Build Chain-of-Thought From Scratch - Tutoria for Dummies

ZACHARY HUANG
APR 16, 2025

> *Ever ask an AI a tricky question and get a hilariously wrong answer? Turns out, AI ne learn how to "think step-by-step" too! This guide shows you how to build that thinking process from scratch, explained simply with code using the tiny PocketFlow framework*

We've all seen it. You ask an AI assistant to solve a multi-step math problem, plan complex trip, or tack w completely misses th en with super-smart AI

Often, it's because th g i through." When *we* face tough problems, we break them down, figure out the

maybe jot down some notes, and work towards the solution piece by piece. Good r
we can teach AI to do the same!

This step-by-step reasoning is called **Chain-of-Thought** (**CoT**), and it's a *huge* deal
AI right now. It's why the latest cutting-edge models like **OpenAI's O1, DeepSeek
Google's Gemini 2.5 Pro, and Anthropic's Claude 3.7** are all emphasizing better
reasoning and planning abilities. They know that **thinking** is key to truly smart A

Instead of just talking about it, **we're going to build a basic Chain-of-Thought sy
ourselves, from scratch!** In this beginner-friendly tutorial, you'll learn:

- What Chain-of-Thought really means (in simple terms).
- How a basic CoT loop works using code.
- How to make an AI plan, execute steps, and even evaluate its own work.

---

*We'll use **PocketFlow** – a super-simple, 100-line Python framework. Forget complicat
frameworks that hide everything! PocketFlow lets you see exactly how the thinking proce
built, making it perfect for understanding the core ideas from the ground up. Let's teach o
to show its work!*

Chain-of-Thought: How AI Learns to Show Its Work

So, what exactly *is* th n A
solve problems like a us
making a wild guess.

Here's how the AI tackles a problem using Chain-of-Thought:

1. **Understand the Case:** First, the AI looks carefully at the problem you gave it like a detective studying the initial crime scene and available clues.
2. **Make an Initial Plan:** Based on the problem, the AI drafts a high-level plan – of steps it thinks it needs to take.

> Detective: "Okay, first I need to interview the witness, then check the alibi, the analyze the fingerprints."

3. **Execute One Step:** The AI focuses on *just the first* pending step in its plan and the work required for that step.

> Detective: Conducts the first witness interview.

4. **Analyze the Results** (**Self-Critique!**): After completing the step, the AI looks a result. Did it work? Does the information make sense? Is it leading towards t solution? This evaluation step is crucial.

> Detective: "Hmm, the witness seemed nervous. Does their story match the kno facts?"

5. **Update the Case File & Plan:** The AI records the results of the step (like addi notes to a case file) and *updates its plan*. It marks the step as done, adds any ne findings, and crucially, might *change* or add future steps based on the evaluati

> Detective: "Okay, interview done. Add 'Verify witness alibi' to the plan. And m 'Re-interview later if alibi doesn't check out'."

6. **Repeat Until Solved:** The AI loops back to Step 3, executing the *next* step on i updated plan, analyzing, and refining until the entire plan is complete and the problem is solved.

The Core Idea: Inste lvi into a cycle of **Plan -**

PocketFlow

Alright, so we understand the *idea* behind Chain-of-Thought – making the AI act a detective, step-by-step. But how do we actually *build* this process in code withou getting lost in a giant, confusing mess?

This is where [PocketFlow] comes in. Forget massive AI frameworks that feel like trying to learn rocket science overnight. PocketFlow is the opposite: it's a brillian simple tool, **exactly 100 lines of Python code**, designed to make building AI workflows crystal clear.

Think of PocketFlow like setting up a mini automated assembly line for our AI's thinking process:

- **Shared Store** (**The Central Conveyor Belt**): This is just a standard Python dictionary (e.g., `shared = {}`). It holds all the information – the problem, th plan, the history of thoughts – and carries it between steps. Every step can rea from and write to this shared dictionary.

```
# Our 'conveyor belt' starts with the problem
shared = {"problem": "Calculate (15 + 5) * 3 / 2", "thoughts": []}
```

- **Node** (**A Worker Station**): This is a Python class representing a single task or stage on our assembly line. For CoT, we'll mostly use just *one* main worker sta node. Each node has a standard way of working defined by three core method

```
class Node:
    def __init__(self): self.successors = {}
    def prep(self, shared): pass
    def exec(self, prep_res): pass
    def post(sel
    def run(self
self.post(shared
```


- Here's what tho

1. `prep`: Gets the station ready by pulling necessary data *from* the `shared` s (conveyor belt).
2. `exec`: Performs the station's main job using the prepared data. This is wh our CoT node will call the LLM.
3. `post`: Takes the result, updates the `shared` store, and signals which stati should run next (or if the process should loop or end).

- **Flow (The Assembly Line Manager):** This is the controller that directs the `shared` data from one `Node` to the next based on the signals returned by the `post` method. You define the connections between stations.

```
class Flow(Node):
    def __init__(self, start): self.start = start
    def _orch(self,shared,params=None):
        curr = self.start
        while curr: action=curr.run(shared);
curr=curr.successors.get(action)
```

- Connecting nodes is super intuitive:

```
# Define our main thinking station
thinker_node = ChainOfThoughtNode()

# Define the connections for looping and ending
thinker_node - "continue" >> thinker_node # Loop back on "continue"
thinker_node - "end" >> None # Stop on "end" (or go to a final node)

# Create the flow manager, starting with our thinker node
flow = Flow(start=thinker_node)
# Run the assemb'
flow.run(shared)
```

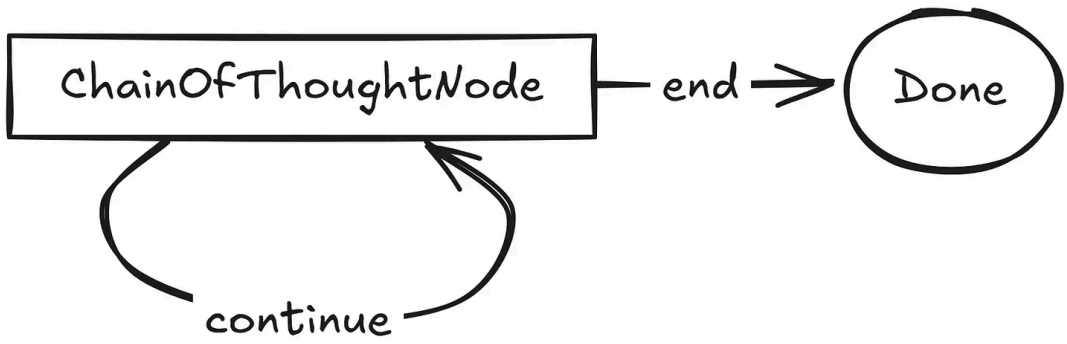
Why PocketFlow Rocks for This: *Its simplicity and clear structure (`Node`, `Flow`, `Shar Store`) make it incredibly easy to visualize and build even sophisticated processes like Chain-of-Thought loop, RAG systems, or complex agent behaviors, without getting bog down in framework complexity. You see exactly how the data flows and how the logic w*

Chain-of-Thought: It's Just a Loop!

Now, here's the really neat part about implementing Chain-of-Thought with PocketFlow, something often obscured by complex frameworks:

> **The entire step-by-step thinking process can be handled by *one single Node* looping back onto itself.**

Forget needing a complicated chain of different nodes for planning, executing, evaluating, etc. We can build this intelligent behavior using an incredibly simple g structure:



That's it! This simple graph structure powers our CoT engine:

1. **One Core Node** (
   contains all the l
   evaluating steps
2. **One Loop Edge**
   the node finishe

`"continue"` signal.

3. **One Exit Edge** (`"end"`): This arrow points away (potentially to `None` or a fina reporting node). When the node determines the problem is solved, it sends th `"end"` signal.
4. **PocketFlow's Role:** The `Flow` manager simply follows these arrows. It runs tl `ChainOfThoughtNode`, gets the signal (`"continue"` or `"end"`) from its `po` method, and routes execution accordingly based on the connections we defin (`thinker_node - "continue" >> thinker_node`).

No complex state machines, no hidden magic – just one node whose internal logic decides whether to continue the loop or finish. The sophisticated, step-by-step reasoning emerges from the combination of:

- The **intelligence** packed inside the `ChainOfThoughtNode` (specifically, how uses the LLM in its `exec` step).
- This ultra-simple **looping graph structure** enabled by PocketFlow.

Let's look inside that `ChainOfThoughtNode` to see how it makes the loop work.

Inside the Thinker Node: The Prep-Think-Upda Cycle

Our `ChainOfThoughtNode` is where the step-by-step magic happens. Each time PocketFlow runs this node (which is every loop cycle), it executes a simple three-p process defined by its `prep`, `exec`, and `post` methods. Let's walk through this cy using our example problem: `Calculate (15 + 5) * 3 / 2`. We'll show simp Python code for each step first, then explain it.

1. `prep`: Gathe

First, the node needs ona (our central data stor

```
def prep(self, shared):
    problem = shared.get("problem")
    thoughts_history = shared.get("thoughts", [])
    # Helper to get the latest plan from the history
    current_plan = self.get_latest_plan(thoughts_history)
    return {"problem": problem, "history": thoughts_history, "plan":
current_plan}
```

- **Explanation:** This `prep` method acts like getting ingredients before cooking. looks into the `shared` dictionary and pulls out:
  - The original `problem` statement.
  - The `thoughts` history: A list containing the results (thinking, plan) from previous cycles. It starts empty.
  - The `current_plan`: Usually found within the *last* entry in the `thought` history. If the history is empty, the plan is initially `None`.
- **Example (Start of Loop 1):** `shared` is `{"problem": "Calculate (15 + * 3 / 2", "thoughts": []}`. `prep` returns `{"problem": "...", "history": [], "plan": None}`.
- **Example (Start of Loop 2):** `shared` now contains the result of Loop 1, like `{"thoughts": [{"thinking": "...", "planning": [...], ...}` `prep` returns `{"problem": "...", "history": [thought1], "plan plan_from_thought1}`.

2. exec: The Core Thinking - Prompting the LLM

This is the main event where we ask the LLM (AI Brain) to perform one cycle of Chain-of-Thought reasoning. The key is the prompt we send.


```
def exec(self, p
    # prep_res h
    problem = pr
    history_text = self.format_history(prep_res["history"]) # Make
history readable
```

```
    plan_text = self.format_plan(prep_res["plan"]) # Make plan readab

    # --- Construct the Prompt ---
    prompt = f"""
You are solving a problem step-by-step.

Problem: {problem}

History (Previous Steps):
{history_text}

Current Plan:
{plan_text}

Your Task Now:
1. Evaluate the LATEST step in History (if any). State if
correct/incorrect.
2. Execute the NEXT 'Pending' step in the Current Plan. Show your wor
3. Update the FULL plan (mark step 'Done', add 'result', fix if neede
4. Decide if the overall problem is solved (`next_thought_needed:
true/false`).

Output ONLY in YAML format:
```yaml
thinking: |
  Evaluation: <Your evaluation>
  <Your work for the current step>
planning:
  # The FULL updated plan list
  - description: ...
    status: ...
    result: ...
next_thought_needed: <true or false>
```
"""
    # --- Call LLM and Parse ---
    llm_response   ` Al
    # Assume ext
    structured_o
yaml.safe_load(s
    return struc
```

- **Explanation:** The `exec` method orchestrates the LLM interaction:
  1. It takes the `problem`, `history`, and `plan` provided by `prep`.
  2. It formats the history and plan into readable text.
  3. It builds the **prompt**. Notice how the prompt explicitly tells the LLM the steps: **Evaluate** -> **Execute** -> **Update Plan** -> **Decide Completion**. It also specifies the required YAML output format.
  4. It calls the LLM with this detailed prompt.
  5. It receives the LLM's response and parses the structured YAML part into Python dictionary (e.g., `{'thinking': '...', 'planning': [...] 'next_thought_needed': True}`).
- **Example Prompt Built (Start of Loop 2):**

```
You are solving a problem step-by-step.

Problem: Calculate (15 + 5) * 3 / 2

History (Previous Steps):
Thought 1:
  thinking: |
    Evaluation: N/A
    Calculate 15 + 5 = 20.
  planning: [...] # Plan with step 1 Done

Current Plan:
- {'description': 'Calculate 15 + 5', 'status': 'Done', 'result': 20}
- {'description': 'Multiply result by 3', 'status': 'Pending'}
# ... other steps ...

Your Task Now:
1. Evaluate the
2. Execute the N
# ... etc ...

Output ONLY in Y
# ...
```

3. `post`: Saving Progress & Signaling the Loop

After the `exec` step gets the LLM's response, `post` saves the progress and tells PocketFlow whether to continue looping or stop.

```
def post(self, shared, prep_res, exec_res):
    # exec_res is the structured dict from exec() like {'thinking': .
'planning': ..., 'next_thought_needed': ...}
    if "thoughts" not in shared: shared["thoughts"] = []
    shared["thoughts"].append(exec_res) # Add the latest thought cycl
results to history

    # Decide the signal based on LLM's decision
    if exec_res.get("next_thought_needed", True):
        signal = "continue"
    else:
        # Optional: Extract final answer for convenience
        shared["final_answer"] =
self.find_final_result(exec_res.get("planning", []))
        signal = "end"

    return signal # Return "continue" or "end" to PocketFlow
```

- **Explanation:** The `post` method is the cleanup and signaling step:
  1. It takes the dictionary returned by `exec` (containing the LLM's latest `thinking`, `planning`, and `next_thought_needed` flag).
  2. It appends this entire dictionary to the `thoughts` list in the `shared` dictionary, building our step-by-step record.
  3. It checks the `next_thought_needed` flag.
  4. It returns the "continue" if the LLM indicat
- **Example (End o** `True, ...}`. p `"continue"`.

- **Example (End of Final Loop):** `exec_res` contains `{'next_thought_need` `False, ...}`. `post` appends this, maybe saves the final answer, and returns `"end"`.

Making it Loop: The PocketFlow `Flow`

So, the `ChainOfThoughtNode` completes one cycle (`prep` -> `exec` -> `post`) and returns a signal (`"continue"` or `"end"`). But how does the *looping* actually happe That's managed by the PocketFlow `Flow` object, which acts like our assembly line manager.

Here's how we set it up in code:

```
# 1. Create an instance of our thinker node
thinker_node = ChainOfThoughtNode()

# 2. Define the connections (the workflow rules)
# If the node signals "continue", loop back to itself
thinker_node - "continue" >> thinker_node
# If the node signals "end", stop the flow (or go to a final node)
thinker_node - "end" >> None

# 3. Create the Flow manager, telling it where to start
cot_flow = Flow(start=thinker_node)

# Now we can run the process
# Remember 'shared' holds our initial problem and accumulates thought
shared = {"problem": "Calculate (15 + 5) * 3 / 2", "thoughts": []}
cot_flow.run(shared)

# After run() finishes, shared["final_answer"] (if set in post)
# and shared["thoughts"] will contain the full process history.
```


This combination of `Flow`'s simple routir d t ou loop.

See It In Action: Solving a Problem Step-by-St

Let's watch our Chain-of-Thought loop tackle a problem. For illustration, we'll us simple multi-step arithmetic calculation: **Calculate `(15 + 5) * 3 / 2`.**

> *Note: While this example is simple, this exact looping structure can handle much more complex reasoning, including problems like the* ***Jane Street die-rolling probability puzz*** often used in quant interviews. To see the detailed output for that challenging problem, check out the [Chain-of-Thought Cookbook example on GitHub](#).

Here's a summary of what happens inside the loop when the `Flow` runs, focusing the LLM's output and highlighting the plan changes:

Loop 1 / Thought 1: Handling the Parentheses

- **LLM Task:** Create initial plan, execute step 1.
- **LLM Output (YAML):** *(Showing full plan this time)*

```
thinking: |
  Evaluation: N/A (First step)
  Problem is (15 + 5) * 3 / 2. Handle parentheses first.
  Calculate 15 + 5 = 20.
planning:
  - {'description': 'Calculate parentheses (15 + 5)', 'status': 'Done
'result': 20} # <-- Step Executed & Updated
  - {'description': 'Multiply result by 3', 'status': 'Pending'}
  - {'description': 'Divide result by 2', 'status': 'Pending'}
  - {'description': 'State final answer', 'status': 'Pending'}
next_thought_needed: true
```

- `post` **Result:** Sa

Loop 2 / Thou

- **LLM Task:** Eval
- **LLM Output (YAML):** *(Using ... for unchanged parts)*

```
thinking: |
  Evaluation: Correct. Addition 15 + 5 = 20 was right.
  Next: Multiply previous result (20) by 3.
  Calculate 20 * 3 = 60.
planning:
  - {'description': 'Calculate parentheses (15 + 5)', 'status': 'Done
'result': 20}
  - {'description': 'Multiply result by 3', 'status': 'Done', 'result
60} # <-- Step Executed & Updated
  - {'description': 'Divide result by 2', 'status': 'Pending'}
  - {'description': 'State final answer', 'status': 'Pending'}
next_thought_needed: true
```

- `post` **Result:** Saves Thought 2. Returns `"continue"`.

Loop 3 / Thought 3: Division

- **LLM Task:** Evaluate Thought 2, execute step 3.
- **LLM Output** (YAML): *(Using ... for unchanged parts)*

```
thinking: |
  Evaluation: Correct. Multiplication 20 * 3 = 60 was right.
  Next: Divide previous result (60) by 2.
  Calculate 60 / 2 = 30.
planning:
  - {'description': 'Calculate parentheses (15 + 5)', 'status': 'Done
'result': 20}
  - {'description': 'Multiply result by 3', 'status': 'Done', 'result
60}
  - {'description': 'Divide result by 2', 'status': 'Done', 'result':
30} # <-- Step Executed & Updated
  - {'description': 'State final answer', 'status': 'Pending'}
next_thought_nee
```


- `post` **Result:** Sa

Loop 4 / Thought 4: Conclusion

- **LLM Task:** Evaluate Thought 3, execute step 4 (Conclusion).
- **LLM Output** (YAML): *(Using ... for unchanged parts)*

```
thinking: |
  Evaluation: Correct. Division 60 / 2 = 30 was right.
  All calculation steps are done. The final result is 30.
planning:
  - {'description': 'Calculate parentheses (15 + 5)', 'status': 'Done
'result': 20}
  - {'description': 'Multiply result by 3', 'status': 'Done', 'result
60}
  - {'description': 'Divide result by 2', 'status': 'Done', 'result':
30}
  - {'description': 'State final answer', 'status': 'Done', 'result':
30} # <-- Step Executed & Updated
next_thought_needed: false
```

- `post` **Result:** Saves Thought 4. Saves `30` to `shared['final_answer']`. Re `"end"`. The `Flow` stops.

Beyond Our Loop: When AI Learns How to Thi

The method we built orchestrates thinking from the *outside*. However, some cuttir edge AI models are being developed to handle complex reasoning more autonomc essentially building the "thinking" process *internally*. Examples include models explicitly trained for reasoning like **OpenAI's O1, DeepSeek-R1, Google's Gemin Pro, and Anthropic's Claude 3.7**.

A key concept powering some of these advanced models is the powerful synergy between step-by-step **Thinking** and automated **Learning** (like Reinforcement Learning). Think abc bc game strategy or solv g random things, but b

1. **Thinking throug**

2. Seeing the outcome.
3. **Reflecting** on which parts of your thinking led to success or failure.

That "thinking hard" part makes the learning stick.

AI Models are Learning This Too (e.g., DeepSeek-R1's Approach):

- **Generate Thinking:** The AI attempts to solve a problem by generating its int reasoning or "thinking" steps.
- **Produce Answer:** Based on its thinking, it comes up with a final answer.
- **Check Outcome:** An automated system checks if the *final answer* is actually correct.
- **Get Reward:** If the answer is correct, the AI gets a positive "reward" signal.
- **Learn & Improve:** This reward signal teaches the AI! It updates its internal parameters to make it more likely to use the *successful thinking patterns* again r time.

This cycle of think -> check answer -> get reward -> learn helps the AI become genuinely at reasoning over time. (For the nitty-gritty details on how DeepSeek-R1 does this wit techniques like GRPO, check out their [research paper](#)!)

Conclusion: You've Taught an AI to Think Step by-Step!

Now you know the secret behind making AI tackle complex problems: **Chain-of-Thought is just a str**

1. **Plan:** Create or
2. **Evaluate:** Check
3. **Execute:** Perform the *next single step* in the plan.

4. **Update:** Record the results and refine the plan based on the evaluation and execution.
5. **Loop (or End):** Decide whether to repeat the cycle or finish.

The "thinking" – evaluating, executing, planning, and deciding – happens within t LLM, guided by the **prompt** we construct in our `exec` method. The **looping** itself effortlessly handled by **PocketFlow's simple graph structure** and the `continue` / signals from our `post` method. Everything else is just organizing the information

Next time you hear about advanced AI reasoning, remember this simple **Evaluate Execute -> Update** loop. You now have the fundamental building block to underst how AI can "show its work" and solve problems piece by piece.

Armed with this knowledge, you're ready to explore more complex reasoning syst

---

Want to see the full code and try the challenging Jane Street quant interview question? C out the [PocketFlow Chain-of-Thought Cookbook on GitHub](#)!

Write a comment...

avi Apr 17

Liked by Zachary Huang

Nice. Does this work only with reasoning models ? Or any LLM? And have you experimentec fine tuning an LLM to implicitly do this cot ?

LIKE (2) REPLY

1 reply by Zachary Huang

Alex Rosenfeld Apr 28

Liked by Zachary Huang

Really appreciate your ability demonstrate the effectiveness of this flow with such an elegar simple framework!

LIKE (1) REPLY

1 reply by Zachary Huang

2 more comments...