ALL THINGS OPEN 2026 · RALEIGH

The Architecture of AI Agents



Patterns, Protocols, and Pitfalls in 2026

Arturo Parra · Senior Software Engineer

WHO AM I?

Arturo Parra

Senior Software Engineer · Charlotte, NC

- 20+ years building enterprise backends in Java and Python
- Day job: workflow and platform engineering at scale
- Exploring AI agent architectures and MCP
- Writes at programandojava.com



AGENDA

01

Workflows vs agents

The five workflow patterns, in diagrams and Python

02

Agents

Anatomy · loops (ReAct, Ralph) · protocols (MCP, OKF)

03

Conclusion

Takeaways and Q&A

PART I

Workflows vs Agents

Who decides the path: your code, or the model?

Two kinds of agentic systems

Workflows

“LLMs and tools are orchestrated through predefined code paths.”

- Your code decides the path
- Predictable, cheaper, easy to test

Agents

“LLMs dynamically direct their own processes and tool usage.”

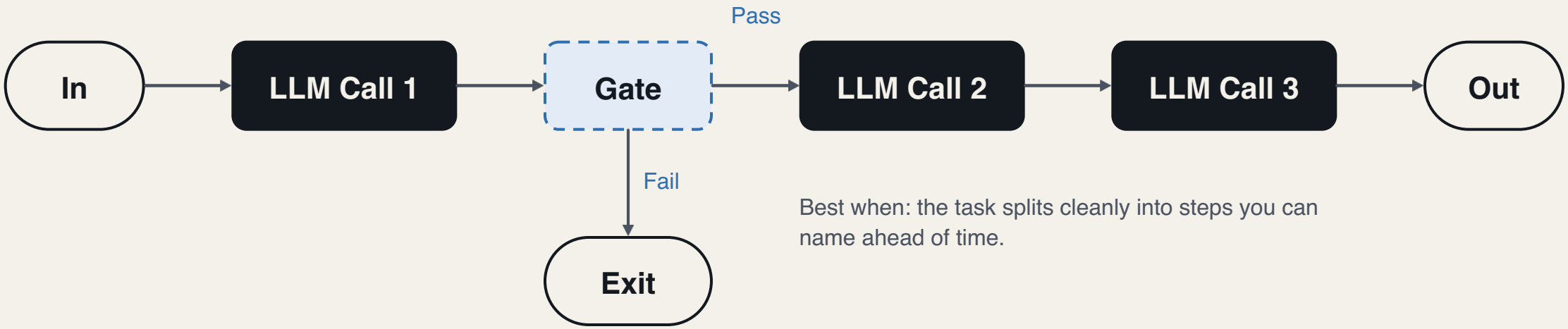
- The model decides the path
- Flexible, but costlier and less predictable

Source: Anthropic, “Building Effective Agents” (December 2024)

WORKFLOW 1 OF 5

Prompt chaining

Split a task into fixed steps. Each call works on the previous output, with optional code checks (gates) in between.



Best when: the task splits cleanly into steps you can name ahead of time.

Prompt chaining in Python

```
def write_post(topic: str) -> str:
    outline = llm(f"Write an outline for a post about {topic}")

    # Gate: a plain code check between steps
    if not passes_checks(outline):
        raise ValueError("Outline failed the gate")

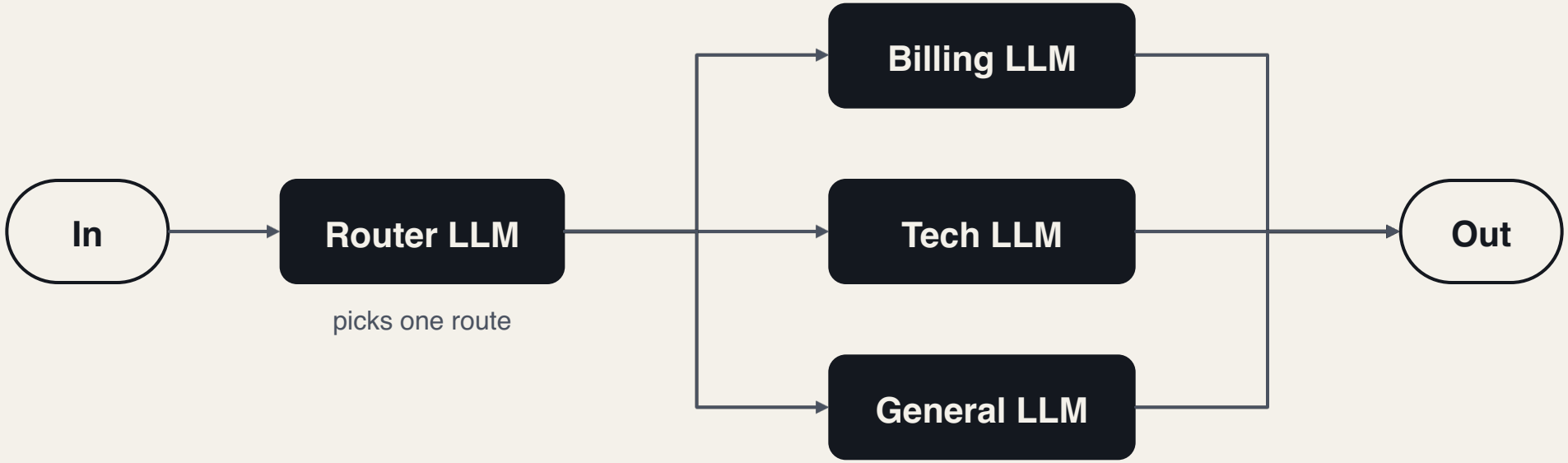
    draft = llm(f"Write the post from this outline:\n{outline}")
    return llm(f"Translate it to Spanish:\n{draft}")
```

llm() = one call to any model API. Plain functions, no framework.

WORKFLOW 2 OF 5

Routing

Classify the input, then send it to a specialized prompt, tool set or model. Each route can be tuned on its own.



Routing in Python

```
ROUTES = {
    "billing":   ("haiku",  "You are a billing specialist..."),
    "technical": ("opus",   "You are a senior support engineer..."),
    "general":   ("sonnet", "You are a friendly support agent..."),
}

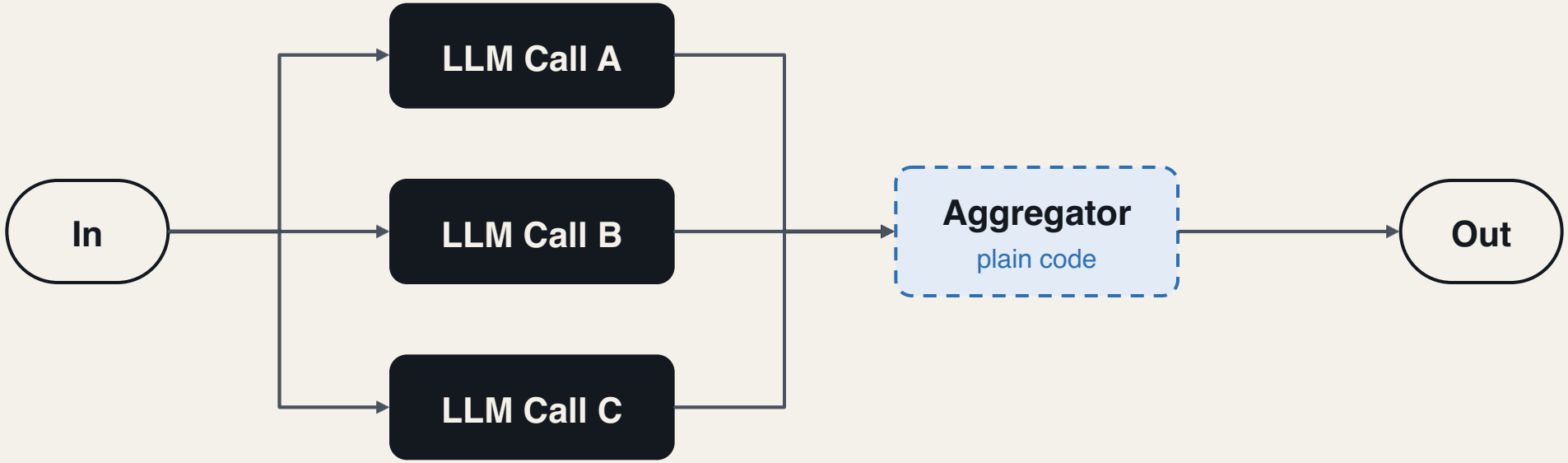
def handle(ticket: str) -> str:
    label = llm(f"Reply billing, technical or general:\n{ticket}")
    model, system = ROUTES.get(label.strip(), ROUTES["general"])
    return llm(ticket, system=system, model=model)
```

Each route gets its own prompt and its own model: cheap where you can, strong where you must.

WORKFLOW 3 OF 5

Parallelization

Run independent calls at the same time. **Sectioning** splits a task into parts; **voting** runs the same task several times and compares.



Parallelization in Python

```python
import asyncio

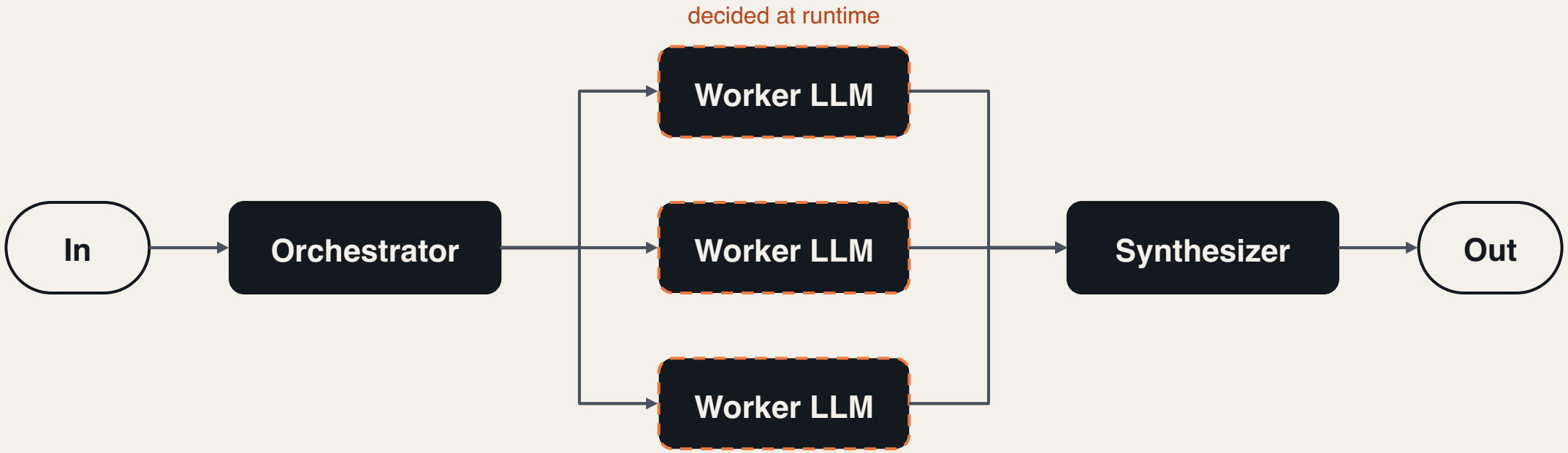
async def review(code: str) -> dict:
    # Sectioning: independent checks, run at the same time
    security, perf, style = await asyncio.gather(
        allm(f"Review for security issues:\n{code}"),
        allm(f"Review for performance issues:\n{code}"),
        allm(f"Review for style issues:\n{code}"),
    )
    # Aggregate in plain code
    return {"security": security, "perf": perf, "style": style}
```

allm() = the async version of llm(). Voting: same prompt N times, take the majority.

WORKFLOW 4 OF 5

Orchestrator-workers

A central LLM breaks the task down at runtime, hands pieces to workers, then combines the results. The subtasks aren’t known in advance.



Orchestrator-workers in Python

```
async def implement(feature: str) -> str:
    # The orchestrator decides the subtasks at runtime
    plan = json.loads(await allm(
        f"Split into subtasks. Return a JSON list:\n{feature}"))

    # Workers: one call per subtask, in parallel
    results = await asyncio.gather(
        *(allm(f"Do this subtask:\n{task}") for task in plan))

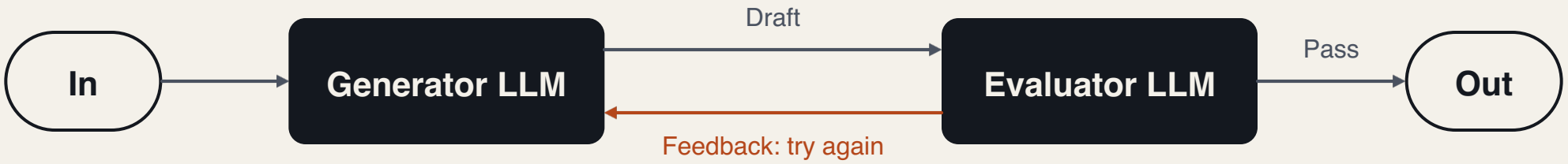
    # Synthesizer
    return await allm(f"Combine these results:\n{results}")
```

The difference from parallelization: the length of plan isn’t known until the model answers.

WORKFLOW 5 OF 5

Evaluator-optimizer

One LLM generates, another evaluates and gives feedback, in a loop until it passes. Works when you have clear criteria.



Always cap the rounds. Remember this one: it comes back in Part II.

Evaluator-optimizer in Python

```
def generate(task: str, max_rounds: int = 3) -> str:
    feedback = ""
    for _ in range(max_rounds):
        draft = llm(f"{task}\n{feedback}")
        verdict = llm(f"Reply PASS or give feedback:\n{draft}")
        if verdict.startswith("PASS"):
            return draft
        feedback = f"Last try:\n{draft}\nFeedback:\n{verdict}"
    return draft  # best effort after max_rounds
```

The feedback lives in memory, inside one run. Hold that thought.

Patterns are a vocabulary, not a recipe

“These building blocks aren’t prescriptive. They’re common patterns that developers can shape and combine to fit different use cases.”

Route, then chain

Each route runs its own prompt chain.

Workers that self-check

Each worker runs its own evaluator-optimizer loop.

Tweak freely

Gates, retries and a human approval can slot in anywhere.

Source: Anthropic, “Building Effective Agents”

PART II

Agents

What they are, the loops that drive them, and the protocols that connect them

WHAT IS AN AGENT?

“…typically just LLMs using tools based on environmental feedback in a loop.”

Anthropic, “Building Effective Agents”



Picks its next step



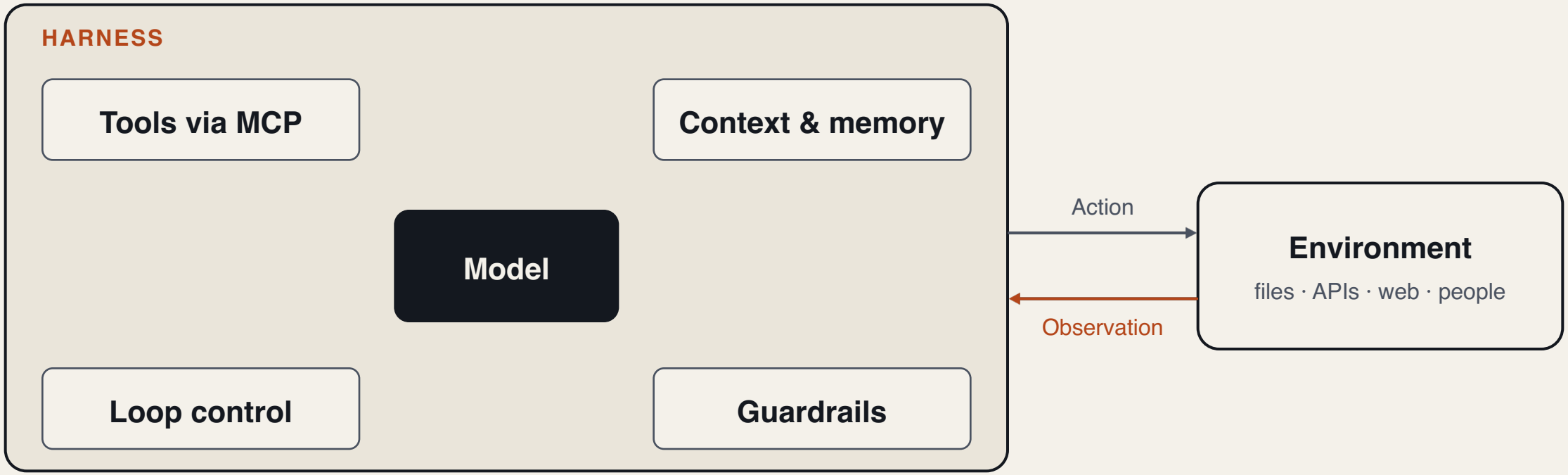
Uses tools to act



Decides when it’s done

ANATOMY OF AN AGENT

Agent = Model + Harness



Agents in the wild

Coding agents

Claude Code, Codex: read a repo, edit, run tests, commit

Research agents

Search many sources, cross-check, write a report

Computer use

Operate apps and websites by clicking and typing

Ops & support

Triage tickets, look things up, act with approval

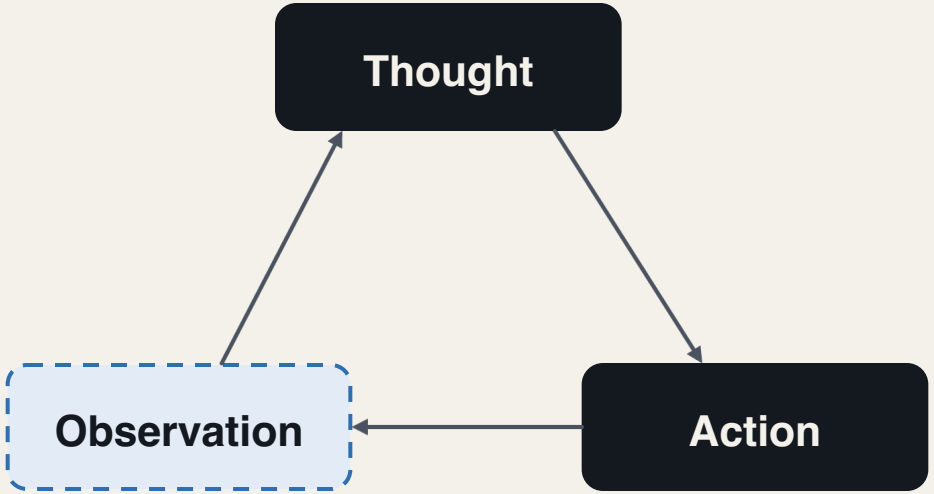
PATTERNS

Pattern loops

ReAct inside a session. Ralph around sessions.

PATTERN LOOP 1 · INNER

ReAct: reason + act



…repeat until the model decides it’s done

EXAMPLE TRACE

Thought: I need the meeting location.
Action: calendar.next_meeting()
Observation: "Raleigh Convention Center"
Thought: Now I need the weather there.
Action: weather.get(city="Raleigh")
Observation: "72°F and sunny"
Answer: 72°F and sunny at your meeting.

Yao et al., “ReAct: Synergizing Reasoning and Acting in Language Models” (2022)

PATTERN LOOP 2 · OUTER

Ralph: a workflow around an agent

```
while :; do
  cat PROMPT.md | claude -p
done
```

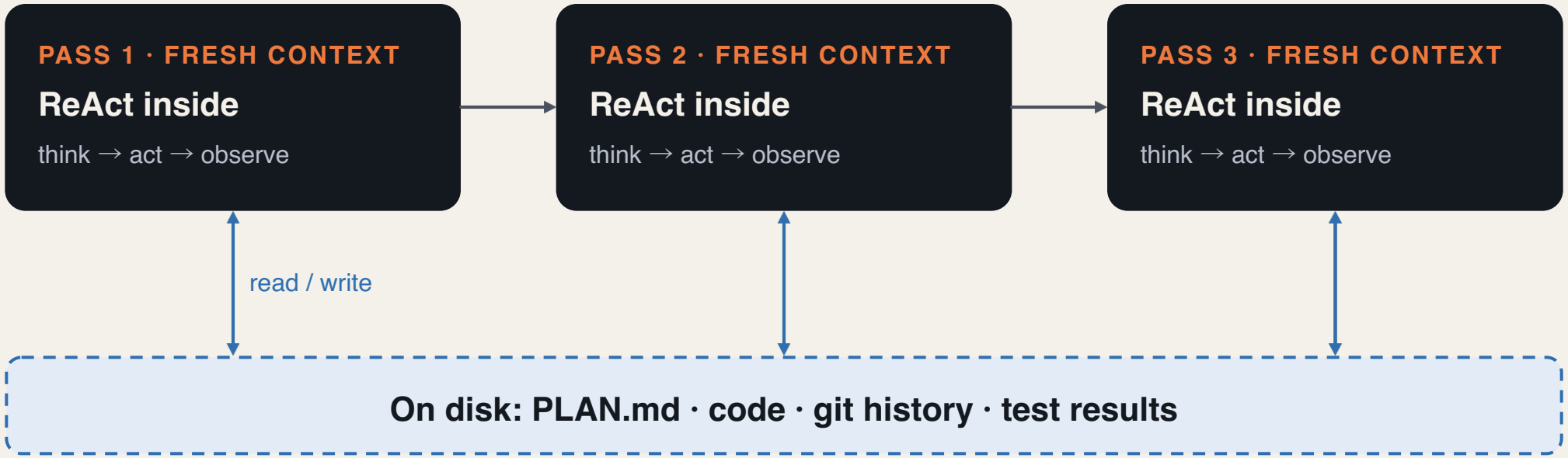
That's the whole loop. Every pass is a fresh session.

File	Job
PROMPT.md	Same instructions, every pass
SPEC.md	What to build
PLAN.md	To-do list + notes: the memory
Tests	Back pressure: "not done yet"
git	History and rollback

Geoffrey Huntley, "Ralph Wiggum as a 'software engineer'" (2025)

INNER AND OUTER

Two loops, two jobs



When Ralph fits, and when it doesn’t

Use it when

- The job is long and splits into small steps
- Tests or builds can verify progress
- State can live in files

Skip it when

- The task is quick or interactive
- Nothing can reliably check the work
- You can’t cap what each pass costs

Guardrails: tests it can’t fake · max passes · budget cap · clear exit condition

LIVE DEMO

Ralph builds a Roman numeral converter

PASS 1

to_roman: additive

I · VIII · MD

PASS 2

to_roman: subtractive

IV · XC · MCMXCIV

PASS 3

from_roman

XIV → 14

PASS 4

Validation

0 · 4000 · "IIII"

Watch the scoreboard go from red to green, one task per pass.

Inner loop: the agent runs its task's tests. Outer loop: the script runs the full suite and doesn't take the agent's word for it.

PROTOCOLS & STANDARDS

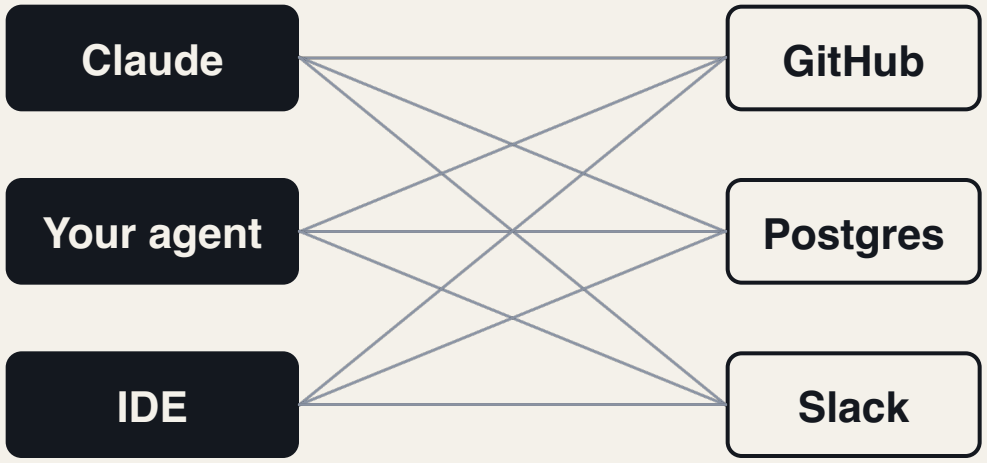
Connecting the pieces

MCP connects agents to tools. OKF packages what they know.

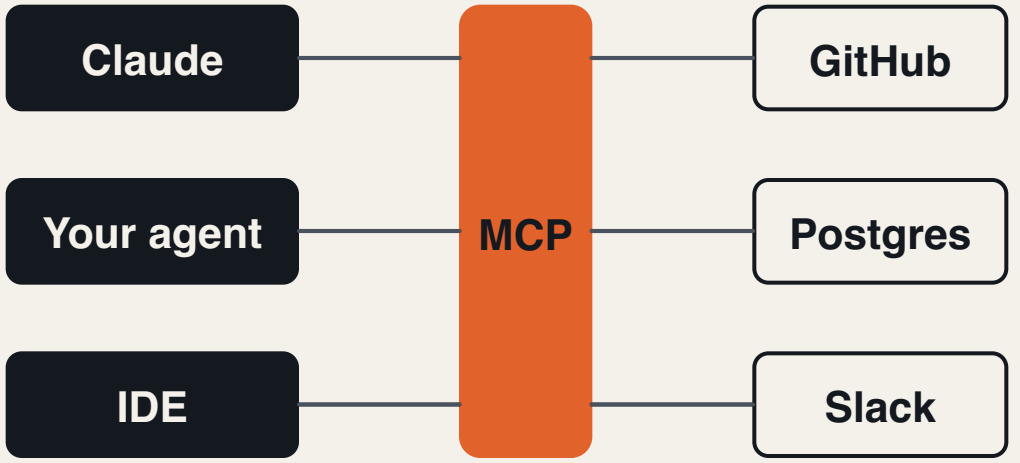
PROTOCOL 1 · MODEL CONTEXT PROTOCOL

The N × M problem

Before: N × M custom integrations

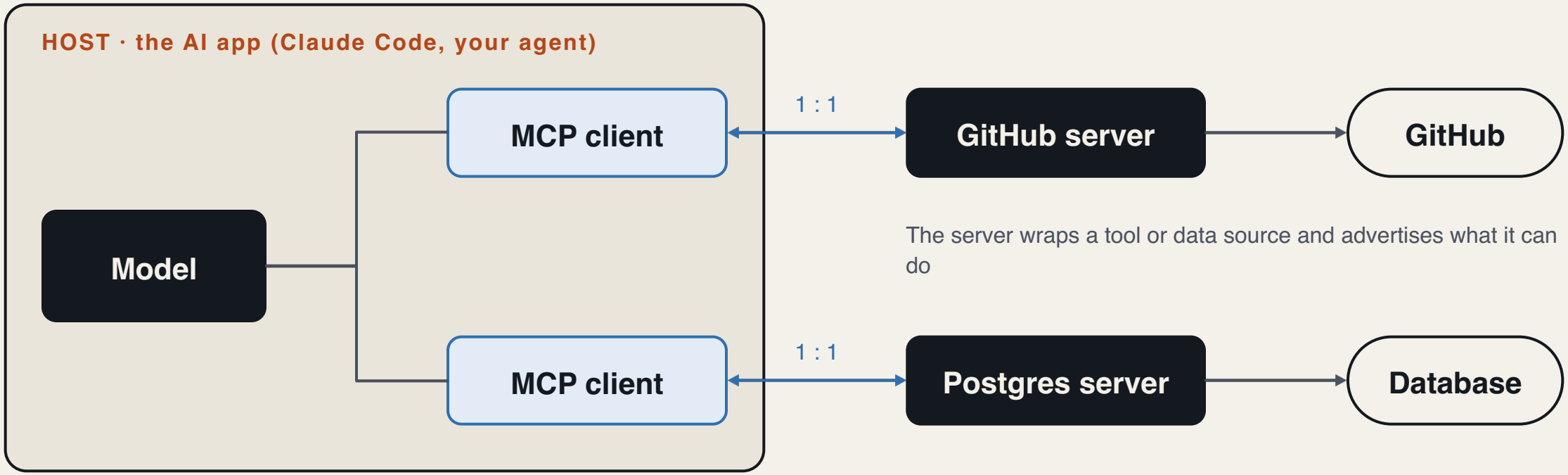


With MCP: N + M



MCP · THE THREE ROLES

Host, client, server



The server wraps a tool or data source and advertises what it can do

MCP · UNDER THE HOOD

How a tool call works

initialize	Client and server agree on capabilities
tools/list	Each tool’s name, description and JSON Schema go into the model’s context
tools/call	The model picks a tool; the client sends the call
result	Back into context: the observation in the ReAct loop

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": { "city": "Raleigh" }
  }
}
```

JSON-RPC 2.0 over stdio (local) or Streamable HTTP (remote). Servers also offer resources and prompts.

An MCP server in Python

```python
from mcp.server.fastmcp import FastMCP
from roman import to_roman

mcp = FastMCP("roman")

@mcp.tool()
def convert_to_roman(number: int) -> str:
    """Convert an integer from 1 to 3999 to a Roman numeral."""
    return to_roman(number)

mcp.run()  # stdio by default
```

Function name, docstring and type hints become the tool's name, description and JSON Schema.

MCP · PITFALLS

What goes wrong



Bad calls

Wrong arguments or made-up IDs. Malformed requests get a JSON-RPC error; a failed tool returns **isError**, which the model reads and retries.



Context bloat

Every tool description costs tokens. With many servers, load tools on demand with tool search.



Trust

Never trust model input. Validate on the server, especially writes and deletes, and authenticate remote servers.

STANDARD 2 · EMERGING

OKF: Open Knowledge Format

- From Google Cloud: v0.1 in June 2026, v0.2 in July
- Knowledge as a folder of Markdown files with YAML frontmatter
- Written by people or agents, portable across tools
- For agents: long-lived memory like decisions, conventions, gotchas

```
---
type: decision
title: Agent runs publish events
tags: [architecture, agents]
---
Each run publishes "run completed".
The evaluator consumes it and
decides whether to start the next pass.
```

Early days: tooling is still young. Try it on a side project before betting a real one on it.

CONCLUSION

Three takeaways

1 **Start with workflows.** Reach for an agent when you can't predict the path.

2 **Patterns compose.** Ralph is a workflow wrapped around an agent running ReAct.

3 **Protocols decouple.** Build a tool once with MCP; keep knowledge portable with OKF.

The model is the engine. The harness is the architecture.

ALL THINGS OPEN 2026

Thank you

Questions?



Arturo Parra · programandojava.com · [LinkedIn]