Preface + Chapters 2,3,4

Preface

What are the key characteristics of AI agents?

What is Agentic AI and how does it differ from traditional AI systems?

What are the core components of an agentic AI architecture?

What are metacognitive abilities in the context of AI agents?

Agentic system is a computational entity designed to perceive its environment (both digital and potentially physical), make informed decisions based on those perceptions and a set of predefined or learned goals, and execute actions to achieve those goals autonomously.

Unlike traditional software, which follows rigid, step-by-step instructions, agents exhibit a degree of flexibility and initiative

Autonomy = acting without constant human oversight;

Proactiveness, initiating actions towards their goals;

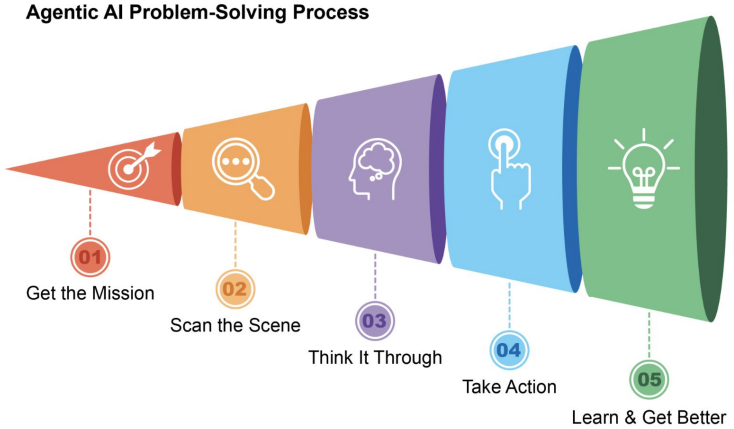
Reactiveness, responding effectively to changes in their environment.

Goal-oriented, constantly working towards objectives.

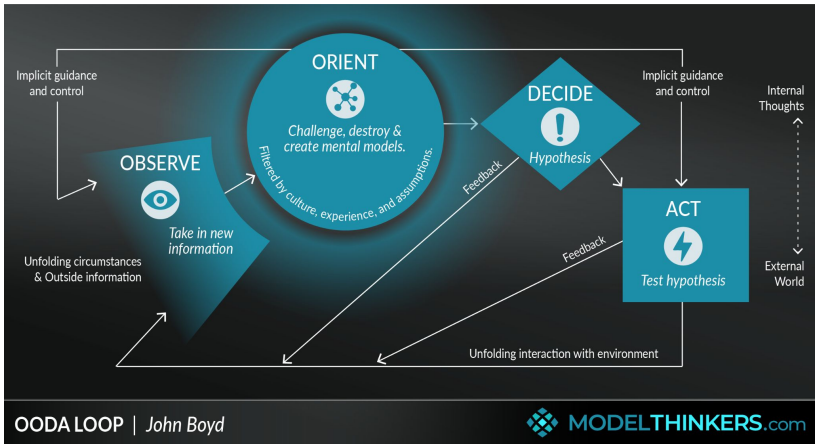
Tool use, enabling them to interact with external APIs, databases, or services – effectively reaching out beyond their immediate canvas.

Memory, retain information across interactions,

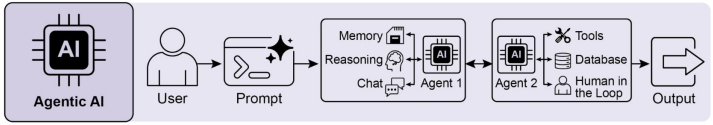
Communication with users, other systems, or even other agents



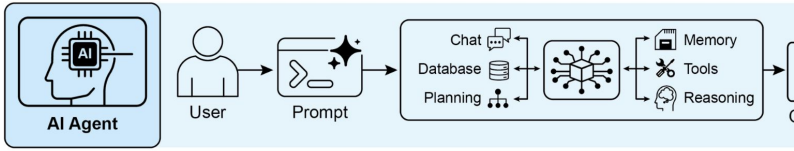
OODA Loop



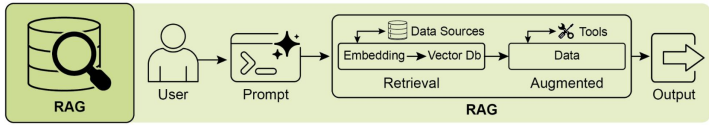
Level 3: The Rise of Collaborative Multi-Agent Systems



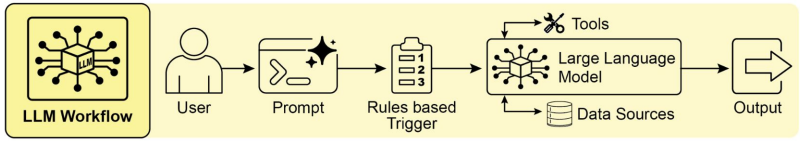
Level 2: The Strategic Problem-Solver



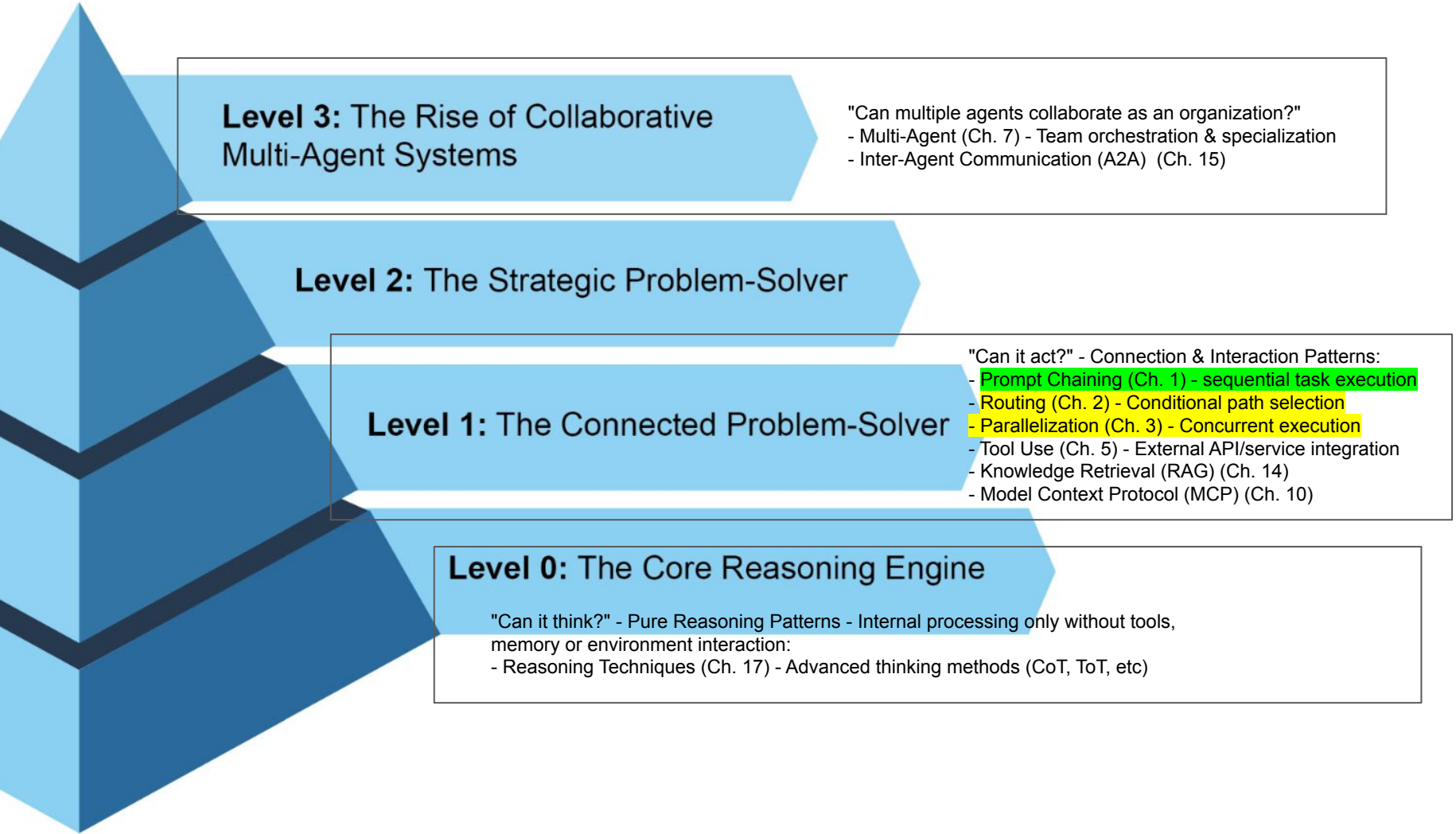
Level 1: The Connected Problem-Solver

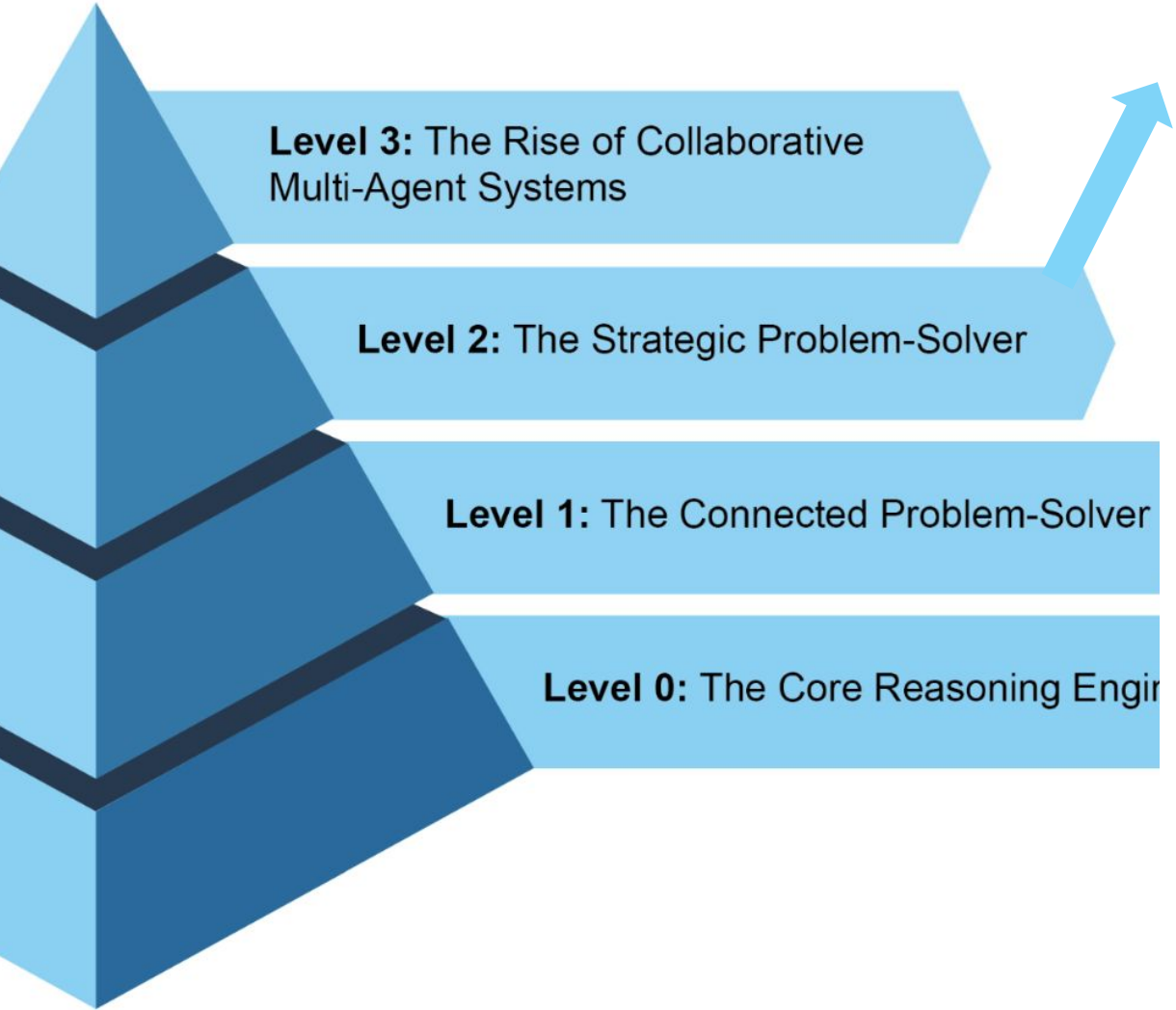


Level 0: The Core Reasoning Engine



[AI Agents vs. Agentic AI: A Conceptual taxonomy, applications and challenges - Feb'26]


Level 3: The Rise of Collaborative Multi-Agent Systems
"Can multiple agents collaborate as an organization?"
- Multi-Agent (Ch. 7) - Team orchestration & specialization
- Inter-Agent Communication (A2A) (Ch. 15)
Level 2: The Strategic Problem-Solver
Level 1: The Connected Problem-Solver
"Can it act?" - Connection & Interaction Patterns:
- Prompt Chaining (Ch. 1) - sequential task execution
- Routing (Ch. 2) - Conditional path selection
- Parallelization (Ch. 3) - Concurrent execution
- Tool Use (Ch. 5) - External API/service integration
- Knowledge Retrieval (RAG) (Ch. 14)
- Model Context Protocol (MCP) (Ch. 10)
Level 0: The Core Reasoning Engine
"Can it think?" - Pure Reasoning Patterns - Internal processing only without tools, memory or environment interaction:
- Reasoning Techniques (Ch. 17) - Advanced thinking methods (CoT, ToT, etc)



Level 2: The Strategic Problem-Solver - "Can it strategize, learn, and operate reliably?"

This section is subdivided for human readability into:

Meta Cognition Patterns:

- Reflection (Ch. 4) - Self-evaluation capabilities
- Planning (Ch. 6) - Multi-step strategy formulation
- Goal Setting & Monitoring (Ch. 11) - Objective tracking
- Prioritization (Ch. 20) - Action selection & ranking

Continuous efficiency:

- Memory Management (Ch. 8)
- Learning & Adaptation (Ch. 9) - self-improvement loops
- Resource-Aware Optimization (Ch. 16)

Production readiness (safety and reliability):

- Exception Handling & Recovery (Ch. 12) - Error resilience
- Guardrails/Safety Patterns (Ch. 18) - Behavioral constraints
- Evaluation & Monitoring (Ch. 19) - Performance tracking

Collaboration and exploration:

- Human-in-the-Loop (Ch. 13) - human oversight & intervention
- Exploration & Discovery (Ch. 21) - novel solution finding

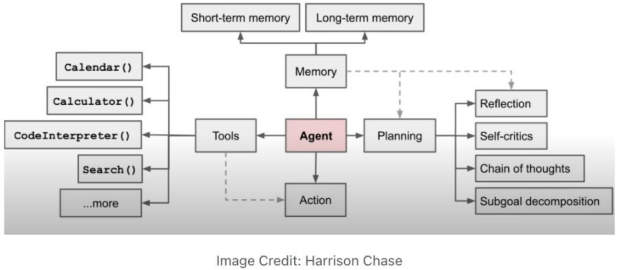


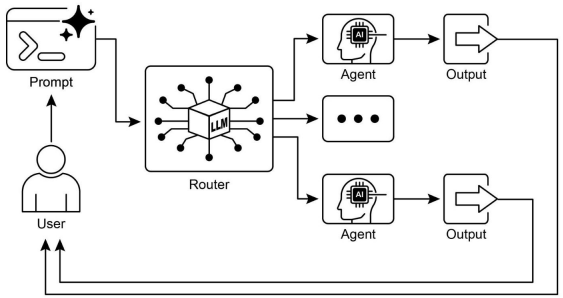
Image Credit: Harrison Chase

Chapter 2 - Routing

How does a routing agent decide which specialized agent should handle a task?

How would you implement dynamic routing based on query intent?

Explain the concept of a "supervisor pattern" in agent orchestration?



Routing - arbitrating between multiple potential actions based on contingent factors (state of the environment, user input or outcome of a preceding operation)

- Rule-based (if-else, switch)
- Semantic or Embedding-based (semantic_router library)
- ML-based using a classifier or any other discriminative model trained on a small set of labeled data
- LLM-based:”analyze following query and output only one of the categories - “Product Status”, “User Support”, “Other”” = supervisor pattern
- Auction-based routing (not in the book) - more than one agent “bids” a score based on their confidence level in handling the given query.

Best Practices for Implementing Agent Routing

- Use unambiguous routing criteria with guardrails (MECE ?)
- Implement default fall-back path (human, general, etc)
- Measure the accuracy of routing separately
- Consider the cost of routing
- Consider load-balancing
- Add feedback loops (user driven or reflection pattern) to improve

https://fme.safe.com/guides/ai-agent-architecture/ai-agent-routing/
https://www.analyticsvidhya.com/blog/2024/08/mastering-llm-routing/

Aspect	Static Routing	Dynamic Routing	Model-Aware Routing
Definition	Uses predefined rules to direct tasks.	Adapts routing decisions in real-time based on current conditions.	Routes tasks based on model capabilities and performance.
Implementation	Implemented through static configuration files or code.	Requires real-time monitoring systems and dynamic decision-making algorithms.	Involves integrating model performance metrics and routing logic based on those metrics.
Adaptability to Changes	Low; requires manual updates to rules.	High; adapts automatically to changes in conditions.	Moderate; adapts based on predefined model performance characteristics.
Complexity	Low; straightforward setup with static rules.	High; involves real-time system monitoring and complex decision algorithms.	Moderate; involves setting up model performance tracking and routing logic based on those metrics.
Scalability	Limited; may need extensive reconfiguration for scaling.	High; can scale efficiently by adjusting routing dynamically.	Moderate; scales by leveraging specific model strengths but may require adjustments as models change.
Resource Efficiency	Can be inefficient if rules are not well-aligned with system needs.	Typically efficient as routing adapts to optimize resource usage.	Efficient by leveraging the strengths of different models, potentially optimizing overall system performance.
Implementation Examples	Static rule-based systems for fixed tasks.	Load balancers with real-time traffic analysis and adjustments.	Model-specific routing algorithms based on performance metrics (e.g., task-specific model deployment).

Aspect	Consistent Hashing	Contextual Routing
Definition	A technique for distributing tasks across a set of nodes based on hashing, which ensures minimal reorganization when nodes are added or removed.	A routing strategy that adapts based on the context or characteristics of the request, such as user behavior or request type.
Implementation	Uses hash functions to map tasks to nodes, often implemented in distributed systems and databases.	Utilizes contextual information (e.g., request metadata) to determine the optimal routing path, often implemented with machine learning or heuristic-based approaches.
Adaptability to Changes	Moderate; handles node changes gracefully but may require rehashing if the number of nodes changes significantly.	High; adapts in real-time to changes in the context or characteristics of the incoming requests.
Complexity	Moderate; involves managing a consistent hashing ring and handling node additions/removals.	High; requires maintaining and processing contextual information, and often involves complex algorithms or models.
Scalability	High; scales well as nodes are added or removed with minimal disruption.	Moderate to high; can scale based on the complexity of the contextual information and routing logic.
Resource Efficiency	Efficient in balancing loads and minimizing reorganization.	Potentially efficient; optimizes routing based on contextual information but may require additional resources for context processing.
Implementation Examples	Distributed hash tables (DHTs), distributed caching systems.	Adaptive load balancers, personalized recommendation systems.

Example - GitHub Copilot Workspace

GitHub Copilot Workspace Uses Routing: From the documentation: "Everything in Copilot Workspace begins with a 'task', which is a natural language description of intent... Copilot Workspace supports four types of tasks: solving issues, refining pull requests, creating repositories from templates and ad-hoc tasks." https://github.com/githubnext/copilot-workspace-user-manual**, technical overview**: https://github.com/githubnext/copilot-workspace-user-manual/blob/main/overview.md

The system routes tasks through specialized agents:

- **Plan Agent**: Captures intent, proposes plan
- **Brainstorm Agent**: Discusses alternatives
- **Repair Agent**: Fixes test failures
- **Specification Agent**: Defines success criteria

https://github.blog/ai-and-ml/github-copilot/how-were-making-github-copilot-smarter-with-fewer-tools/

*"We're u**sing embedding-guided tool routing, adaptive clustering, and a streamlined 13-tool core to deliver faster experience** in we applied our internal Copilot embedding model optimized for semantic similarity tasks to generate embeddings for each tool and group them using cosine similarity."*

Routing may require additional promoting for quality and efficiency

Tool Selection Guide

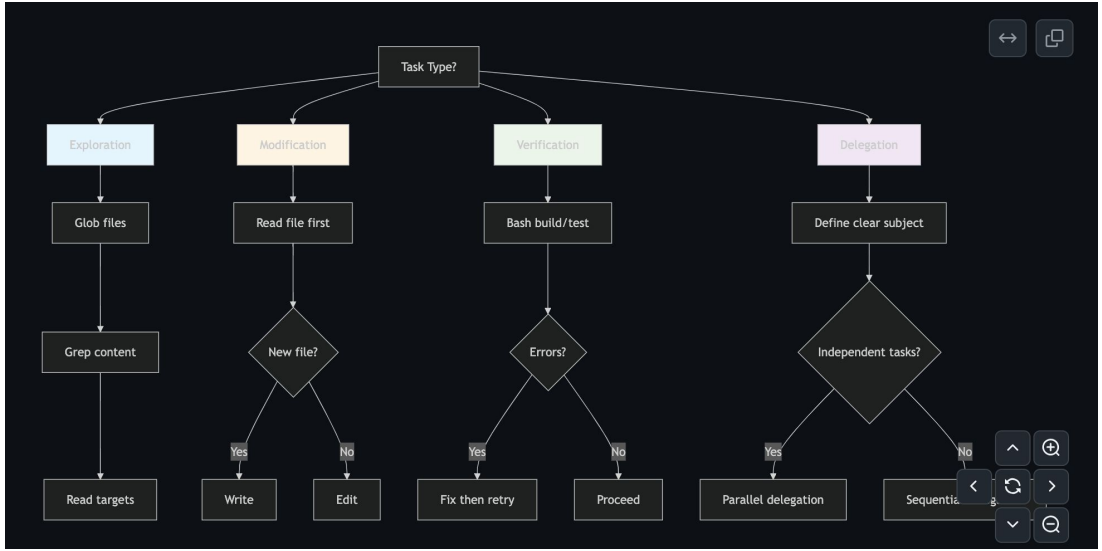
Problem: AI agents often struggle to select the optimal tool for a given task, leading to inefficient workflows. Common anti-patterns include:

- Using Write when Edit would be more appropriate
- Launching subagents for simple exploration tasks
- Skipping build verification after code changes
- Performing sequential exploration when parallel would be faster

These inefficiencies compound across long sessions, resulting in wasted tokens, slower completion, and more user corrections.

Solution: **provide tool selection guide for repeating tasks.**

Also see: query reformulation after intent detection: https://github.com/arunpshankar/Agentic-Workflow-Patterns



Routing is relevant for security and guardrails

Action-Selector Pattern

Problem: Untrusted input can hijack an agent's reasoning once tool feedback re-enters the context window, leading to arbitrary, harmful actions.

Solution: treat the LLM as an **"instruction decoder" only - provide a hard allowlist of safe actions (API calls, SQL templates, page links)**. Useful for customer-service bots, notification routers, kiosk interfaces:

- Map the user's natural-language request to a pre-approved action (or action template).
- No tool outputs are fed back into the LLM.
- The agent therefore cannot be influenced after selecting the action.

action = LLM.translate(prompt, allowlist)
execute(action)
 # tool output NOT returned to LLM

Trade-offs

- Pros: Near-immunity to prompt injection; trivial to audit.
- Cons: Limited flexibility; new capabilities require code updates.

Dual-LLM Pattern

Problem: A privileged agent that both sees untrusted text and wields tools can be coerced into dangerous calls.

Solution: **Split roles as privileged - quarantined**, especially for email/calendar assistants, booking agents, API-powered chatbots:

- Privileged LLM: Plans and calls tools but never sees raw untrusted data.
- Quarantined LLM: Reads untrusted data but has zero tool access.
- Pass data as symbolic variables or validated primitives; privileged side only manipulates references.

var1 = QuarantineLLM("extract email", text) # returns $VAR1
PrivLLM.plan("send $VAR1 to boss") # no raw text exposure
execute(plan, subst={ "$VAR1": var1 })

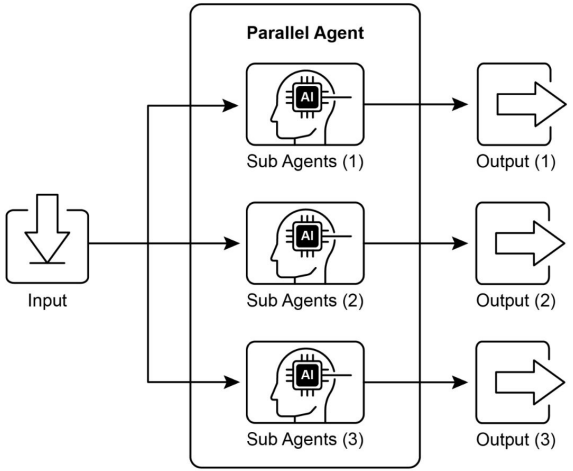
Trade-offs

- Pros: Clear trust boundary; compatible with static analysis.
- Cons: Complexity; debugging across two minds.

Chapter 3 - Parallelisation

What are the benefits and trade-offs of parallel versus sequential execution?

What are the challenges of coordinating results from parallel agents?



Parallelisation allows independent tasks to run at the same time, significantly reducing the overall execution time

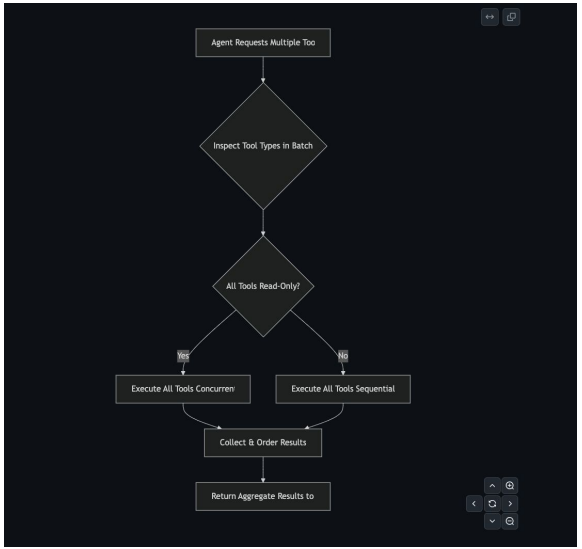
Parallelisation requires frameworks that support asynchronous execution or multi-threading/multi-processing. High Impact Scenarios:

1. Tool execution << Inference time: If tools return in 100ms but inference takes 2s, parallelization saves multiple inference rounds
    - Example: API calls, database queries, file reads
2. **Independent information gathering (“read-only”)**
    - Multiple search queries
    - Reading multiple files
    - Checking multiple conditions
3. Broad exploration phases
    - Initial reconnaissance (find all Python files, check all tests, etc.)
    - Multi-source research (check docs, code, tests simultaneously)

Low Impact Scenarios:

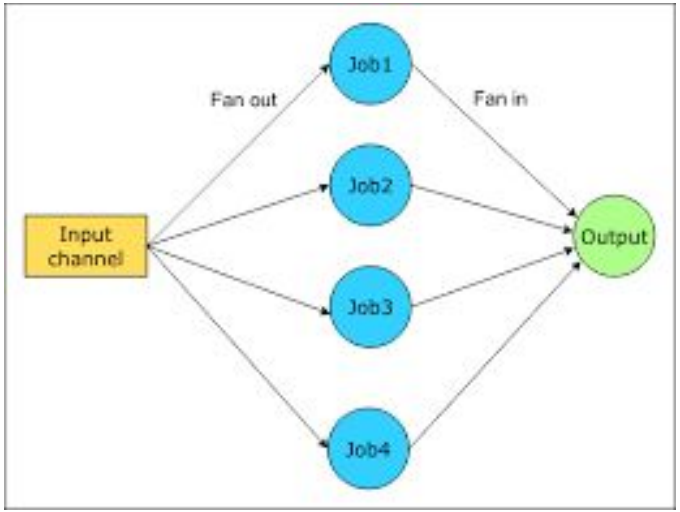
1. Sequential dependencies
    - Must read file A to know which file B to read
    - **Each tool result determines next action or changes environment (“write”)**
2. Tool execution >> Inference time
    - If each tool takes 10s and inference takes 1s, parallelization saves less
    - Example: Heavy computation, large file processing
3. **Rate-limited APIs**
    - **External APIs that throttle concurrent requests**
    - **Parallel calls just hit rate limits**

Read-Write Tool Classification for Parallelisation, Parallel Tool Calling (RF exploration)



Parallelisation is known as “fan-out fan-in pattern” in software development

LLM Map-Reduce uses parallelisation to eliminate injecting a single poisoned document that can manipulate global reasoning if all data is processed in one context.
- Map: Spawn lightweight, sandboxed LLMs - each ingests one untrusted chunk and emits a constrained output (boolean, JSON schema, etc.).
- Reduce: Aggregate those safe summaries with either deterministic code or a privileged LLM that sees only sanitized fields.



Parallelisation used to improve quality of the outputs

Best of N-delegation

Recursive delegation (parent agent -> sub-agents -> sub-sub-agents) is great for decomposing big tasks, but it has a failure mode:

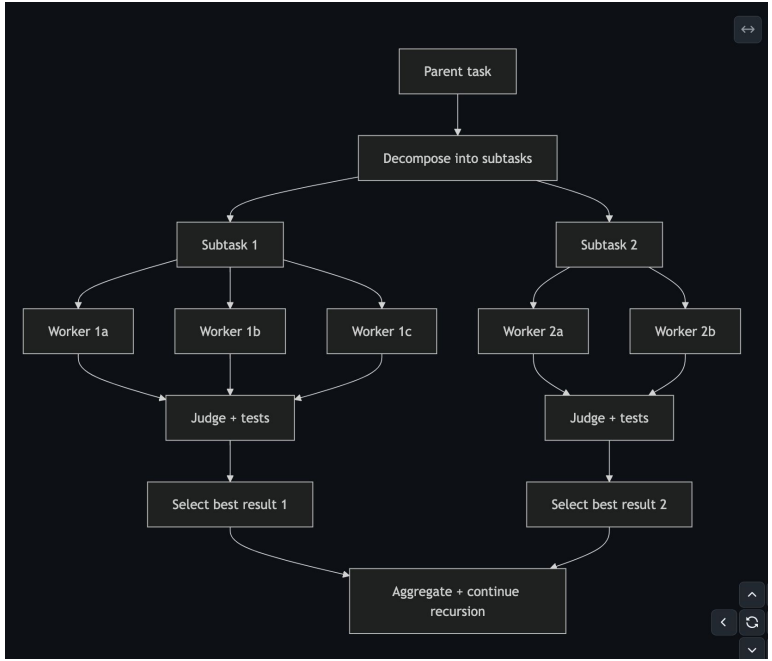
- A single weak sub-agent result can poison the parent's next steps (wrong assumption, missed file, bad patch)
- Errors compound up the tree: "one bad leaf" can derail the whole rollout
- Pure recursion also underuses parallelism when a node is uncertain: you really want multiple shots right where the ambiguity is

Meanwhile, "best-of-N" parallel attempts help reliability, but without structure they waste compute by repeatedly solving the same problem instead of decomposing it.

Solution

At each node in a recursive agent tree, run best-of-N for the current subtask before expanding further:

1. Decompose: Parent turns task into sub-tasks (like normal recursive delegation)
2. Parallel candidates per subtask: For each subtask, spawn K candidate workers in isolated sandboxes (K=2-5 typical)
3. Score candidates: Use a judge that combines:
    - Automated signals (tests, lint, exit code, diff size, runtime)
    - LLM-as-judge rubric (correctness, adherence to constraints, simplicity)
4. Select + promote: Pick the top candidate as the "canonical" result for that subtask
5. Escalate uncertainty: If the judge confidence is low (or candidates disagree), either:
    - Increase K for that subtask, or
    - Spawn a focused "investigator" sub-agent to gather missing facts, then re-run selection
6. Aggregate upward: Parent synthesizes selected results and continues recursion



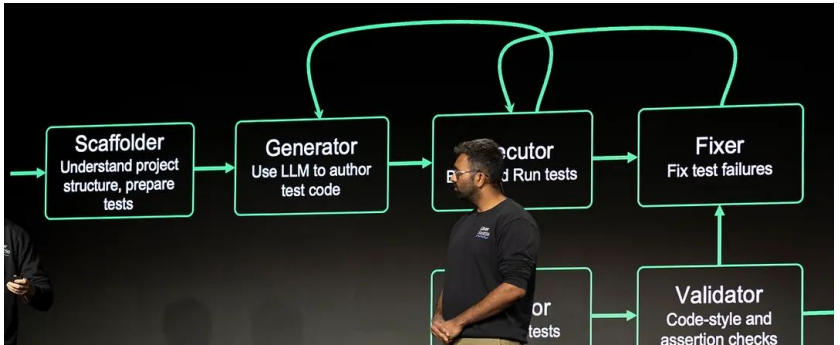
Example - UBER illustrates both conversations about parallelisation: tools and agents

UBER: LangGraph AI Agents (2025) - AutoCover - Parallel Test Generation System - 21,000 developer hours saved (Video, Case Study, Medium)
Key Details:

- Parallel Execution: Up to 100 test generation iterations running simultaneously on large source files 2-3x faster than industry-standard agentic coding tools
- Scale: Supporting 5,000 engineers, hundreds of millions of lines of code, 33 million trips daily
- The first major product was Validator LangGraph agent (consists of multiple sub-agents working together under a central coordinator) that automatically flags best practices violations and security issues providing real-time feedback directly in the developer's IDE environment.
    a. (tools use, ch.5) Value of combining LLM-based agents with deterministic sub-components when possible. The lint agent within Validator demonstrates this principle - by using reliable static analysis tools for certain types of issues, they achieve consistent output quality while reserving LLM capabilities for more complex reasoning tasks.
- **How parallelisation is used:**
    a. **Specialized agents (scaffolder, generator, executor) working in parallel:** a scaffolder that prepares the test environment and identifies business cases, a generator that creates new test cases, and an executor that runs builds and coverage analysis.
    b. Validator agent reused as a component, demonstrating the composability benefits of their architecture.
    c. **Executor agent is domain-context-rich - it includes sophisticated knowledge about Uber's build system that enables parallel test execution without conflicts and separate coverage reporting**

Specific Parallelization Patterns:

- Parallel test execution without conflicts
- Separate coverage reporting per parallel worker
- Domain-expert agents with rich state management



Chapter 4 - Reflection

What is reflection in AI agents and why is it important?

How do you implement self-correcting loops in agent workflows?

What's the difference between reflection and simple error handling?

Reflection is ability to think about own actions and results to self-correct and improve dynamically

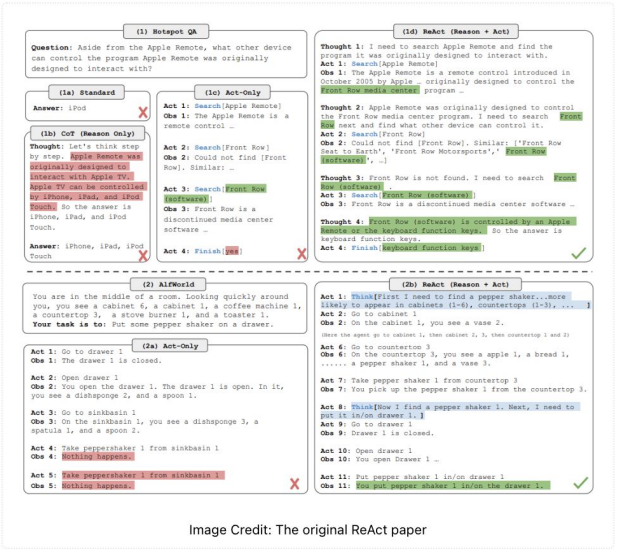
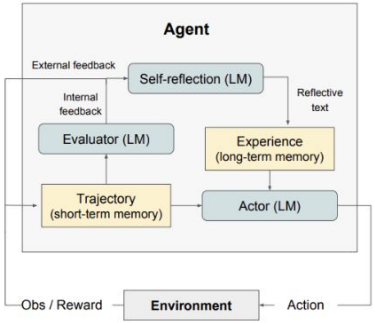


Image Credit: The original ReAct paper



Algorithm 1 Reinforcement via self-reflection

Initialize Actor, Evaluator, Self-Reflection: M_a, M_e, M_{sr}
Initialize policy $\pi_\theta(a_i|s_i), \theta = \{M_a, mem\}$
Generate initial trajectory using π_θ
Evaluate τ_0 using M_e
Generate initial self-reflection sr_0 using M_{sr}
Set $mem \leftarrow [sr_0]$
Set $t = 0$
while M_e not pass or $t <$ max trials **do**
  Generate $\tau_t = [a_0, o_0, \dots a_i, o_i]$ using π_θ
  Evaluate τ_t using M_e
  Generate self-reflection sr_t using M_{sr}
  Append sr_t to mem
  Increment t
end while
return

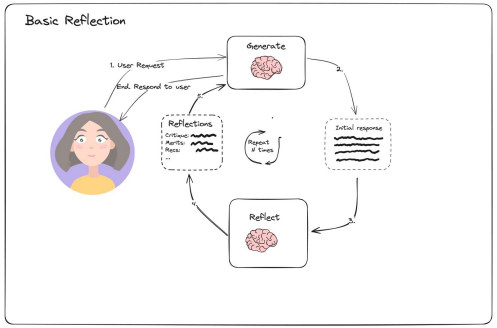
Image Credit: The original Reflexion paper

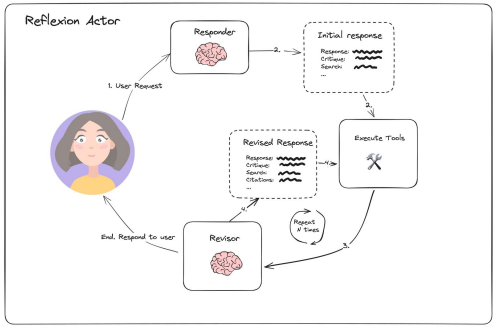
[ReAct: Synergizing Reasoning and Acting in Language Models] (2022)
(reason - act (mid-reasoning) - observe - reason - act - observe)
“Think and Do” = to answer this I need more research (call a tool)

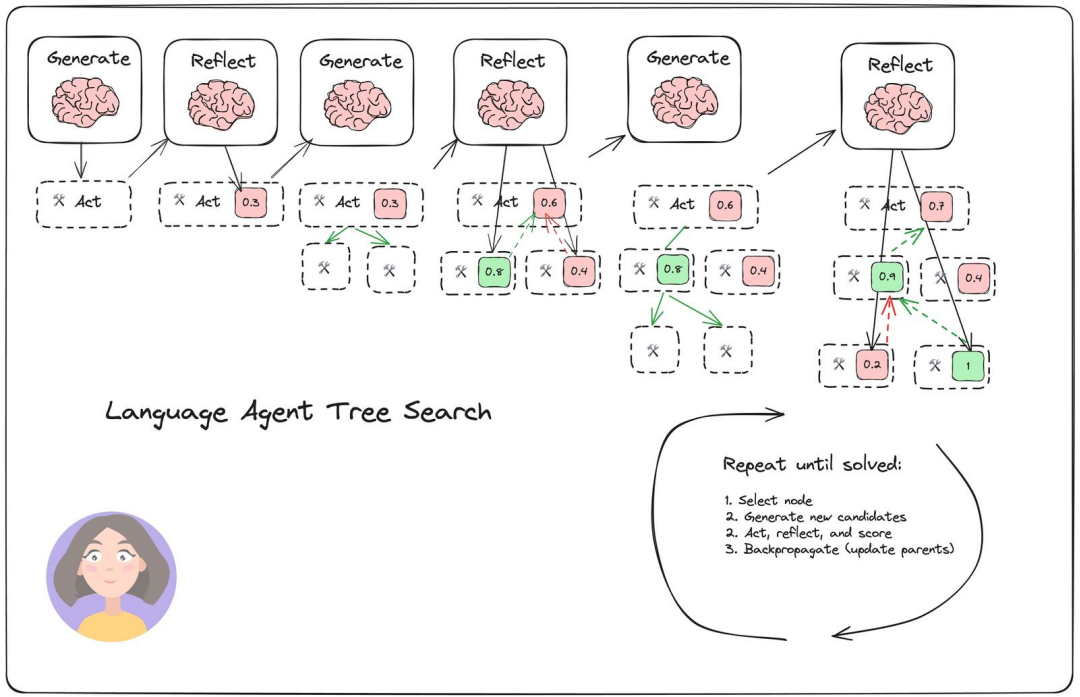
Can include simple Error Handling (Reactive): Stops, catches, or retries a specific action based on predefined rules when something goes wrong (e.g., "if API fails, retry 3 times").

[Reflexion: Language Agents with Verbal Reinforcement Learning] (2023)
(generate - evaluate - generate (revised) - evaluate - generate (revised))
“Learn from Mistakes” = generate an initial answer using ReAct, evaluate, refine or re-generate

Reflection means that the operational flow is cyclic







All of the above from https://blog.langchain.com/reflection-agents/ (with code) , also Language Agent Tree Search (LATS)
Other terms used for reflection are “on-the-fly adaptation” or “verbal reinforcement learning” from https://huggingface.co/blog/Kseniase/reflection,
Also from Raising An Agent - Episode 1 & 3 discussions on "give it errors, not bigger prompts." Historically Agentic AI was called Auto-GPT

Reflection is one of metacognition patterns (level 2)

1. Self-Reflection: Single agent reviews its own work (ReAct)
2. Producer-Critic (Evaluator) Model: Separate agent evaluates outputs

Key Benefits:

- Iteratively self-corrects for higher quality, accuracy
- Catches errors, hallucinations, missing information
- Produces more polished outputs

Trade-offs:

- Increased latency (multiple LLM calls)
- Higher computational cost (2-3x+ API calls)
- Risk of context window exhaustion

Potential next steps:

- Memory (Ch. 8): Learn from past critiques, avoid repeating errors
- Goal Setting & Monitoring (Ch. 11): grounded corrective engine & watching out for evaluator-model collusions
- Example of reflection at scale - Constitutional AI by Anthropic checking against human values and stuff)
- Self-taught Evaluators (Arxiv) to solve for costly human preference labels as base models improv by training a self-taught evaluator:
    1. Generate multiple candidate outputs for an instruction.
    2. Ask judge model to explain which is better (reasoning trace).
    3. Fine-tune that judge on its own traces; iterate.
    4. Use the judge as a reward model or quality gate for the main agent.
    5. Periodically refresh with new synthetic debates to stay ahead of model drift.

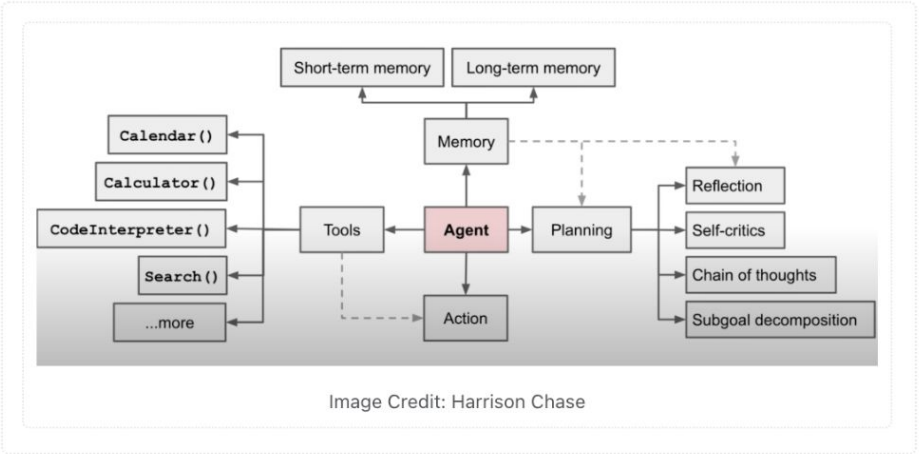


Image Credit: Harrison Chase