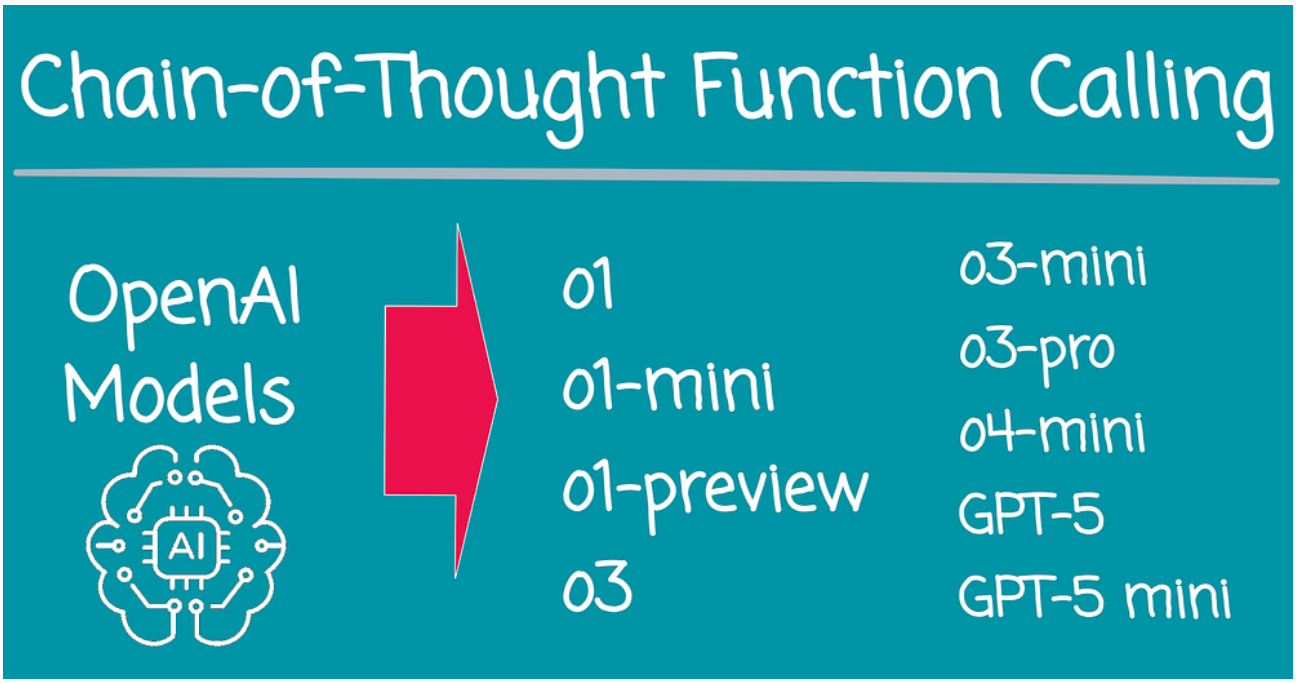
Chain-of-Thought Function Calling

Chain-of-Thought Function Calling (CoT-FC) is the idea that a model doesn't just answer and then call a function — rather, it reasons & while reasoning it decides when to call tools/functions, integrates their results into its train of thought, and keeps going. The tool calls are woven into the model's internal reasoning, not bolted on externally.

Cobus Greyling • 9 min read • 2025-08-25

https://cobusgreyling.medium.com/chain-of-thought-function-calling-2a4f84f0f9c1




```
==========================================================================================================
                    Old Orchestration                     |                    CoT-Woven Tool Use
==========================================================================================================
Developer: ask model to check delivery                    | Model: let's check delivery
Tool says: {'order_id': 'A100', 'delivered': True}        | -> {'order_id': 'A100', 'delivered': True}
Developer: ask model to check policy                      | Model: delivered. check policy
Tool says: {'order_id': 'A100', 'eligible': True}         | -> {'order_id': 'A100', 'eligible': True}
Developer: ask model to create refund                     | Model: eligible. compute amount via tool (python repl)
Tool says: {'refund_id': 'R-A100-XYZ', 'amount': 20.0}    | -> {'code': 'math.sqrt(16) * 5', 'result': 20.0}
Developer: ask model to notify user                       | Model: create refund
Tool says: {'user_id': 'U1', 'notified': True, 'mes…      | -> {'refund_id': 'R-A100-XYZ', 'amount': 20.0}
DONE (old style).                                         | Model: notify user
                                                          | -> {'user_id': 'U1', 'notified': True, 'message': '…
                                                          | DONE (woven).
==========================================================================================================
```

Output from the Python Notebook below….

Ceding Too much Autonomy /Control To The Model

This is a danger and one of the big **trade-offs** with Chain-of-Thought Function Calling and agent-style architectures…

Because the model decides **when** and **how** many times to call tools, devs can unintentionally

cede too much autonomy if you don't set boundaries.

Risks

The model might loop, retry or chain many tool calls in ways you didn't anticipate — potentially expensive (*API usage, database queries*) or unsafe (*too many writes*).

If you expose sensitive or dangerous tools (for instances one that modifies customer data), the model could pick it at the wrong time or with wrong parameters.

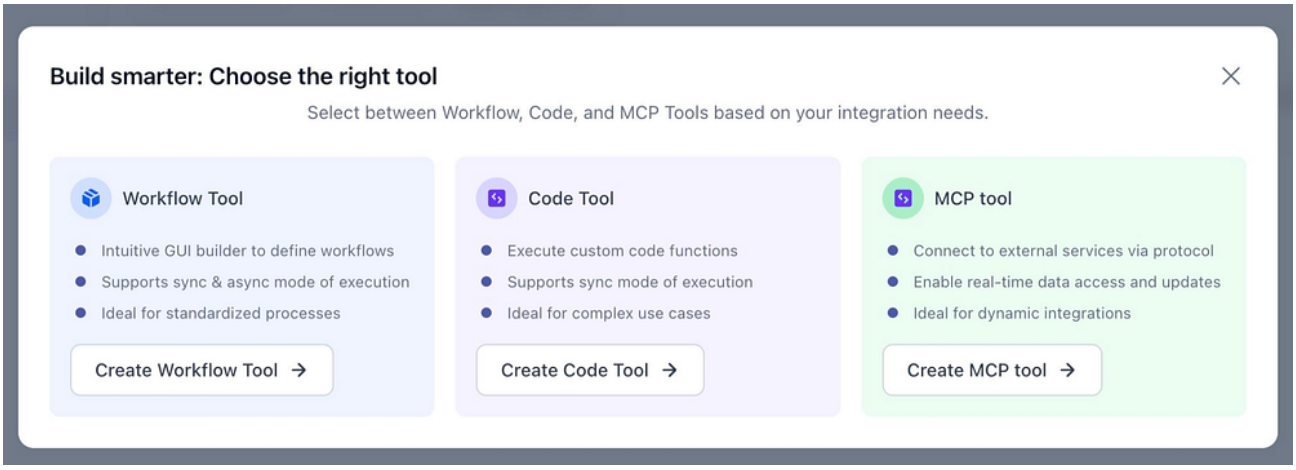
This an example of where a workflow can work better, and hence an approach of appropriating the best tool for the best use-case is important.

Notice below the three tool types catered for in an AI Agent build framework.

There is a workflow tool…more sequential and predictable for sensitive processes.

Code tools custom integrations; or where the AI Agent can write/edit the code.

And lastly, MCP tools…



Source

Since you don't see the *full* internal chain-of-thought, you only see the *effect* (the tool call). You don't always know *why* it made that choice.

If you rely entirely on the model for workflows (like refunds), you're letting it decide policy enforcement — which can conflict with your rules.

What Is Internal — The Model Does Automatically

This happens **inside the model's reasoning loop**, invisible to us…

You don't program it step-by-step — the model decides:

When to call a function/tool

The model chooses the moment in its chain of thought where a call makes sense. You don't say "call this function now," you just define what exists.

Which tool to call

If you define three tools, the model might pick one over another depending on the task.

How many times to call it

It might call a tool once, retry it, or chain multiple calls together, for example, check order status, then compute a refund then send notification.

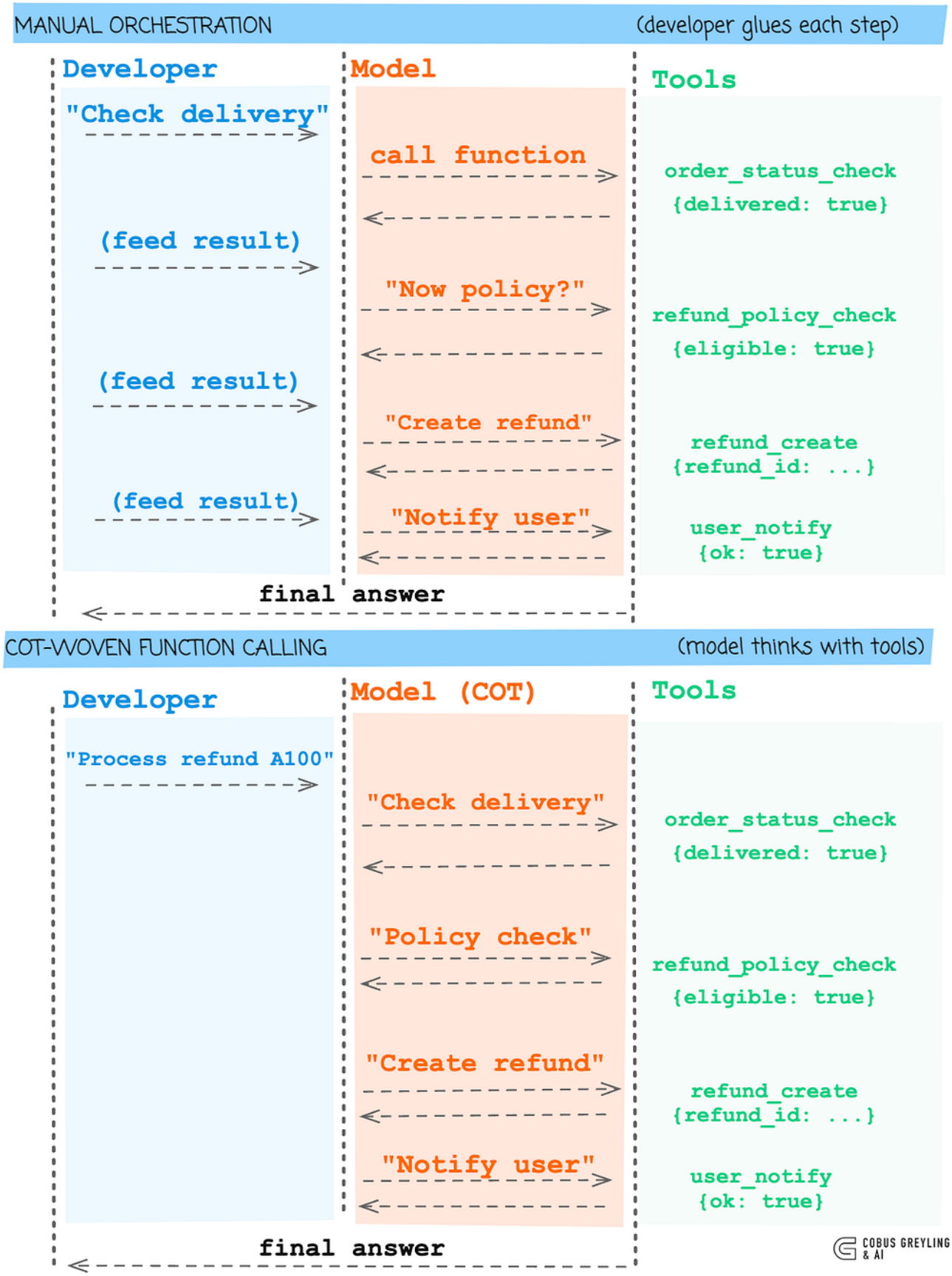
How to fold results back into reasoning

The tool's output is treated like ***new knowledge*** for the model. It immediately continues reasoning with it in context, without you piping data back in manually.

Internal chain-of-thought reasoning

The model may be testing ideas, planning next steps, or *thinking aloud* before deciding to call a tool.

You don't see all of this, but the function call you do see is a product of that reasoning.



What is External — What Devs Can Control

This is the **surface area of the API** — what you configure and enforce…

Which functions/tools exist

You define the JSON schema for each function, or enable built-in tools (like `python`, `web-search`, `file-search`).

The “sandbox” they run in

You decide what environment tools can access — for example, can the model run Python, hit your database, or only a mock API…

Constraints & validation

You can enforce strict schemas, for example refund amount must be a number, reject malformed calls, and re-prompt the model.

Execution policy

Limit how many calls are allowed, decide if parallel calls are enable, control retries, timeouts, error handling.

```
#########

!pip -q install openai

#########

import os, math, getpass
from openai import OpenAI

#########

os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter your OpenAI API key: ")
client = OpenAI(api_key=os.environ["OPENAI_API_KEY"])

#########

# --- Simulated business "functions" (structured, predictable schemas) ---
def order_status_check(order_id: str):
    """Return delivery status in a strict schema."""
    db = {"A100": True, "B200": False}
    return {"order_id": order_id, "delivered": db.get(order_id, False)}

def refund_policy_check(order_id: str):
    """Return eligibility in a strict schema."""
    # Let's say B200 is not eligible
    eligible = order_id != "B200"
    return {"order_id": order_id, "eligible": eligible}

def refund_create(order_id: str, amount: float = 100.0):
    """Return refund id in a strict schema."""
    return {"refund_id": f"R-{order_id}-XYZ", "amount": amount}
```

```
def user_notify(user_id: str, message: str):
    """Return notification outcome in a strict schema."""
    return {"user_id": user_id, "notified": True, "message": message}

#########

# --- Example 'tool' (broader capability) ---
def python_repl(code: str):
    """Very tiny, sandboxed evaluator for demo purposes."""
    try:
        # expose only math
        result = eval(code, {"__builtins__": {},"math": math}, {})
        return {"code": code, "result": result}
    except Exception as e:
        return {"code": code, "error": str(e)}

#########

# --- OLD orchestration: developer glues steps manually ---
def process_refund_old(order_id: str, user_id: str):
    log = []
    log.append("Developer: ask model to check delivery")
    status = order_status_check(order_id)
    log.append(f"Tool says: {status}")
    if not status["delivered"]:
        log.append("STOP: not delivered.")
        return log

    log.append("Developer: ask model to check policy")
    policy = refund_policy_check(order_id)
    log.append(f"Tool says: {policy}")
    if not policy["eligible"]:
        log.append("STOP: not eligible.")
        return log

    log.append("Developer: ask model to create refund")
    refund = refund_create(order_id, amount=python_repl("math.sqrt(16) *
5").get("result", 100))
    log.append(f"Tool says: {refund}")

    log.append("Developer: ask model to notify user")
    note = user_notify(user_id, f"Refund {refund['refund_id']} created for
${refund['amount']}.")
    log.append(f"Tool says: {note}")
    log.append("DONE (old style).")
    return log

#########
```

```
# --- CoT-woven (simulated): model "thinks" and calls tools inline ---
def process_refund_woven(order_id: str, user_id: str):
    log = []
    log.append("Model: let's check delivery")
    status = order_status_check(order_id)
    log.append(f"-> {status}")
    if not status["delivered"]:
        log.append("STOP: not delivered.")
        return log

    log.append("Model: delivered. check policy")
    policy = refund_policy_check(order_id)
    log.append(f"-> {policy}")
    if not policy["eligible"]:
        log.append("STOP: not eligible.")
        return log

    log.append("Model: eligible. compute amount via tool (python repl)")
    calc = python_repl("math.sqrt(16) * 5")  # 4 * 5 = 20.0
    log.append(f"-> {calc}")
    amt = calc.get("result", 100.0)

    log.append("Model: create refund")
    refund = refund_create(order_id, amount=amt)
    log.append(f"-> {refund}")

    log.append("Model: notify user")
    note = user_notify(user_id, f"Refund {refund['refund_id']} created for
${refund['amount']}.")
    log.append(f"-> {note}")

    log.append("DONE (woven).")
    return log

#########

# --- Run both and print side-by-side ---
def print_side_by_side(left, right, w=54):
    max_len = max(len(left), len(right))
    left += [""]*(max_len - len(left))
    right += [""]*(max_len - len(right))
    print("="*(w*2+7))
    print(f"{'Old Orchestration':^{w}} | {'CoT-Woven Tool Use':^{w}}")
    print("="*(w*2+7))
    for a,b in zip(left,right):
        la = (a if len(a)<=w else a[:w-3]+"…").ljust(w)
        rb = (b if len(b)<=w else b[:w-3]+"…").ljust(w)
        print(f"{la} | {rb}")
    print("="*(w*2+7))
```

```
old_log  = process_refund_old("A100", "U1")
new_log  = process_refund_woven("A100", "U1")
print_side_by_side(old_log, new_log)
```

The output…

```
==================================================================================================
                   Old Orchestration                    |
CoT-Woven Tool Use
==================================================================================================
Developer: ask model to check delivery                  | Model: let's check
delivery
Tool says: {'order_id': 'A100', 'delivered': True}      | -> {'order_id': 'A100',
'delivered': True}
Developer: ask model to check policy                    | Model: delivered. check
policy
Tool says: {'order_id': 'A100', 'eligible': True}       | -> {'order_id': 'A100',
'eligible': True}
Developer: ask model to create refund                   | Model: eligible.
compute amount via tool (python repl)
Tool says: {'refund_id': 'R-A100-XYZ', 'amount': 20.0} | -> {'code':
'math.sqrt(16) * 5', 'result': 20.0}
Developer: ask model to notify user                     | Model: create refund
Tool says: {'user_id': 'U1', 'notified': True, 'mes…    | -> {'refund_id':
'R-A100-XYZ', 'amount': 20.0}
DONE (old style).                                       | Model: notify user
                                                        | -> {'user_id': 'U1',
'notified': True, 'message': '…
                                                        | DONE (woven).
==================================================================================================
```

```
# --- Optional: ask GPT-5 mini to summarize (uses your API key) ---
summary = client.responses.create(
    model="gpt-5-mini",
    input=[{
        "role":"user",
        "content":(
            "Given these two logs, explain the difference between function calls
"
            "and tool use, and why CoT-woven calls reduce orchestration:\n\n"
            f"OLD:\n{old_log}\n\nNEW:\n{new_log}"
        )
    }]
)
print("\n--- GPT-5 mini summary ---\n")
print(summary.output_text)
```

The output…

--- GPT-5 mini summary ---

Short answer
- Function calls = developer-driven, structured invocations of external APIs/tools (the system asks the tool and then feeds the result back to the model).
- Tool use (CoT woven calls) = the model itse tools as part of its internal reasoning, so the model sequences calls and uses results inline with its chain-of-thought.
- CoT woven calls reduce orchestration becaus developer!' model!' tool round trips into a singl calls, less middleware state handling, and fewer separate prompts.

Explain with the logs

OLD (developer-orchestrated):
- Developer asks model to "check delivery" !'
- Developer asks model to "check policy" !' to
- Developer asks model to "create refund" !' t
- Developer asks model to "notify user" !' too
This requires the developer/system to: issue each tool call, collect results, re-insert results into the model prompt/context, and manage sequencing, error handling and state. Each step is an explicit orchestration step (many round trips).

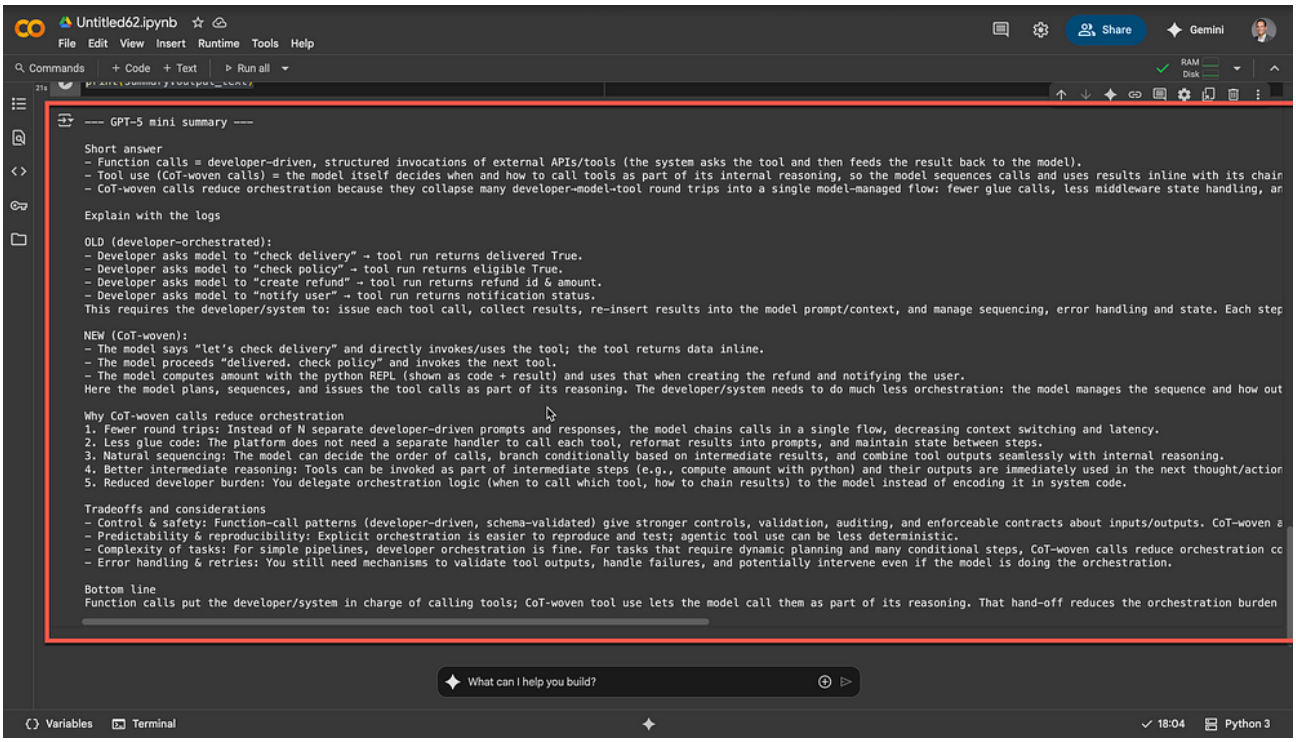
NEW (CoT woven):
- The model says "let's check delivery" and directly invokes/uses the tool; the tool returns data inline.
- The model proceeds "delivered. check policy" and invokes the next tool.
- The model computes amount with the python REPL (shown as code + result) and uses that when creating the refund and notifying the user.
Here the model plans, sequences, and issues the tool calls as part of its reasoning. The developer/system needs to do much less orchestration: the model manages the sequence and how outputs are consumed.

Why CoT woven calls reduce orchestration
1. Fewer round trips: Instead of N separate developer-driven prompts and responses, the model chains calls in a single flow, decreasing context switching and latency.
2. Less glue code: The platform does not need a separate handler to call each tool, reformat results into prompts, and maintain state between steps.
3. Natural sequencing: The model can decide the order of calls, branch conditionally based on intermediate results, and combine tool outputs seamlessly with internal reasoning.
4. Better intermediate reasoning: Tools can be invoked as part of intermediate steps (e.g., compute amount with python) and their outputs are immediately used in the next thought/action without manual re-injection.
5. Reduced developer burden: You delegate orchestration logic (when to call which tool, how to chain results) to the model instead of encoding it in system code.

```
Tradeoffs and considerations
- Control & safety: Function-call patterns (developer-driven, schema-validated)
give stronger controls, validation, auditing, and enforceable contracts about
inputs/outputs. CoT-woven agentic calls require careful sandboxing, monitoring
and result validation.
- Predictability & reproducibility: Explicit orchestration is easier to
reproduce and test; agentic tool use can be less deterministic.
- Complexity of tasks: For simple pipelines, developer orchestration is fine.
For tasks that require dynamic planning and many conditional steps, CoT-woven
calls reduce orchestration complexity and latency.
- Error handling & retries: You still need mechanisms to validate tool outputs,
handle failures, and potentially intervene even if the model is doing the
orchestration.

Bottom line
 F u n c t i o n  c a l l s  p u t  t h e  d e v e l o p e r / s y s t e m  i n  c h
tool use lets the model call them as part of its reasoning. That hand-off
reduces the orchestration burden (fewer explicit round trips and less glue
logic), but it shifts responsibility for safe, validated execution and auditing
to the system that exposes tools to the model.
```

From the notebook…



Chief Evangelist *@ Kore.ai | I'm passionate about exploring the intersection of AI and language. Language Models, AI Agents, Agentic Apps, Dev Frameworks & Data-Driven Tools shaping tomorrow.*

Other mentions by Author

- www.cobusgreyling.com | COBUS GREYLING
- cobusgreyling.medium.com | Chain-of-Thought Function Calling (CoT-FC) is the idea that a model doesn't just answer and then call a function — rather, it reasons & while reasoning it decides when to call tools/functions, integrates their results into its train of thought, and keeps going. The tool calls are woven into the model's internal reasoning, not bolted on externally.