekgaOtiCsJZPFpHO92wki62LaGC-nQ9wfvh8k6SNmavzlHi