Assignment 12 — Solutions

Agents: Tools, LangGraph, CrewAI, MCP

Part C — Building GenAI Applications · 26 questions with answers

Q1. What is an AI agent? How is it different from a chain? ★ · Asked: EPAM, PwC, Google

Answer

An **AI agent** is a system where an **LLM decides the control flow**: given a goal, it reasons about what to do, **chooses tools/actions**, observes the results, and loops until the task is done.

- **Chain** – a **fixed, pre-defined sequence of steps** written by the developer (prompt → LLM → parser, or retrieve → generate). Predictable, cheap, easy to test.
- **Agent** – the **sequence is decided dynamically by the LLM** at runtime (which tool, how many steps, when to stop). Flexible for open-ended tasks, but less predictable, slower and costlier.

Rule of thumb: use a chain/workflow when the steps are known; use an agent only when the path genuinely depends on the input.

Q2. What are tools and how do they improve an LLM's capability? How does tool/function calling work? ★ · Asked: TCS, Google

Answer

Tools are functions that an LLM can call to interact with the outside world – search the web, query a database, call an API, run code, do maths, send email. Each tool has a **name, description and input schema**.

How they improve capability: they give the LLM **up-to-date information**, **exact computation**, access to **private data**, and the ability to **take actions** – overcoming knowledge cut-off, poor arithmetic and hallucination.

How tool/function calling works:

1. The app sends the user message **plus tool definitions (JSON schemas)** to the model.
2. The model decides a tool is needed and returns a **structured tool call** (tool name + JSON arguments) instead of plain text.
3. **The application (not the model) executes the function** and sends the result back as a ToolMessage.
4. The model uses the result to answer, or calls more tools.

```
from langchain_core.tools import tool
@tool
def get_weather(city: str) -> str:
    '''Get current weather for a city.'''
    return f"32°C and sunny in {city}"
llm_with_tools = llm.bind_tools([get_weather])
msg = llm_with_tools.invoke("Weather in Pune?")
print(msg.tool_calls)   # [{'name': 'get_weather', 'args': {'city': 'Pune'}, ...}]
```

Q3. How does a ReAct agent work (Thought → Action → Observation loop)? ★ ·

Asked: Google

Answer

A **ReAct agent** runs a loop of **Thought → Action → Observation**:

1. **Thought** – the LLM reasons about what it needs next.
2. **Action** – it selects a tool and its input.
3. **Observation** – the tool runs and its output is added to the context.
4. Repeat until the LLM has enough information → **Final Answer**.

Example – "What is the GDP of India divided by its population?"

Thought: need GDP → Action: search("India GDP") → Observation: ~$3.9 trillion → Thought: need population → Action: search(...) → Observation: ~1.44 billion → Action: calculator(3.9e12/1.44e9) → Observation: 2708 → Final Answer: about $2,700 per person.

Modern implementations use native tool calling instead of parsing "Action:" text; in LangGraph: `create_react_agent(llm, tools)` (LangChain 1.x also offers the equivalent `from langchain.agents import create_agent`). Set a **max iterations** limit.

Q4. Give an LLM the capability to use a calculator. [CODE]

Answer

```
from langchain_core.tools import tool
from langgraph.prebuilt import create_react_agent
from langchain_openai import ChatOpenAI
import numexpr

@tool
def calculator(expression: str) -> str:
    '''Evaluate a math expression, e.g. "2**10 / 7". Use for ANY arithmetic.'''
    return str(numexpr.evaluate(expression).item())     # safe: no arbitrary Python

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
agent = create_react_agent(llm, tools=[calculator])
result = agent.invoke({"messages": [("user", "What is 18.5% of 72,450 plus 3^7?")]})
print(result["messages"][-1].content)
```

(Legacy alternative: `load_tools(["llm-math"], llm=llm)` with `initialize_agent`.) Never use Python `eval()` on model output.

Q5. Give an LLM the capability to use web search. [CODE]

Answer

```
# pip install langchain-community langchain-tavily google-search-results
from langgraph.prebuilt import create_react_agent

# Option 1: Tavily (search API designed for LLM agents)
from langchain_tavily import TavilySearch
search = TavilySearch(max_results=3)                    # needs TAVILY_API_KEY

# Option 2: Google Search via SerpAPI
from langchain_community.utilities import SerpAPIWrapper
from langchain_core.tools import Tool
serp = SerpAPIWrapper()                                 # needs SERPAPI_API_KEY
google_tool = Tool(name="google_search", func=serp.run,
                   description="Search Google for current events and facts.")

# Option 3 (no key): from langchain_community.tools import DuckDuckGoSearchRun

agent = create_react_agent(llm, tools=[search, calculator])
out = agent.invoke({"messages": [("user", "Who won the latest IPL final, and by how many runs?")]})
print(out["messages"][-1].content)
```

(Also available: GoogleSearchAPIWrapper using Google Custom Search API.)

Q6. What is a Python REPL tool? What are the security risks of letting an LLM execute code?

Answer

REPL = Read–Eval–Print Loop – an interactive environment that reads code, executes it and returns the output (like the Python shell). A **Python REPL tool** lets the LLM write Python code and run it, then read the result – useful for maths, data analysis (pandas), plotting and file processing.

```
# pip install langchain-experimental
from langchain_experimental.tools import PythonREPLTool
agent = create_react_agent(llm, tools=[PythonREPLTool()])
```

Security risks: arbitrary code execution – the model (or a prompt-injected document) could delete files, read secrets/environment variables, exfiltrate data over the network, install malware or consume resources.

Mitigations: run code in an **isolated sandbox** (Docker container, E2B, gVisor) with no secrets, no or restricted network, read-only file system, CPU/memory/time limits; human approval for risky actions; never run it on production servers.

Q7. Create one custom tool and one custom agent. [CODE]

Answer

```
from langchain_core.tools import tool
from pydantic import BaseModel, Field
from langgraph.prebuilt import create_react_agent

ORDERS = {"A101": "Shipped - arriving 14 Oct", "A102": "Processing"}

class OrderInput(BaseModel):
    order_id: str = Field(description="Order ID like A101")

@tool(args_schema=OrderInput)
def get_order_status(order_id: str) -> str:
    '''Look up the delivery status of a customer order.'''
    return ORDERS.get(order_id.upper(), "Order not found")

@tool
def refund_policy() -> str:
    '''Return the company's refund policy.'''
    return "Refunds within 7 days of delivery for unused items."

system = "You are a polite e-commerce support agent. Use tools; never invent order data."
agent = create_react_agent(llm, tools=[get_order_status, refund_policy], prompt=system)

res = agent.invoke({"messages": [("user", "Where is my order a101 and can I return it?")]})
print(res["messages"][-1].content)
```

A fully custom agent can also be built as a LangGraph graph (LLM node + ToolNode + conditional edge) – see the LangGraph answers.

Q8. What is LangGraph? Why use it instead of a simple agent loop? ★ · Asked: Google

Answer

LangGraph is a library (from LangChain) for building **stateful, controllable agents and multi-step workflows as graphs**: nodes are functions/LLM calls, edges define the flow, and a shared **state** is passed between them.

Why use it instead of a simple agent loop:

- **Explicit control flow** – loops, branches, retries, parallel steps; mix deterministic steps with LLM decisions.
- **Persistence/memory** via checkpointers – resume conversations, long-running tasks, "time travel" debugging.
- **Human-in-the-loop** – pause before sensitive actions for approval/editing.
- **Multi-agent** architectures (supervisor, hierarchical).
- **Streaming** of tokens and intermediate steps; fault tolerance.
- Production deployment and observability (LangSmith, LangGraph Platform).

Simple loops break down when you need reliability, state and control; LangGraph makes them explicit.

Q9. What is State in LangGraph? How are nodes and edges added? (add_node, add_edge) · Asked: EPAM, Google

Answer

State is the shared data structure (usually a `TypedDict` or Pydantic model) that flows through the graph. Every node **receives the current state and returns updates** to it. **Reducers** define how updates are merged – e.g. `Annotated[list, add_messages]` **appends** messages instead of overwriting.

```
from typing import TypedDict, Annotated
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages

class State(TypedDict):
    messages: Annotated[list, add_messages]
    category: str

def classify(state: State):
    return {"category": "billing"}           # partial state update

def respond(state: State):
    return {"messages": [("ai", f"Routing you to {state['category']}")]}

graph = StateGraph(State)
graph.add_node("classify", classify)          # add_node(name, function)
graph.add_node("respond", respond)
graph.add_edge(START, "classify")             # add_edge(from, to)
graph.add_edge("classify", "respond")
graph.add_edge("respond", END)
app = graph.compile()
print(app.invoke({"messages": [("user", "I was charged twice")]}))
```

Q10. How do you add a conditional edge? What are START and END, set_entry_point and set_finish_point?

Answer

Conditional edge – after a node runs, a **routing function** reads the state and returns the name of the next node:

```
def route(state: State) -> str:
    return "billing_node" if state["category"] == "billing" else "tech_node"

graph.add_conditional_edges("classify", route,
                            {"billing_node": "billing_node", "tech_node": "tech_node"})
# Typical agent loop: from langgraph.prebuilt import tools_condition
# graph.add_conditional_edges("llm", tools_condition)   # -> "tools" or END
```

- **START** – a special virtual node representing the **entry** of the graph; `add_edge(START, "x")` makes x the first node.
- **END** – a special node meaning **the graph finishes**; `add_edge("x", END)`.
- **set_entry_point("x")** – older equivalent of `add_edge(START, "x")`.
- **set_finish_point("x")** – older equivalent of `add_edge("x", END)` – marks x as a node after which the graph stops.

Q11. How does LangGraph support memory/checkpointing and human-in-the-loop? · Asked: Google

Answer

Memory / checkpointing: compile the graph with a **checkpointer**; after every step LangGraph saves the full state for a `thread_id`. Calling again with the same thread_id continues the conversation; you can inspect history and "time travel" to earlier states.

```
from langgraph.checkpoint.memory import MemorySaver       # SqliteSaver / PostgresSaver in prod
app = graph.compile(checkpointer=MemorySaver())
cfg = {"configurable": {"thread_id": "customer-42"}}
app.invoke({"messages": [("user", "Hi")]}, cfg)
```

Human-in-the-loop: pause execution so a human can approve, edit or reject:

```
app = graph.compile(checkpointer=MemorySaver(), interrupt_before=["issue_refund"])
app.invoke(inputs, cfg)               # stops before the refund node
print(app.get_state(cfg).values)      # human reviews the state
app.update_state(cfg, {"approved": True})
app.invoke(None, cfg)                 # resume
```

(Newer API: call `interrupt(value)` inside a node and resume with `Command(resume=...)`.) The **Store** API adds long-term memory shared across threads.

Q12. What is CrewAI? How do you create agents, tasks and run a crew? ★

Answer

CrewAI is a Python framework for building **multi-agent systems** where several **role-playing agents** (each with a role, goal and backstory) collaborate on **tasks** as a **crew**, like a team of employees.

```
# pip install crewai crewai-tools
from crewai import Agent, Task, Crew, Process
from crewai_tools import SerperDevTool

researcher = Agent(role="Senior Research Analyst",
                   goal="Find the latest developments in {topic}",
                   backstory="Expert at finding and verifying information.",
                   tools=[SerperDevTool()], llm="gpt-4o-mini", verbose=True)
writer = Agent(role="Tech Content Writer",
               goal="Write an engaging blog post on {topic}",
```

```
                backstory="Turns complex research into simple articles.", llm="gpt-4o-mini")

research_task = Task(description="Research the top 5 trends in {topic}.",
                     expected_output="Bullet list of 5 trends with sources.", agent=researcher)
write_task = Task(description="Write a 400-word blog post using the research.",
                  expected_output="Markdown blog post.", agent=writer,
                  context=[research_task], output_file="blog.md")

crew = Crew(agents=[researcher, writer], tasks=[research_task, write_task],
            process=Process.sequential, verbose=True)     # or Process.hierarchical
result = crew.kickoff(inputs={"topic": "Agentic AI"})
print(result)
```

Agent = role, goal, backstory, tools, llm; **Task** = description, expected_output, agent, context; **Crew.kickoff()** runs it.

Q13. Single-agent vs multi-agent systems — when is multi-agent worth the complexity? · Asked: Google

Answer

Single agent – one LLM with a set of tools. Simpler, cheaper, faster, easier to debug. Good for most use cases (support bot, RAG assistant).

Multi-agent – several specialized agents (researcher, coder, reviewer) coordinated by a supervisor or a workflow.

Worth the complexity when:

- The task has **clearly separable sub-tasks** needing different expertise, prompts or tools.
- There are **too many tools** for one agent to choose from reliably (> ~10–15).
- You need **parallel work** or a **reviewer/critic** step that checks another agent's output.
- Different sub-tasks need different models (cheap vs strong) or permissions.

Costs: more LLM calls (cost/latency), coordination failures, harder debugging and evaluation. Start with one agent and split only when you measure a benefit.

Q14. What is MCP (Model Context Protocol) and what problem does it solve? ★ · Asked: TCS, EPAM, Infosys, Google

Answer

MCP (Model Context Protocol) is an **open standard** (introduced by Anthropic in late 2024, now widely adopted by OpenAI, Google, Microsoft and IDEs) for connecting AI applications to external **tools, data sources and prompts** in a uniform way – often described as "**USB-C for AI applications**".

Problem it solves: without it, every AI app had to build custom integrations for every tool (M apps × N tools = M×N connectors). With MCP, a tool provider builds one **MCP server** (e.g. for GitHub, Slack, a database, Google Drive), and any **MCP client/host** (Claude, ChatGPT, Cursor, VS Code, your own LangGraph agent) can use it → M + N.

Architecture: host app → MCP client → MCP server, communicating with JSON-RPC over stdio (local) or HTTP (remote). Servers expose **tools** (actions), **resources** (data/files) and **prompts** (templates).

Security: treat third-party servers as untrusted code; review permissions; beware tool poisoning and prompt injection.

Q15. How do you evaluate and debug agents (traces, LangSmith/Langfuse, step-level metrics)? · Asked: Google

Answer

- **Tracing/observability:** log every step – LLM inputs/outputs, tool calls and arguments, tool results, latency, tokens and cost – with **LangSmith, Langfuse, Arize Phoenix**, or OpenTelemetry. Inspect traces of failed runs.
- **End-to-end metrics:** task success rate / goal completion, final-answer correctness (vs reference or LLM-as-judge), user feedback.
- **Step-level (trajectory) metrics:** was the correct tool chosen, were arguments valid, number of steps vs optimal, redundant calls, error/retry rate, did it stop correctly.
- **Efficiency:** latency, cost per task, tokens.
- **Safety:** policy violations, unauthorized actions, prompt-injection resistance (red teaming).
- Build a **test suite of scenarios** (including edge cases) and run it on every prompt/model/tool change (regression testing); unit-test tools separately; use deterministic mocks for external APIs.

Q16. What are common agent failure modes (infinite loops, wrong tool choice, cost blow-ups) and how do you guard against them? · Asked: TCS, Google

Answer

Common failure modes:

- **Infinite loops / repeated tool calls** – the agent keeps searching or retrying.
- **Wrong tool choice** or wrong/hallucinated arguments (invented IDs).
- **Cost and latency blow-ups** – too many steps or huge contexts.
- **Hallucinated tool results** or claiming an action was done when it wasn't.
- **Prompt injection** through tool outputs; unsafe or irreversible actions.
- Losing track of the goal in long tasks; context overflow.

Guards:

- **Max iterations / recursion limit** (LangGraph `recursion_limit`), timeouts, and per-task **token/cost budgets**.
- Clear tool names, descriptions and **Pydantic schemas**; validate arguments; fewer, well-designed tools.
- Error handling in tools that returns helpful messages instead of crashing.
- **Human approval** for high-impact actions; least-privilege permissions; sandboxing.
- Deterministic workflow steps where possible; caching repeated calls.
- Monitoring, alerts and evaluation suites.

Q17. What is Agentic AI, and how is it different from standard Generative AI or a RAG chatbot? ★ · Asked: TCS, Infosys, PwC

Answer

- **Generative AI (standard):** the model takes an input and **produces content** in one shot – answer, summary, code. **RAG** adds a fixed retrieve-then-generate step. The flow is defined by the developer and the model does not act on the world.
- **Agentic AI:** the LLM works toward a **goal autonomously over multiple steps** – it **plans**, decides **which tools/APIs to call**, observes results, keeps **state/memory**, can correct itself, and **takes actions** (create ticket, query DB, send email), possibly coordinating with other agents.

Example: a RAG bot answers "What is the refund policy?". An agent handles "Refund my last order" – looks up orders, checks eligibility, calls the refund API, asks for confirmation and updates the CRM.

Trade-offs: agents are more capable but less predictable, slower, costlier, and need guardrails, permissions and human approval for risky actions.

Q18. Who decides which tool to call? What if the model selects the wrong tool? Why shouldn't an agent have access to every tool? ★ · Asked: Infosys, Google

Answer

Who decides: the **LLM** decides, based on the user request, the conversation and the **tool names, descriptions and argument schemas**; the **application** executes the call and can block it.

If it picks the wrong tool:

- Improve **tool descriptions** (when to use / when NOT to use), names and schemas; add few-shot examples.
- **Reduce and group tools** – fewer, distinct tools; route to a sub-agent with a small toolset.
- Validate arguments; return clear error messages so the agent can recover.
- Add deterministic routing for known intents; evaluate tool-selection accuracy in tests.

Why not every tool: more tools → more confusion and wrong selections, bigger prompts (cost/latency), and a larger **attack surface** – a prompt injection could trigger dangerous tools. Follow **least privilege**: give each agent only the tools and permissions it needs for its task.

Q19. Your agent has 10 tools. The user asks a simple question, but the agent calls 5 different tools. How do you fix this? ★ · Asked: Infosys

Answer

1. **Inspect traces** – which tools, why (its reasoning), and whether descriptions overlap.
2. **Route simple queries first:** a classifier/router answers FAQ-type questions directly or with one retrieval call; only complex requests go to the full agent.
3. **Tighten tool descriptions and system prompt:** "Call a tool only if the answer requires it; prefer answering directly; use the minimum number of tools."
4. **Reduce the toolset** exposed per request (dynamic tool selection based on intent).
5. **Limits:** max tool calls / recursion limit, per-request token budget, caching of repeated tool results.
6. Use a **cheaper/faster model** for routing and a stronger one only when needed.
7. Add **tests and metrics** for tool-call count and unnecessary calls; regression-test after changes.

Q20. Would you let an LLM perform a sensitive action automatically, like deleting a customer record or making a financial transaction? Should the LLM decide permissions? ★ · Asked: Infosys, Google

Answer

No – not without controls. The LLM can **propose** the action, but:

- **Permissions are enforced by the application/backend, never by the LLM** – the tool checks the authenticated user's role, scope and limits in code.
- **Human-in-the-loop approval** for destructive, financial or irreversible actions (show a summary, require confirmation/maker-checker approval).

- **Least privilege:** read-only tools by default; separate, narrowly scoped write tools; soft-delete instead of hard delete.
- **Validation and limits:** business rules in code (amount caps, eligibility), idempotency keys, rate limits.
- **Audit logs** of every action; ability to roll back; monitoring and alerts.
- Protection against prompt injection (an email or document must never be able to trigger a transfer).

Q21. What is the difference between normal function calling and MCP? In MCP, what is the difference between tools and resources? ★ · Asked: Infosys, EPAM

Answer

- **Function calling** – a **model/API feature**: you define functions (JSON schemas) inside your application; the model returns a call; **your app executes it**. Tools are hard-wired into one application and each provider has its own format.
- **MCP (Model Context Protocol)** – an **open protocol/standard** for how AI applications **discover and connect** to external capabilities. Tools live in separate **MCP servers** that any MCP-compatible client (Claude, ChatGPT, IDEs, your agent) can connect to. Under the hood the model still uses function calling – MCP standardizes the **integration layer** (discovery, schemas, transport, auth), so one server can be reused by many apps.

MCP primitives:

- **Tools** – actions the **model** decides to invoke, which may have side effects (create ticket, run query).
- **Resources** – read-only **data/context** exposed by the server (files, DB records, documents), usually selected by the application/user and loaded into context.
- **Prompts** – reusable prompt templates exposed by the server.

Q22. How would you run three independent agents in parallel and make sure one agent's failure doesn't break the others? How do you add retries with backoff? ★ · Asked: EPAM

Answer

Parallel execution: run them concurrently with `asyncio.gather` (I/O-bound LLM/API calls) or as parallel branches in **LangGraph** (fan-out from one node to three nodes, then a join node).

Failure isolation:

- `asyncio.gather(..., return_exceptions=True)` or wrap each agent in try/except so one exception doesn't cancel the others.
- **Per-agent timeouts** (`asyncio.wait_for`), separate error states, and partial results – the aggregator handles missing outputs gracefully.
- Circuit breakers for repeatedly failing dependencies; bulkheads (separate worker pools/queues) in production.

Retries: retry only transient errors (timeouts, 429, 5xx) with **exponential backoff + jitter** and a max attempt count (e.g. `tenacity`), and make actions **idempotent**.

```
import asyncio
async def run_agent(name, coro, timeout=30):
    try:
        return name, await asyncio.wait_for(coro, timeout)
    except Exception as e:
        return name, {"error": repr(e)}          # isolate failure

async def main():
    results = await asyncio.gather(
        run_agent("research", research_agent(q)),
```

```
        run_agent("sql", sql_agent(q)),
        run_agent("policy", policy_agent(q)))
    return dict(results)
```

Q23. LangChain vs LangGraph – when would you use each? ★ · Asked: Google

Answer

- **LangChain** – building blocks and integrations (models, prompts, loaders, splitters, vector stores, retrievers, output parsers) and **LCEL** for **linear or simple branching pipelines** (prompt → model → parser; standard RAG chain).
- **LangGraph** – orchestration for **stateful, cyclic, multi-step workflows and agents**: loops, conditional routing, retries, persistence (checkpointing), human-in-the-loop, multi-agent coordination, streaming of intermediate steps.

Rule: a straightforward chain or RAG pipeline → LangChain/LCEL; anything with loops, decisions, tools, long-running state or approvals → LangGraph (which uses LangChain components inside its nodes).

Q24. What is the planner–executor architecture? How does it differ from a supervisor (orchestrator) agent? · Asked: Google

Answer

- **Planner–executor (plan-and-execute):** a **planner** LLM first creates a step-by-step **plan** for the whole task; an **executor** (agent with tools) carries out each step; the planner can **re-plan** based on results. Good for long, multi-step tasks; fewer expensive reasoning calls; more predictable.
- **Supervisor / orchestrator agent:** a central agent receives the request and **delegates sub-tasks to specialized worker agents** (research agent, SQL agent, writer agent), routing dynamically at each step and combining their outputs. Focus is on **delegation and coordination** rather than an upfront plan.

They can be combined: a supervisor whose first step is to create a plan.

Q25. What is agent memory? What are the risks of long-term memory and stale information? · Asked: Google

Answer

- **Short-term memory** – the current conversation/thread state (messages, tool results), kept in the context or a checkpointer.
- **Long-term memory** – facts persisted across sessions (user preferences, past decisions, summaries), stored in a DB/vector store and retrieved when relevant. Types: semantic (facts), episodic (past interactions), procedural (learned instructions).

Risks:

- **Stale or wrong facts** reused confidently (old address, outdated policy).
- **Privacy** – storing personal data; consent and right-to-delete.
- **Memory poisoning** – injected instructions saved as "memories".
- Context bloat and cost.

Mitigations: timestamps and TTL/expiry, re-validate against the source system of record for important facts, store facts not instructions, user-visible and deletable memories, PII policies, scoping memory per user/tenant.

Q26. In a Text-to-SQL agent, how do you prevent the LLM from executing destructive or hallucinated SQL queries? ★ · Asked: Deloitte

Answer

- **Read-only database user** (SELECT-only grants) on a **read replica**; never give DDL/DML rights.
- **SQL validation layer** before execution: parse with `sqlglot`/`sqlparse`; allow only SELECT; block DROP/DELETE/UPDATE/INSERT/ALTER, multiple statements and comments; allow-list tables/columns; enforce LIMIT and query timeouts.
- **Schema grounding:** give the LLM the exact schema, column descriptions and sample values so it doesn't invent tables/columns; validate identifiers against the real schema.
- **Row-level security / views** so users only see permitted data.
- **Dry-run / EXPLAIN** to catch errors and expensive queries; self-correction loop with the DB error message (max 2–3 tries).
- Human approval for any write operation; audit logging of generated SQL.
- Evaluate on a test set of question→SQL pairs (execution accuracy).