Agents reporting for duty!

An (in)complete guide to LLM agents and their limits

Tommaso Radicioni

Pycon 2026

The paradigm shift

LLM as a Chatbot 2023–24

Q&A system with strong linguistic fluency

Stateless and reactive - it answers, but cannot act

$\Rightarrow$

★ LLM as an Agent 2025+

Multi-step reasoning with access to tools and the external world

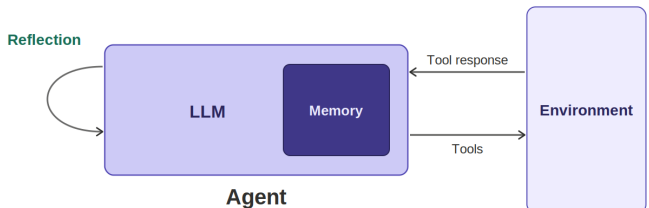
Stateful and proactive - It plans, acts, and iterates

*"The chatbot answers your question. The agent **gets things done**."*

How does an LLM agent work?

An operational definition

System with complex reasoning capabilities, memory, and the means to execute tasks



- **LLM** is the **cognitive core** of the agent.
- **Reasoning** is the mechanism for solving **complex problems**.
- **Memory** maintains **continuity** across interactions and actions.
- **Tools** extend agents' capabilities **beyond language**.

In the next slides. . .

➥ Patterns (increasing autonomy)

- ❖ **Reflection** - The simplest loop
- ♣ **ReAct** - Reasoning + Acting
- ✻ **Multi-Agent** - Orchestrating agents at work

Three patterns of agentic LLM - then what it is needed to deploy agents into production.
90% of the complexity lies in the second half

The experimental setup

Local inference stack

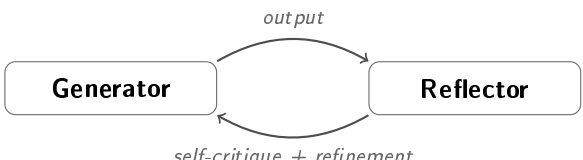
- **Model:** Qwen2.5-7B-Instruct-AWQ
- **Server:** vLLM served via Docker
- **Context window:** 8 192 tokens
- **GPU:** NVIDIA GeForce RTX 3070 Mobile

Why this stack? Small, fully local, reproducible - no API keys, no cloud costs.

Three showcase examples:

1. **Code review agent** - Reflection agent iteratively improving a code snippet.
2. **GitHub assistant** - ReAct agent listing open issues and posting comments.
3. **Project manager agent** - Multi-agent orchestrator building a full FastAPI CRUD service, complete with tests and documentation.

Reflection - The simplest loop



N iterations until convergence

Use case

Any task requiring iterative refinement

Reflection pattern iteratively self-correct and refine outputs:

- Step 1: **Generator** produces an output
- Step 2: **Reflector** critiques and suggests improvements
- Step 3: Iterates until convergence (or budget exhausted)

Reflection - Core loop in 10 lines

```
def run(task):
 response = llm(generate_prompt(task))
 for step in range(MAX_STEPS):
  critique = llm(critique_prompt(task, response))
  if critique.startswith("NO_IMPROVEMENT_NEEDED"):
   return response
  response = llm(refine_prompt(task, response,
      critique))
 return response
```

- 3 prompt roles (`generate`, `critique`, `refine`)
- Loop: evaluate $\rightarrow$ refine $\rightarrow$ repeat
- Stop: sentinel phrase or *MAX_STEPS* reached

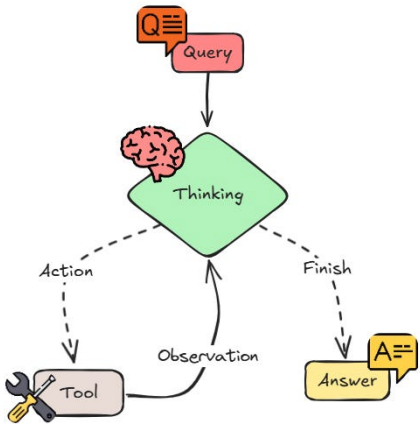
+ Pros

High-quality, accurate, or nuanced outputs thanks to a simple feedback loop.

- Cons

At least 2-3$\times$ LLM calls per iteration: higher cost and latency.

ReAct: Reasoning + Acting



Use case

Any task requiring interaction with external tools or real-world state

The agent iterates over a three-step loop until the task is resolved:

- **Thought** - The agent reasons over the current context
- **Action** - The agent selects and invokes a tool from the registry
- **Observation** - The tool result is appended to the context of the next reasoning step

Snippet - The loop ReAct from scratch

```
tools = [LIST_ISSUES_TOOL, POST_COMMENT_TOOL]

def run(task, max_steps=10):
 messages = [{"role": "user", "content": task}]
 for step in range(max_steps):
  resp = client.chat.completions.create(model=
      model, messages=[SYS] + messages, tools=
      tools)
  choice = resp.choices[0]
  if choice.finish_reason == "tool_calls":
   thought = choice.message.content or ""
   tool_call = choice.message.tool_calls[0]
   args = json.loads(tool_call.function.arguments)
   result = dispatch(tool_call.function.name, args
       )
   messages += [choice.message, {"role": "tool", "
       content": result}]
  else:
   return choice.message.content  # final answer
 return "Max iterations reached."
```

- 2 tools: `list_issues`, `post_comment`
- Reflection: model embeds its Chain-of-Thought step alongside each tool call
- Stop: `finish_reason != "tool_calls"` → risposta finale
- Fallback: regex parsing of `<tool_call>...<tool_call>` in plain text output

Demo - GitHub assistant

Prompt

Review the open issues in the repository and leave a comment on each issue.

Expected trajectory (∼3 steps):

1. `list_issues()` - fetch all open issues
2. *model reasons in* `message.content` - prioritises which ones matter
3. `post_comment(N, ...)` - write to GitHub

What to watch:

- Step 2: agent autonomously prioritises issues inside its own message content
- Step 2→3: chains read + reasoning before acting
- Step 3: **key moment** - writes to GitHub

★ The whole point

"Reflection critiques text.
*ReAct **changes the world**.*
That is the difference."

ReAct - Trade-offs

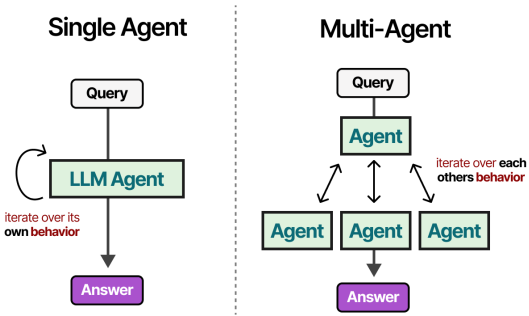
+ Pros

- The agent **acts in the real world** by invoking external tools and services
- Tools can be added dynamically through a registry, without retraining
- Agentic workflows adapt flexibly to user intent at runtime

- Cons

- Fragile output parsing without native function calling support
- More tools $\Rightarrow$ larger prompt context consumed by tool definitions
- **WRITE** actions are potentially harmful: human-in-the-loop oversight is required

Multi-Agent - Orchestrating agents in action



Use case

Any task requiring a simulated debate among domain experts

- **Specialized** agents = unique system prompt defining role, persona, and expertise.
- **Orchestration**: controls turn-taking, context and agents routing.

+ Pros

Decomposes complex tasks into focused sub-problems.

- Cons

Shared conversation history grows fast - context window limits are easily exceeded.

Live demo: A Project Manager and its team

Prompt to the ProjectManager

Build a FastAPI CRUD endpoint. Write full pytest tests and a documentation.

Flow:

1. **ProjectManager** receives the task
2. **DevAgent** writes the FastAPI code
3. **TesterAgent** writes the pytest suite
4. **DocAgent** produces the README
5. **ProjectManager** writes the final answer

```
def dispatch(tool_name, args):
    handlers = {
     "call_dev": self.
         _handle_call_dev,
     "call_tester": self.
         _handle_call_tester,
     "call_doc_expert": self.
         _handle_call_doc
    }
    handler = handlers.get(
        tool_name)
    return handler(args)
```

★ Technical core

Each sub-agent runs its own **internal Chain-of-Thought loop**: it calls the `reflect` tool N times before producing its final answer.

SLM in agentic patterns

Belcak et al., NVIDIA (2025)

"SLMs are sufficiently powerful,inherently more suitable, and necessarily more economical for agentic systems."

- **60-70%** of LLM calls replaceable by SLMs
- **10-30**$\times$ cheaper inference (7B vs. 70-175B)
- **Few hours** to fine-tune a specialist SLM

♣ Why do we mix different models?

- **Privacy** - sensitive data stays on-premise
- **Sustainability** - less CO_2/token at scale
- **Latency** - critical with 5+ concurrent agents

From your laptop to production

1

Tool reliability

Model Context Protocol

2

Evals

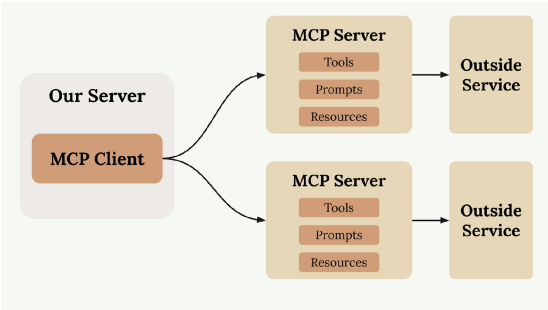
Agentic metrics

3

Observability

Structured tracing

Tool reliability - MCP



- **MCP** = Model Context Protocol (open standard by Anthropic)
- Disentagle tools from agents: versioned, reusable, and independently deployable services
- Declared JSON schema ⇒ eliminates fragile string parsing
- Rapidly growing adoption: GitHub, Slack, DB,...

➥ MCP is the USB for agents

Agent acts as the **MCP client**: it discovers available tools via a declared schema, invokes them with typed arguments, and receives structured responses. All through a standardized protocol.

Evals - Agentic metrics

Metric	What it measures	Example
Tool Correctness	Right tools, right order?	Expected vs. actual tool path
Arg. Correctness	Right arguments per tool call?	Wrong `issue_id` $\Rightarrow$ fail
Trajectory	Redundant steps? Loops?	Step count vs. minimum path

Frameworks & methods

- **deepeval** - open-source LLMs and agents evaluation framework (also: RAGAS, Braintrust, Galileo...)
- **Offline and online eval**: CI/CD gate with eval dataset and live production traces

△ Key challenges

- Unit tests don't work: agents are *probabilistic*
- Outcome $\neq$ Process: a correct answer via a broken path is still broken
- Calibrate the judge: human-labeled examples before trusting scores

Observability: Structured tracing

Traditional log (flat list)

```
[12:01] Thought: ...
[12:01] Action: list_tables
[12:02] Observation: [...]
[12:02] Thought: ...
[12:03] Action: run_sql
[12:03] Observation: [...]
[12:04] Final Answer
```

× Can't see WHERE time went

× Can't pinpoint what failed

Structured trace (tree of spans)

```
v agent_run              (4.2s total)
|-- v llm_call           (0.8s)
|    +-- thought
|-- v tool_call: list_tables   (0.1s)
|    +-- observation
|-- v llm_call           (1.1s)
|-- v tool_call: run_sql       (0.4s)
+-- v llm_call           (1.8s)
     +-- final_answer
```

✓ LLM vs tool time split is instant

✓ One broken span = root cause found

Key metrics to monitor

- **Tool success rate** - most sensitive signal
- **Steps/task** - spikes signal loops
- **Tokens/task** - direct cost proxy
- **Step budget exceeded rate** - lost agents

Standard: **OpenTelemetry** GenAI SemConv

Minimal stack

Agent → OTel SDK → Collector → Backend

Open-source: Langfuse, Arize Phoenix
ML platform: Weights & Biases Weave

Key takeaways

The three patterns

- **Reflection** - self-critique loop.
  Start here. It costs 2-3× more inference, but delivers outsized quality gains.
- **ReAct** - reasoning + tool use.
  The agent stops talking and starts *doing*. Always gate WRITE actions behind human-in-the-loop.
- **Multi-Agent** - specialized roles, orchestrated.
  Reach for it only when the task genuinely requires multiple independent viewpoints.

The production gap

- **MCP** turns ad-hoc tools into a pluggable, versioned ecosystem
- **Evals** replace unit tests with probabilistic, LLM-graded quality metrics
- **Tracing** is the difference between systematic debugging and guessing in the dark

. . . and yes, this guide is deliberately incomplete - agents are a moving target.
The patterns are stable; treat everything else as a working assumption until verified.

Thanks for the attention!

- github.com/tradicio/ai_agents_reporting

✉ tommaso.radicioni@vargroup.com