

PRACTICAL AI AGENT WORKFLOWS

Sabrina Goldfarb



linkedin.com/in/sabrinagoldfarb



github.com/sgoldfarb2

TODAY'S SCORECARD

- Context – how much information do we need to keep in context? (the less we need, the better)
- Reliability – if we run it 100 times, how frequently does it get the right answer?
- Cost – how much does it cost (money, latency, engineering effort)?

OUR PRODUCT TODAY

- First, what do they want to do as a career?
- Next, what colleges/trade schools/courses do they need?
- Finally, how to pay for it all? (scholarships, loans, etc.)

WHAT IS AN AGENT

REASONING LLMS VS AGENTS

WHAT THE AGENT CAN DO

TOOLS

- Allow LLMs to interact with their environment
- Can call information/external APIs, execute code, call other models
- Can be read only or read/write (with guardrails hopefully)
- Even LLMs not trained to use tools can use them with proper prompting

MCP

- Model Context Protocol
- A standardization for connecting tools and APIs to LLMs
- MCP server provides us with the tools via the MCP standard
- MCP client then communicates with the MCP server to execute the tool
- Information comes back to the MCP client and fed back to our LLM
- MCP standardizes the transport/format, not the tool itself (so it will work with any client ex. Claude, Copilot, etc.)

SKILLS

- Give instructions to your agent
- Skills are loaded in a way that protects context
- You can bundle additional information in scripts
- Anatomy:
 - Folder with required SKILL.md and optional subfolders
- Open standard and used industry wide now
- Complements MCP, doesn't replace it
- Can contain executable code, so only load skills from trusted sources

WHAT THE AGENT KNOWS

RAG

- Retrieval Augmented Generation
- Steps
 - Ingest, Chunk, Embed, Store, Retrieve, Augment
- Searches for relevance (similarity metric)
- Gives LLM/agent “memory” by finding relevant info to add to context without giving everything
- Reduces hallucinations
- RAG adds context at query time, fine-tuning is used for re-training, but these are often used together

CONTEXT

- Context is finite and expensive
- RAG: helps retrieve only what is relevant right now
- Re-rankers: a second pass at what is needed in context
- Context can have short term and long term memory within it

SHORT TERM MEMORY

- LLMs are stateless
- The context window is everything the model “knows” on that call
- Agents keep a running state (goal, conversation, tool results, notes)
- Context is limited in size so the agent has to choose what goes in

LONG TERM MEMORY

- Semantic memory – docs, knowledge bases, product catalogues
 - This is what RAG usually searches
- Episodic memory – past conversations, logs, lessons learned
- Procedural memory – the models trained weights, plus the system prompt, and things that tell the agent how to act

HOW THE AGENT THINKS

PLANNING

- Without planning: agents lose track of goals on multi-step tasks
- Planning: decompose a goal into a sequence of steps
- Two patterns
 - Plan then execute
 - ReAct
- Reasoning models plan better
- Plans are hypotheses and can change with feedback
- Complex goals can be split into many subtasks

REFLECTION

- Agent evaluates its output and revises before finishing
- Self-refine: agent critiques its own output and tries again
- Reflexion: after a failed attempt, agent generates a written lesson learned to try again with
- Reflection is how an agent decides a plan needs updating
- More reflection means better accuracy but more latency and cost

WORKFLOWS WITH LLMS (NOT AGENTS)

HOW MUCH DO I CONTROL?

- Workflow: I write the arrows
- Single Agent: the model picks arrows from a set I defined
- Multi-Agent: the model picks the arrows and who makes the decision next

PROMPT CHAINING

- Output of one call becomes input of the next
- Used when tasks decompose cleanly
- Better than one large prompt because smaller prompts are easier
- A “gate” can sit between steps to validate outputs
- Paying for latency to buy accuracy

LLM Workflow: Prompt Chaining

A workflow, not an agent — each LLM call has a fixed job, and the output of one call becomes the input to the next.

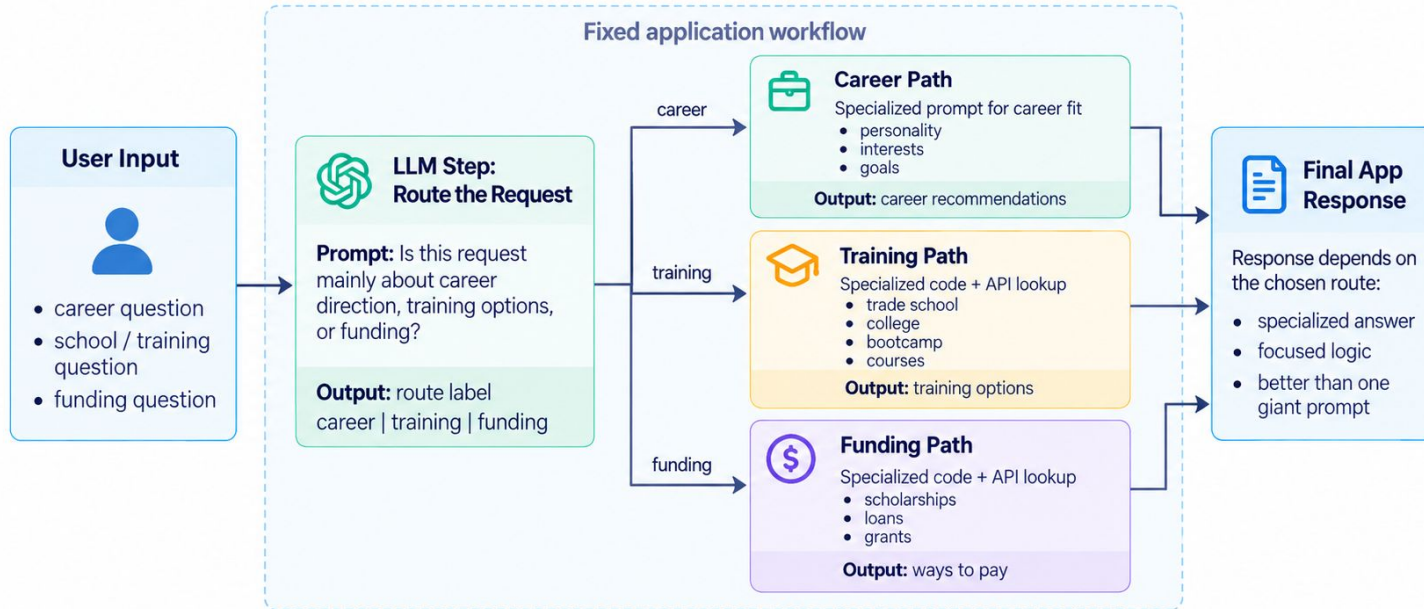


ROUTING

- Classify the input and send it down a path
- Can route to LLMs, different prompts, or deterministic code
- Classification step is usually cheap
- Specialize each path and optimize for cost/latency
- Routing criteria can be:
 - Topic/intent
 - Cost/complexity
 - Confidence (ex. escalate to a human for low confidence)
- Routing is a *fork* not a sequence

LLM-Powered Workflow: Routing (Not an Agent)

Classify the request first, then send it to a specialized fixed path. The LLM only routes; each path is predefined.

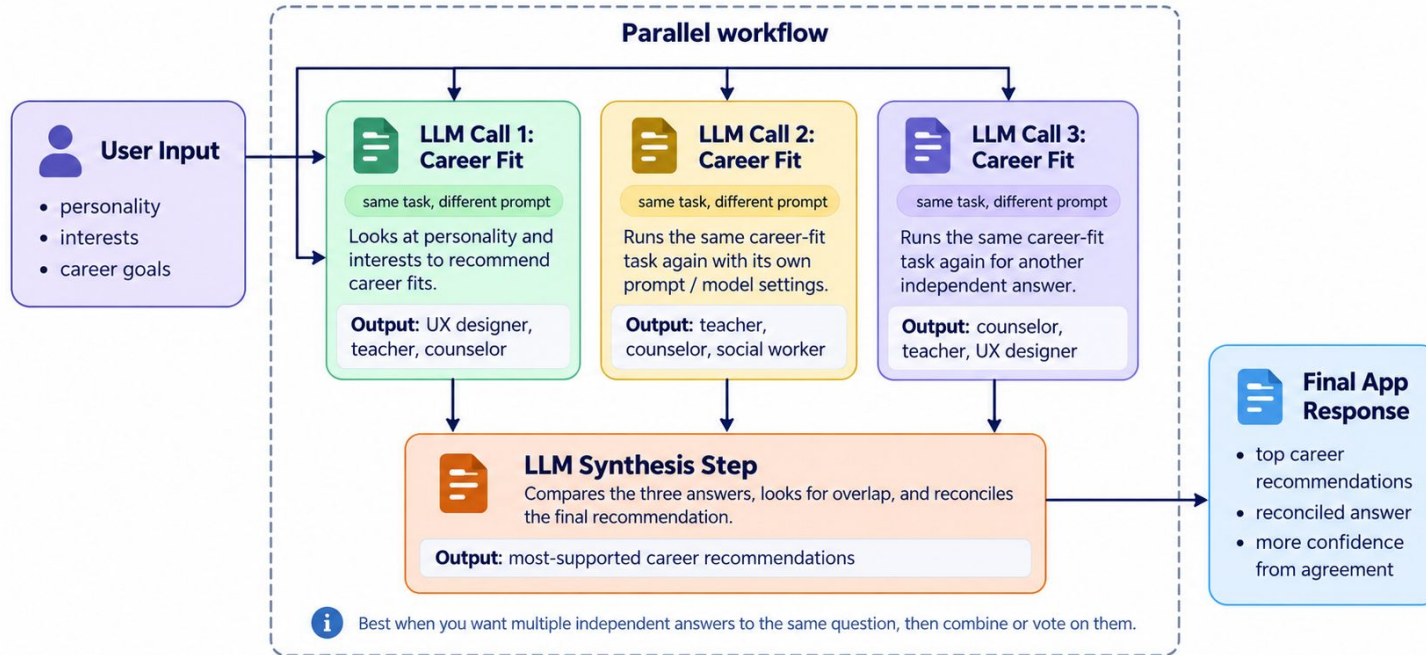


PARALLELIZATION

- Run multiple LLM calls simultaneously instead of one after another
- At the end, results are aggregated
- Sectioning: split a task into independent subtasks that run in parallel
- Voting: run the *same* task multiple times and aggregate outputs
- Faster than prompt chaining and can improve accuracy through redundancy

LLM-Powered Workflow: Parallelization (Not an Agent)

The same request goes to multiple LLM calls at the same time. A final synthesis step compares the outputs and combines them into one response.

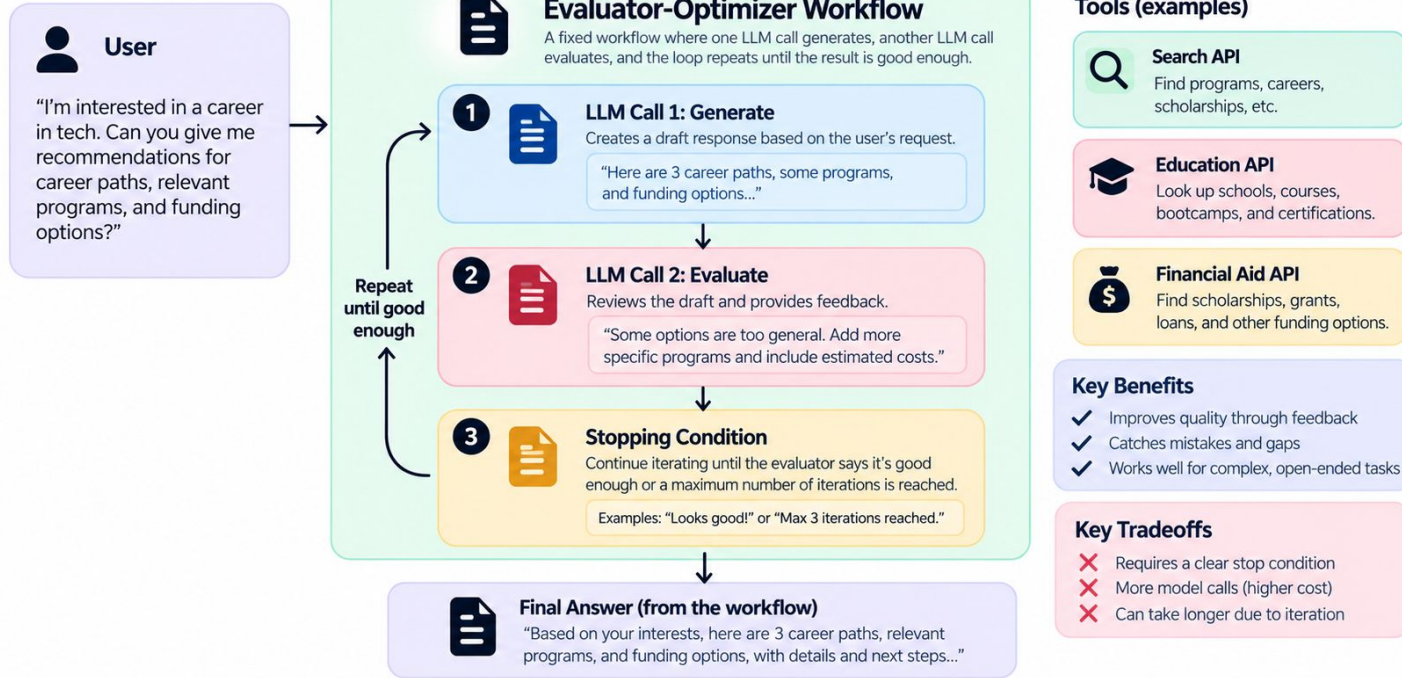


EVALUATOR-OPTIMIZER

- One call generates a response, a second call evaluates it and gives feedback on a loop
- This is reflection but as a workflow
- Best fit: evaluation criteria are clear and easy to capture
- Trade-off: better quality but more calls

LLM-Powered Workflow: Evaluator-Optimizer (Not an Agent)

One LLM call generates a solution, another LLM call evaluates it and gives feedback.
The generator uses that feedback to improve, repeating until the evaluator says it is good enough or a stop condition is reached.

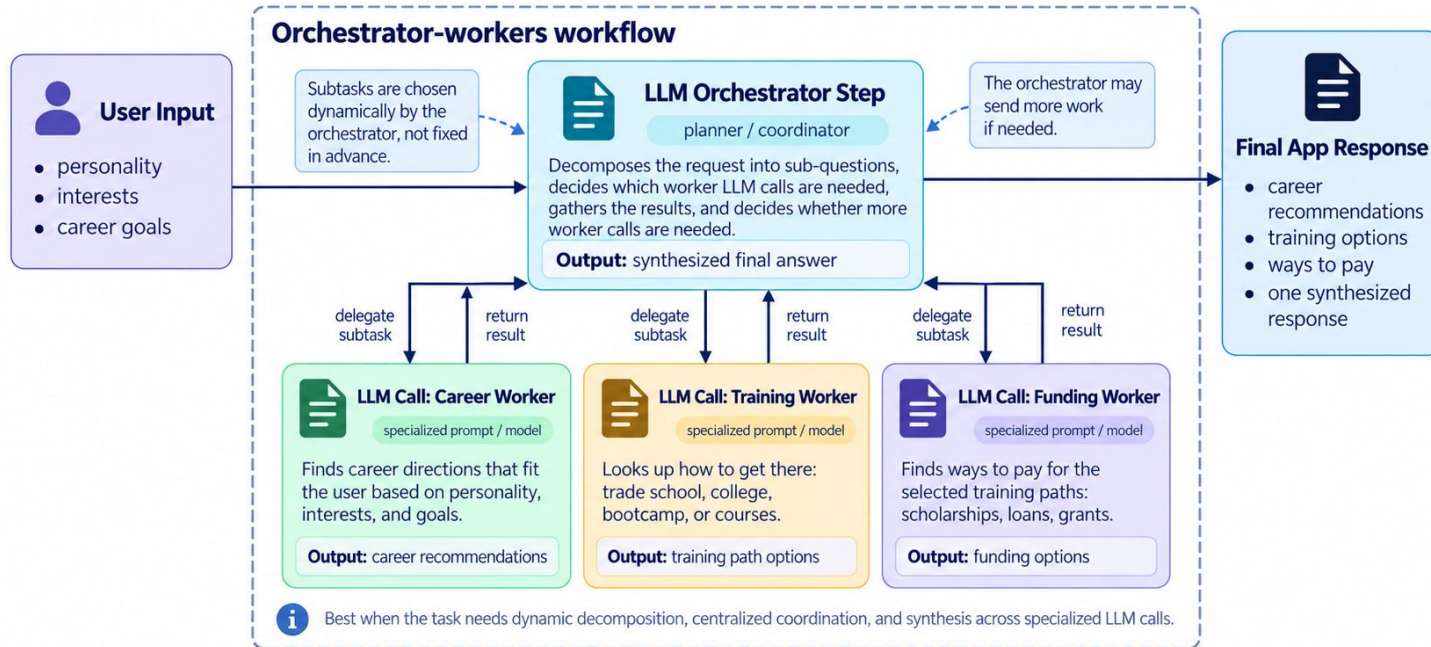


ORCHESTRATOR-WORKERS

- A central LLM call breaks the task into subtasks and delegates to each worker LLM
- Orchestrator synthesizes all the results
- Different from prompt chaining: orchestrator decides the subtasks
- Sits at the edge of workflow/agent
- Fits where you can't predict subtasks in advance (code)

LLM-Powered Workflow: Orchestrator-Workers (Not an Agent)

An orchestrator LLM call breaks the task into sub-questions, sends them to worker LLM calls, and then synthesizes the results into one final response.



WORKFLOWS SCORECARD

- Context:
 - Bounded, each step sees what we give it
- Reliability:
 - I can test any step in isolation
- Cost:
 - I control the amount of calls

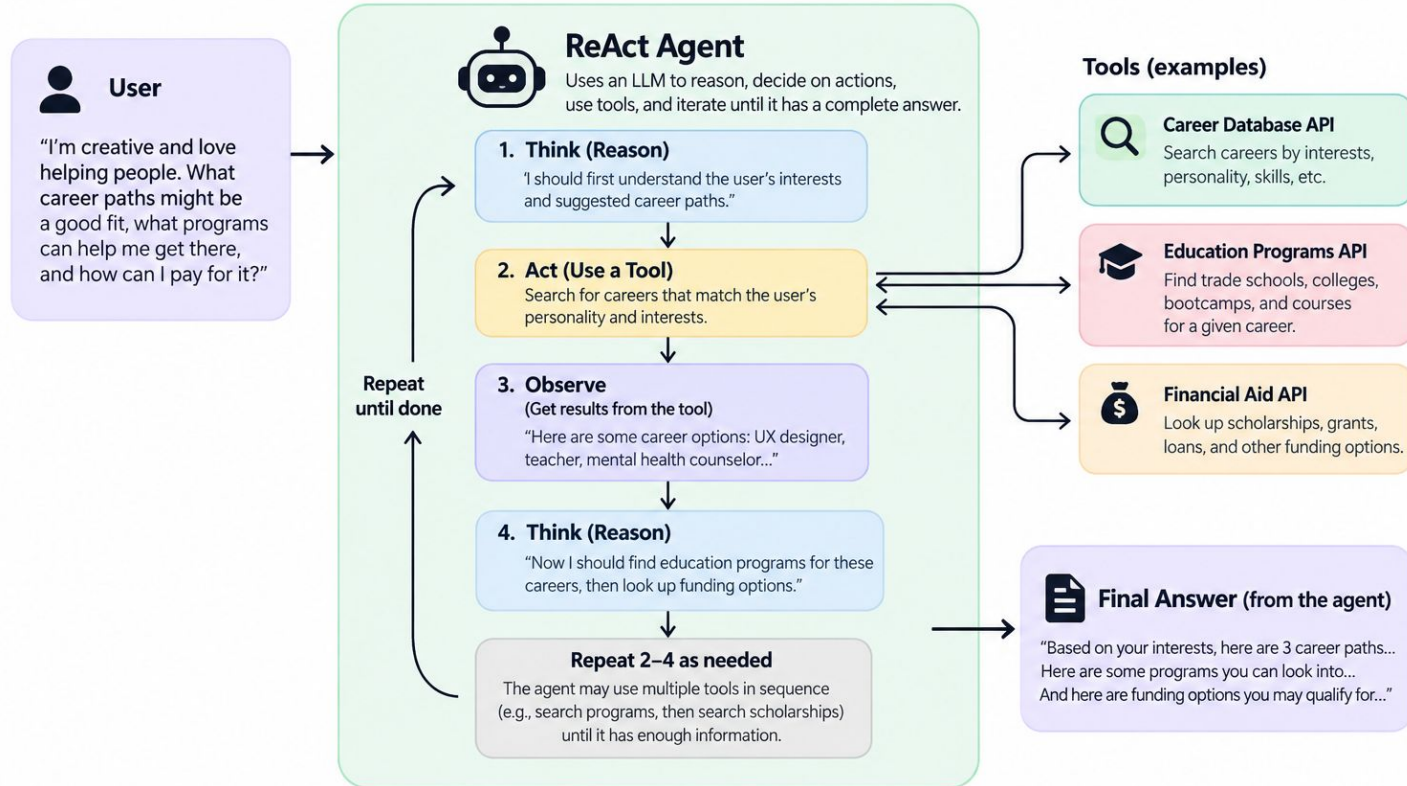
SINGLE AGENT SYSTEMS

REACT

- ReAct, not React
- Reasoning and Acting
- Think, act, observe, think
- From 2022, chain-of-thought mixed with acting (do something)

Single Agent System: ReAct (Reason + Act)

The agent reasons about what to do, takes an action (uses a tool), observes the result, and repeats until it can provide a final answer.

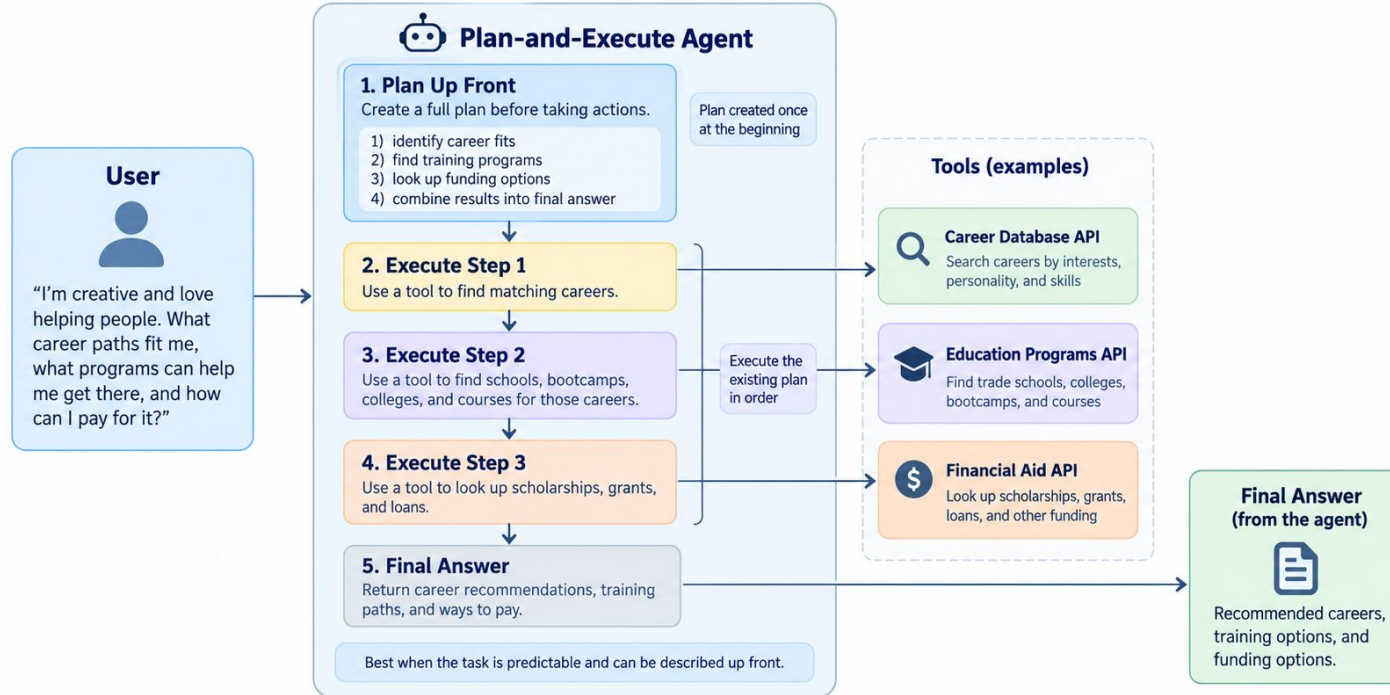


PLAN AND EXECUTE

- Makes the full plan up front
- Executes the plan
- Fewer model calls (usually cheaper)
- Worse at adapting
- Food for thought:
 - If we know the steps ahead of time, why not just make a workflow?

Single Agent System: Plan-and-Execute

The agent makes a plan up front, then executes the steps in order. Usually cheaper than ReAct, but less adaptive if the plan needs to change.

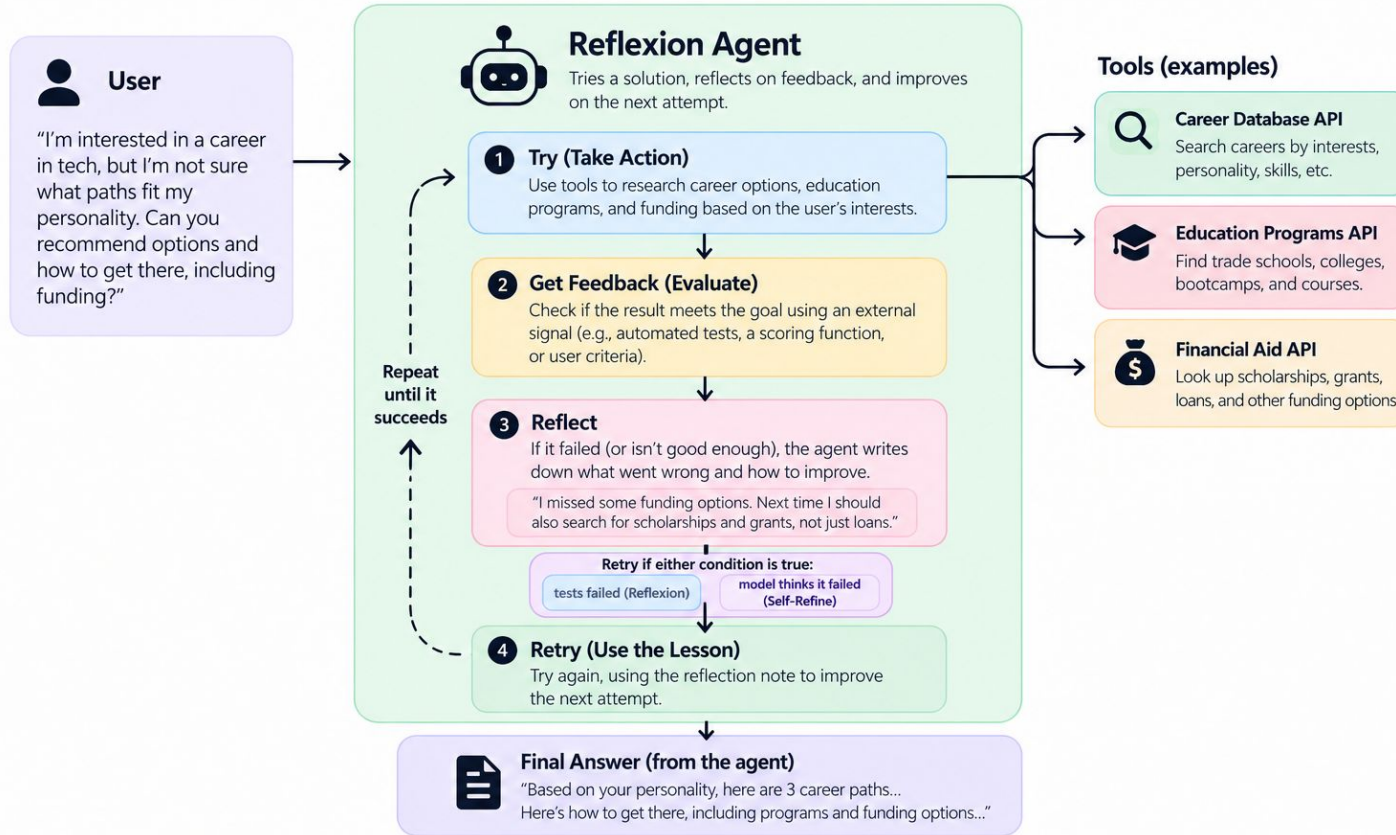


REFLEXION AND SELF-REFINE

- Something fails
- External failures like tests (reflexion) or model checking itself (self-refine)
- The agent writes down what it thinks went wrong
- Carries the failure information into the next attempt

Single Agent System: Reflexion / Self-Refine

The agent tries a solution, gets feedback on whether it worked, reflects on what went wrong, and carries that lesson into the next attempt.

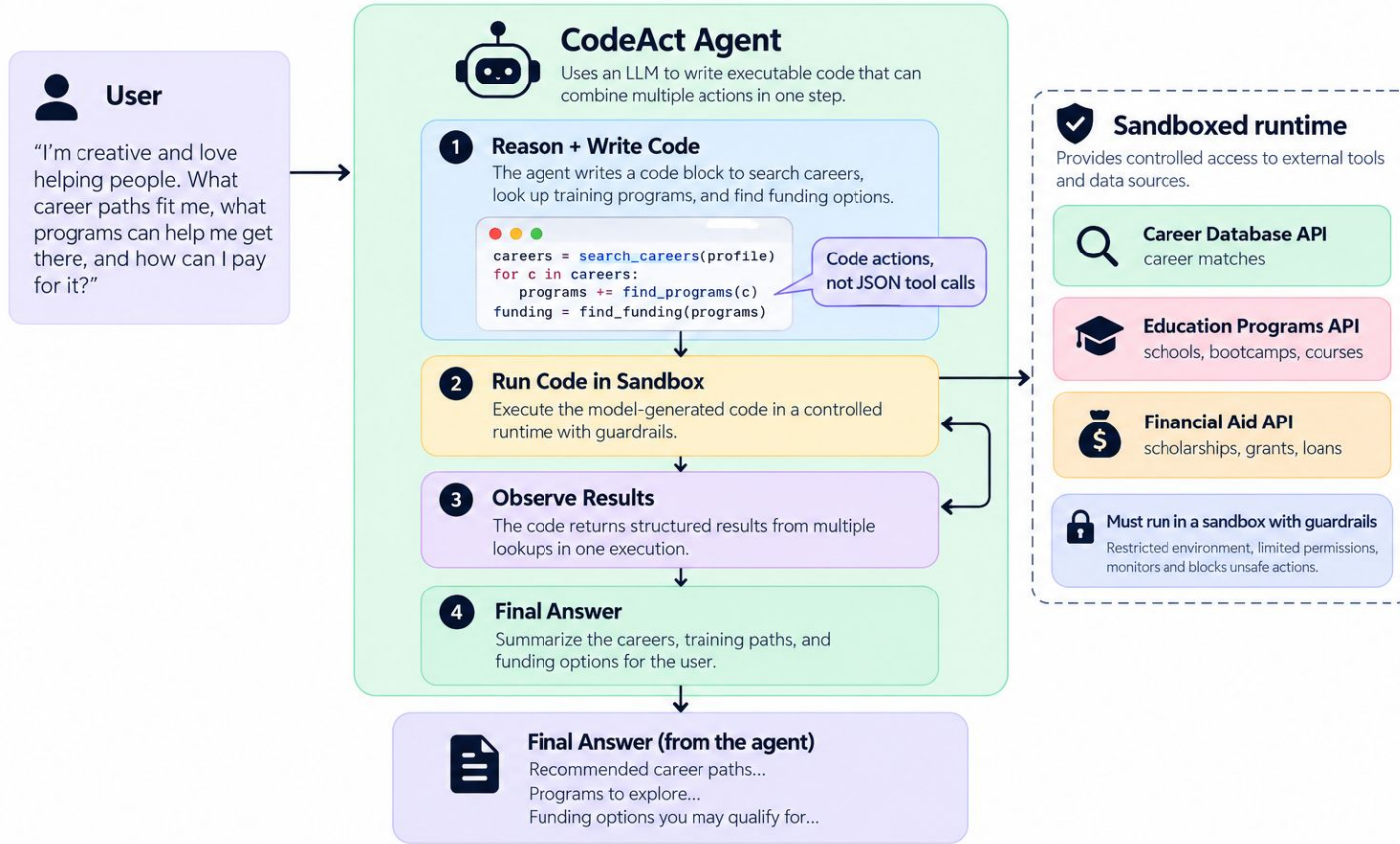


CODEACT

- CodeAct
- Agents actions are executable code, not JSON tool calls
- Code is in training data, your custom tools are not
- Tradeoff: executing model-generated code, so should be done in a sandbox!

Single Agent System: CodeAct

The agent writes executable code as its action, runs it in a sandbox, observes the result, and then returns an answer.



SINGLE AGENT WORKFLOW SCORECARD

- Context:
 - Worse, one window holds everything
- Reliability:
 - Worse, we don't know how many steps, so we can't judge how many places we can go wrong
- Cost:
 - Worse than workflows, more expensive running an agent loop
 - Better than multi-agent setups
- BUT - we got adaptability

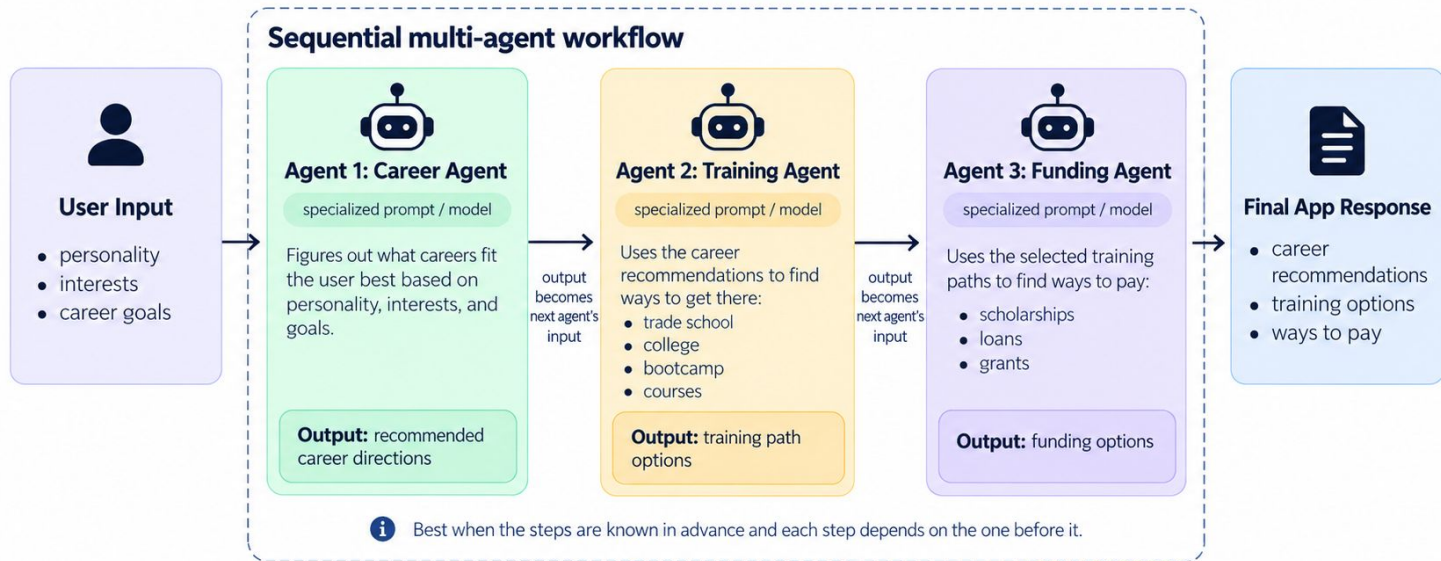
MULTI-AGENT SYSTEMS

SEQUENTIAL

- Output of one agent becomes in the input of the next
- Each step is a full, independent agent that can plan, use tools, reflect, etc.
- Each agent can run a different model/prompt/toolset
- Same topology as workflow prompt chaining just with agents
- Steps can fail, retry, and reflect before passing output downstream

Multi-Agent System: Sequential

A line of specialized agents: each agent finishes its step, then hands its output to the next agent.



SEQUENTIAL SCORECARD

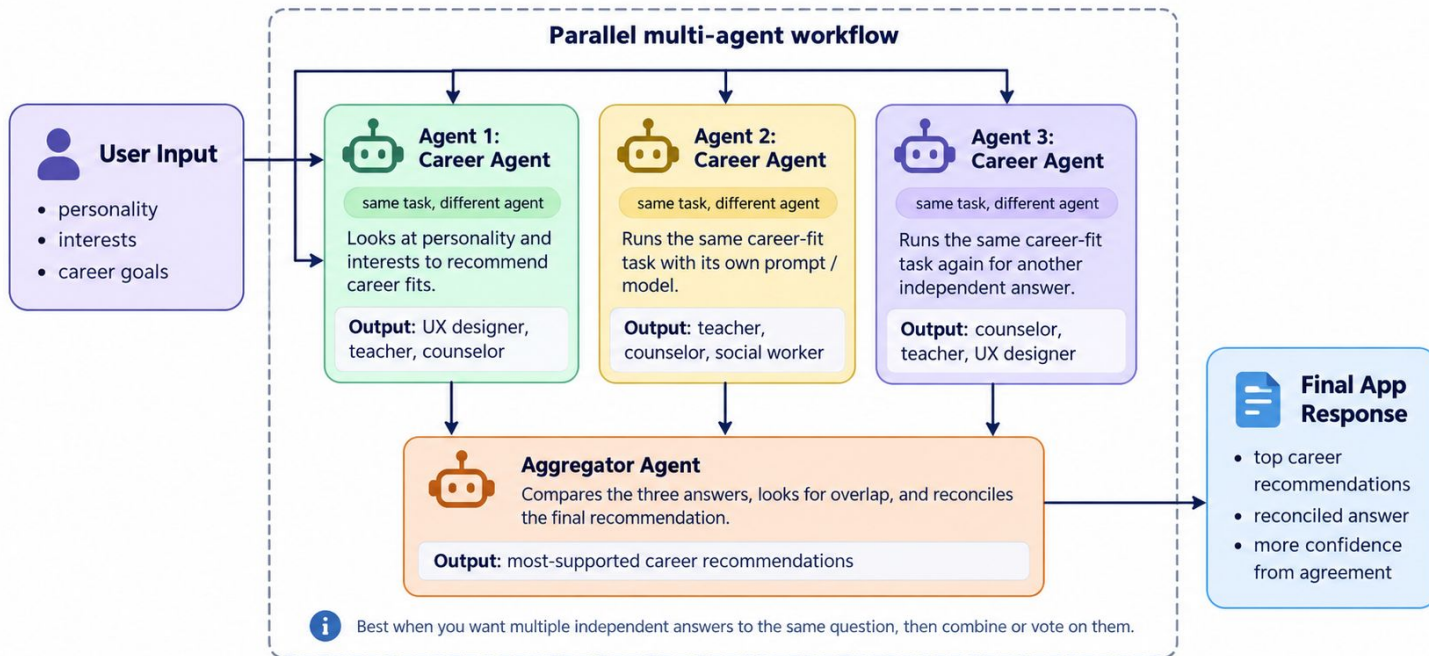
- Context
 - Good, each agent gets its own context
- Reliability
 - Errors compound down the chain, so validation is very important
- Cost
 - Usually the cheapest multi-agent
 - Latency is high and stacks linearly

PARALLELIZATION

- A request is split across multiple agents at the same time
- Two subtypes:
 - Sectioning – different agents handle subtasks in parallel
 - Voting – multiple agents attempt the same task and get reconciled
- Mirror-image tradeoffs on latency and cost with sequential

Multi-Agent System: Parallelization (Voting)

The same request goes to multiple agents at the same time. A final agent compares the answers and reconciles them into one response.



PARALLELIZATION SCORECARD

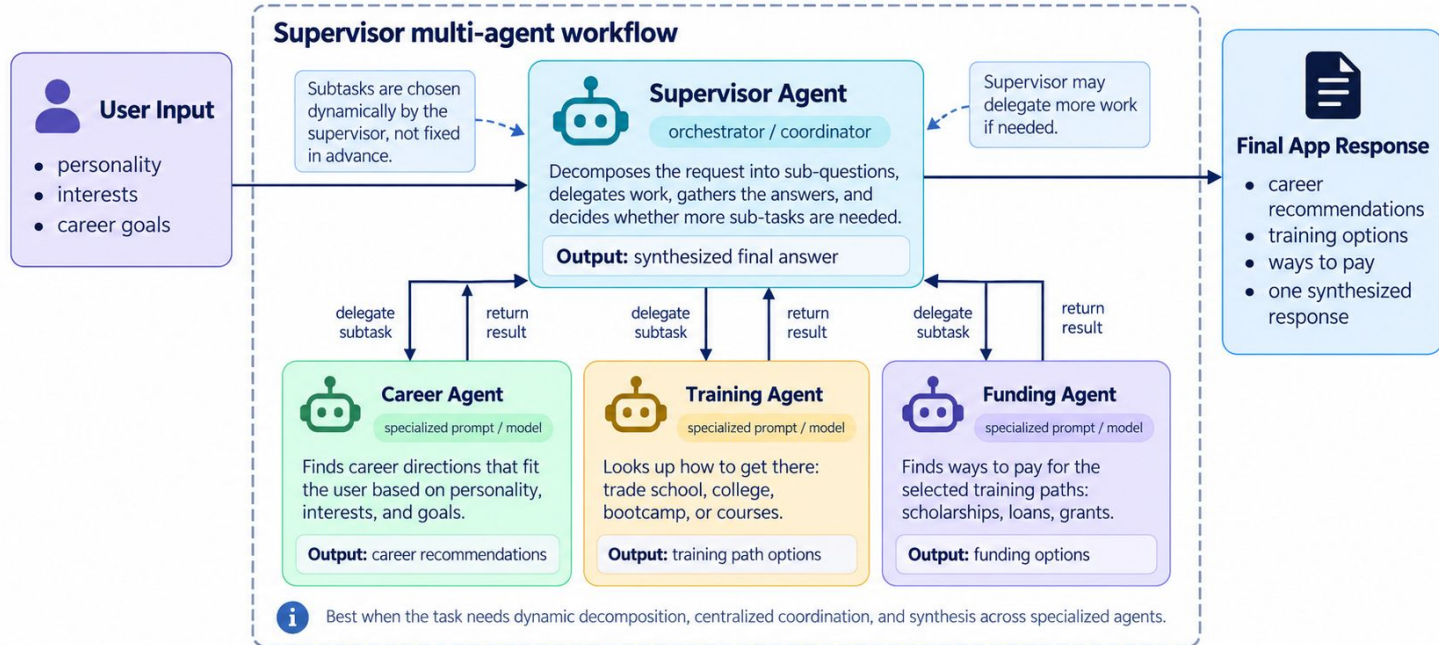
- Context:
 - Good, each agent has its own context
- Reliability:
 - Depends on subtypes, voting is better for redundancy
- Cost:
 - High in money
 - Low in latency

SUPERVISOR

- One main agent (the supervisor) coordinates the whole system
- Supervisor breaks the query into subtasks and delegates to subagents
- Subagent output gets fed back to the supervisor
- Subagents don't need to know about each other
- Can be nested
 - A supervisor subagent can itself be a supervisor over other subagents

Multi-Agent System: Supervisor

A supervisor agent receives the user request, decides the subtasks, delegates them to specialized agents, and then synthesizes the results into one final response.



SUPERVISOR SCORECARD

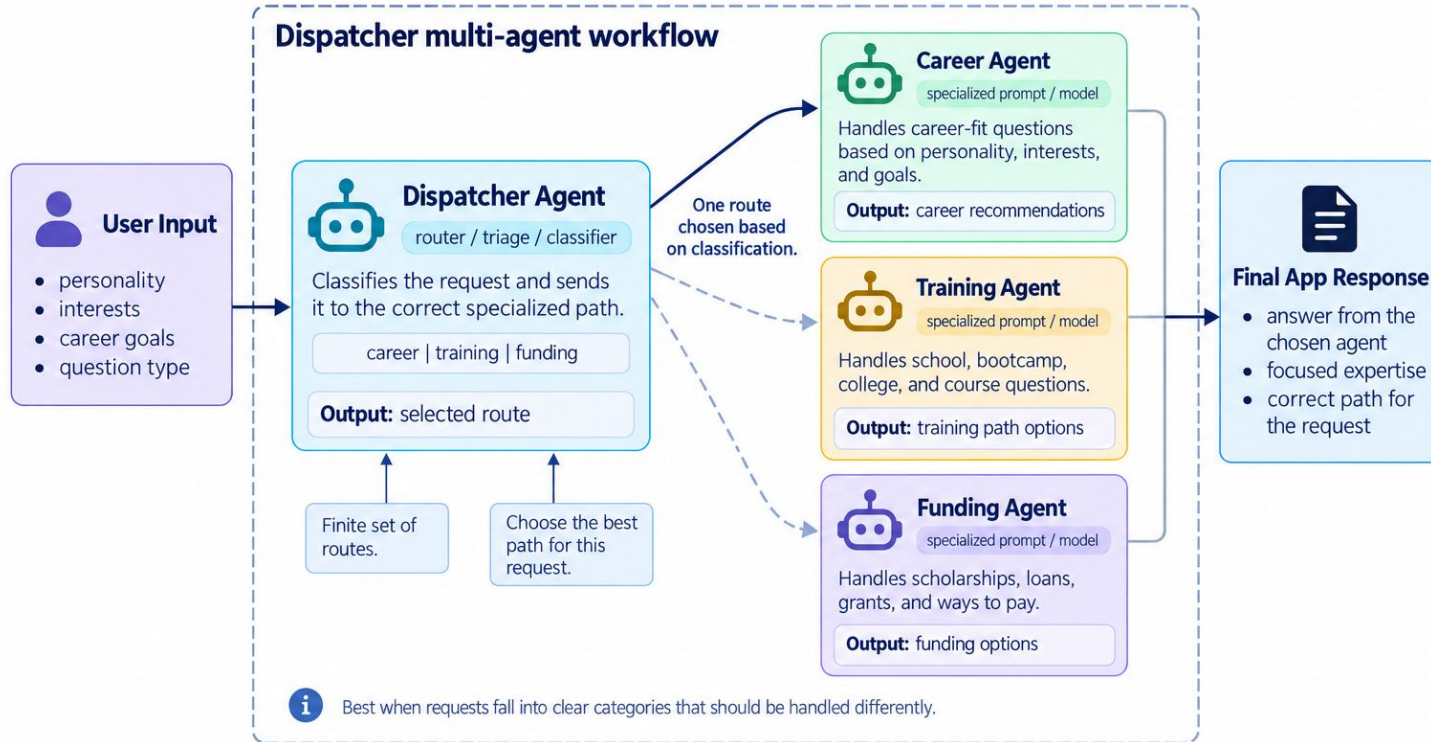
- Context:
 - Excellent – the supervisor holds the big picture, subagents hold narrow context
- Reliability:
 - Strong – focused chunks means subagents have simpler tasks to get right
 - Supervisor can catch/redirect if a subagent's output looks wrong
- Cost:
 - Very high
 - Supervisor model needs to be capable enough for context and subagent management
 - Many agents running and possibly a lot of supervisors

DISPATCHER

- Classification step examines input and decides which agent handles it
- Mutli-agent version of the routing workflow pattern
- Great for inputs that fall into clear, distinct categories
- Unlike supervisor delegation, only one path executes per input
- It's a fork, not a decomposition
- Bad routes are very hard to recover from

Multi-Agent System: Dispatcher

A dispatcher agent classifies the user request, then routes it to the one specialized agent that should handle it.



DISPATCHER SCORECARD

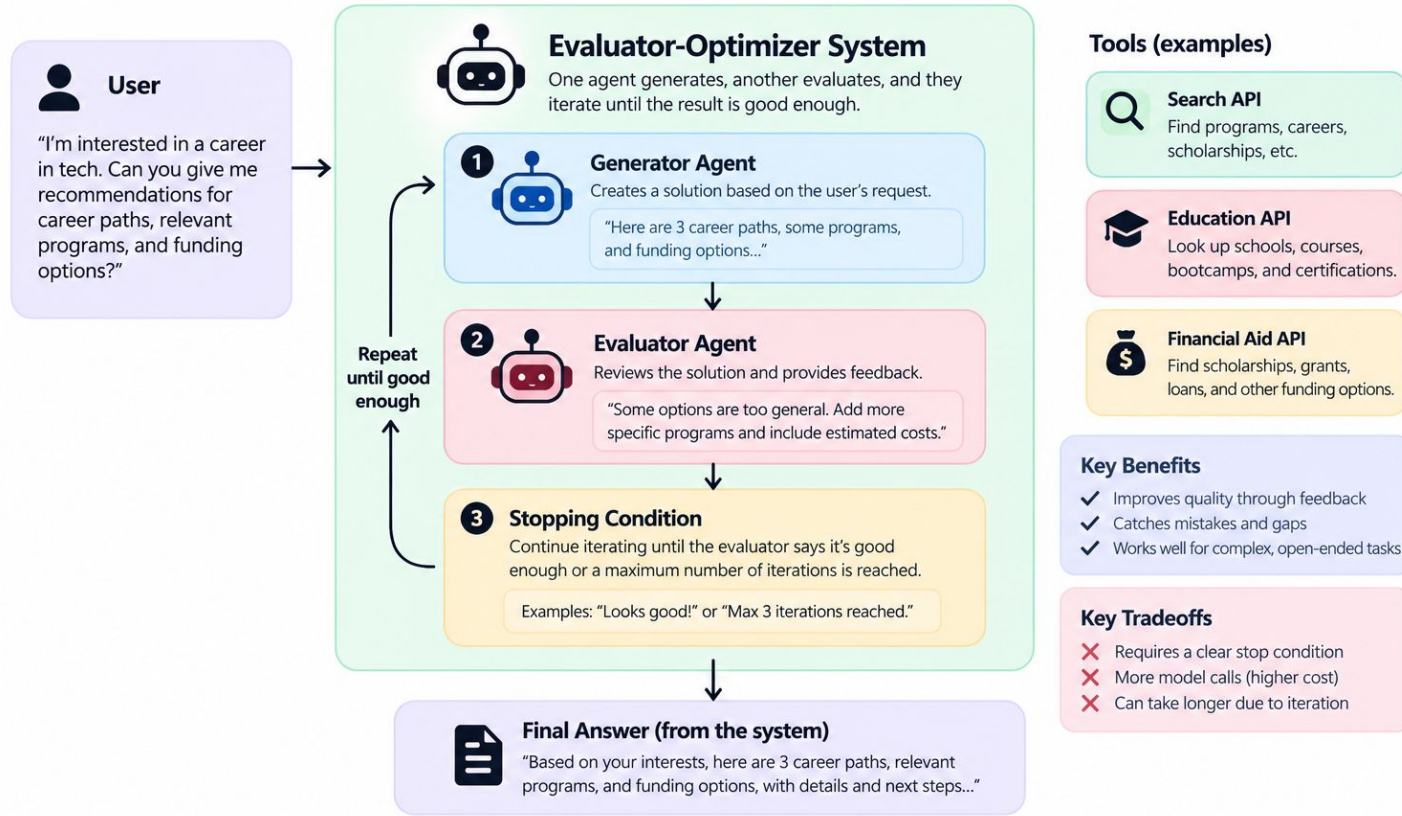
- Context:
 - Good - Stays small per agent
- Reliability:
 - Good/Bad - Bottlenecked on the classifier
- Cost:
 - Can be the cheapest of the multi-agent patterns
 - Route simple inputs to smaller models and reserve expensive models for harder work

EVALUATOR-OPTIMIZER

- One agent (the optimizer) does the work
- A second agent (the evaluator) checks it and gives critiques
- The loop repeats until the evaluator is satisfied or a stop condition is hit
- The multi-agent version of the workflow with the same name
 - But now each step is its own agent with its own reasoning and tools
- Needs an explicit stop condition for safety
 - Max-iteration cap
 - Timeout
- Reliability and cost are directly connected, high cost for high reliability

Multi-Agent System: Evaluator-Optimizer

One agent generates a solution, another evaluates it and gives feedback.
The generator uses that feedback to improve, repeating until the evaluator is satisfied
or a stop condition is reached.



EVALUATOR-OPTIMIZER SCORECARD

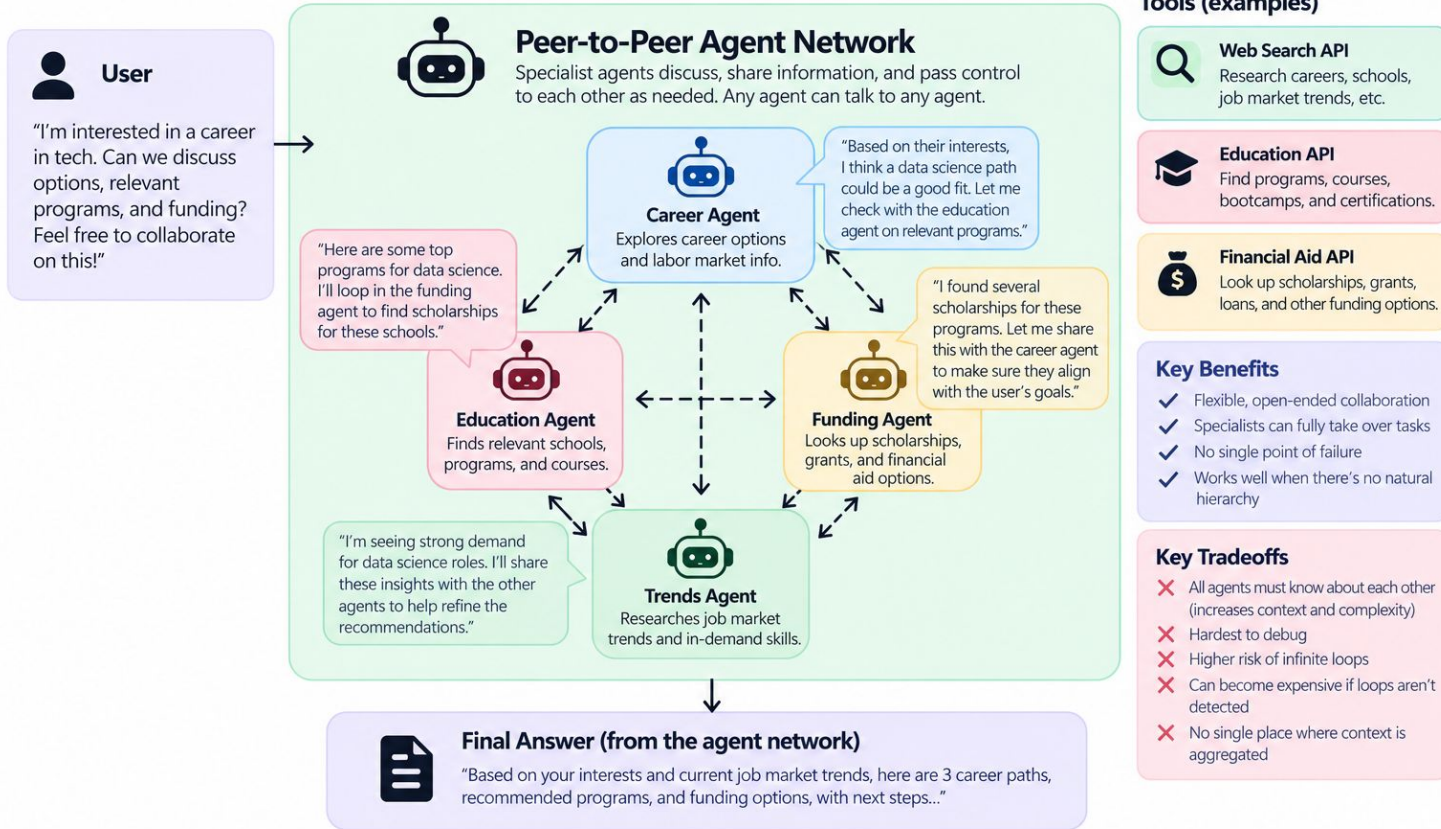
- Context:
 - Bad – Grows every round, each iteration adds the prior attempt + critique into context
- Reliability:
 - Strong – a dedicated critique step run repeatedly is good at catching issues
- Cost:
 - Can get very high fast
 - Without a stop condition can loop forever

PEER-TO-PEER

- No central coordinator, agents communicate directly with each other
- Every agent can talk to any other agent
- Every agent needs awareness of every other agent
- Agents pass info back and forth as needed until a stop condition is reached
- This pattern has no direct workflow equivalent
- Best fit: problems with no natural hierarchy and decomposition isn't known upfront

Multi-Agent System: Peer-to-Peer

Agents all work on equal footing and can pass control to any other agent at any time.
There is no central coordinator — they collaborate directly with each other.



PEER-TO-PEER SCORECARD

- Context:
 - Bad – every agent needs context on every other agent
 - Context scales with number of relationships not just number of agents
- Reliability:
 - Depends but without a central agent checking, usually worse than other patterns
 - Coordination failures are easy to happen as well
- Cost:
 - Very expensive, combines many agents running and compounding rounds
- Usually hardest to debug, and last resort rather than the default choice!

MULTI-AGENT SYSTEMS WRAP UP

EVALUATIONS

WHY BENCHMARKS FALL SHORT

- Leaderboards usually show one scorecard column (reliability)
 - Hide both cost and context
- Final answer vs trajectory
 - Right answer, but how did it get there?
- Trajectory checks are easy to make too rigid, agents find valid routes we didn't plan
- Many benchmarks run only once!

METRICS

- $\text{pass}@k$
 - At least one success in k tries
 - Should rise as k rises
 - Important where one success matters
- pass^k
 - All k tries succeed
 - Will fall as k increases
 - Suits customer facing agents

BUILDING YOUR OWN EVALS

- Start with 20-50 simple tasks from real failures
- Clear pass/fail beats vague rubrics
- Test when behaviors *should* happen and when it *should not*
- Clean environments, as close to production as possible
 - But with as little extra environment config as possible

SO, WHAT DO I USE AND WHEN?

WHEN TO REACH FOR EACH

- Know the steps every time with one hard step in the middle?
 - LLM powered workflow
- Unknown path but a known set of paths?
 - Single agent
- Independent subproblems?
 - Multiple agents