Agentic Systems:

Architecture, Evaluation, and Design Foundations

Dr Maria Prokofieva

Maria Aise

What we will cover

- What an AI agent is
- Core architecture
- Key reasoning patterns
- Tools use
- Evaluation

From Passive AI to Autonomous Agents

Agent:

- a complete,
- goal-oriented application that
- combines a reasoning model, actionable tools, and a governing orchestration layer to
- perform autonomous problem-solving.

Passive → Active

Autonomous Problem Solving

Complex, multi-step tasks

Non-determinism

Real-world side effects

The Problem:

Solo researcher needs to submit a paper to a conference on Cancel Culture

Deadline: 48 hours

Task: ” M*easure stock dips after brand controversies go viral on Tiktok.* "

Reality: One person doing everything

ALL WITH:
One brain
Limited time
Limited skills
High Cognitive switching costs

1. Search psychology papers (Semantic Scholar)
2. Pull market data (Yahoo Finance API)
3. Develop Event Study design
4. Run statistical analysis (R/Python)
5. Write **manuscript** using a conference-ready LaTeX template.
6. Review for rigor (peer critique)

Task Breakdown: Cancel Culture & Stock Impact Research (48hr Deadline)

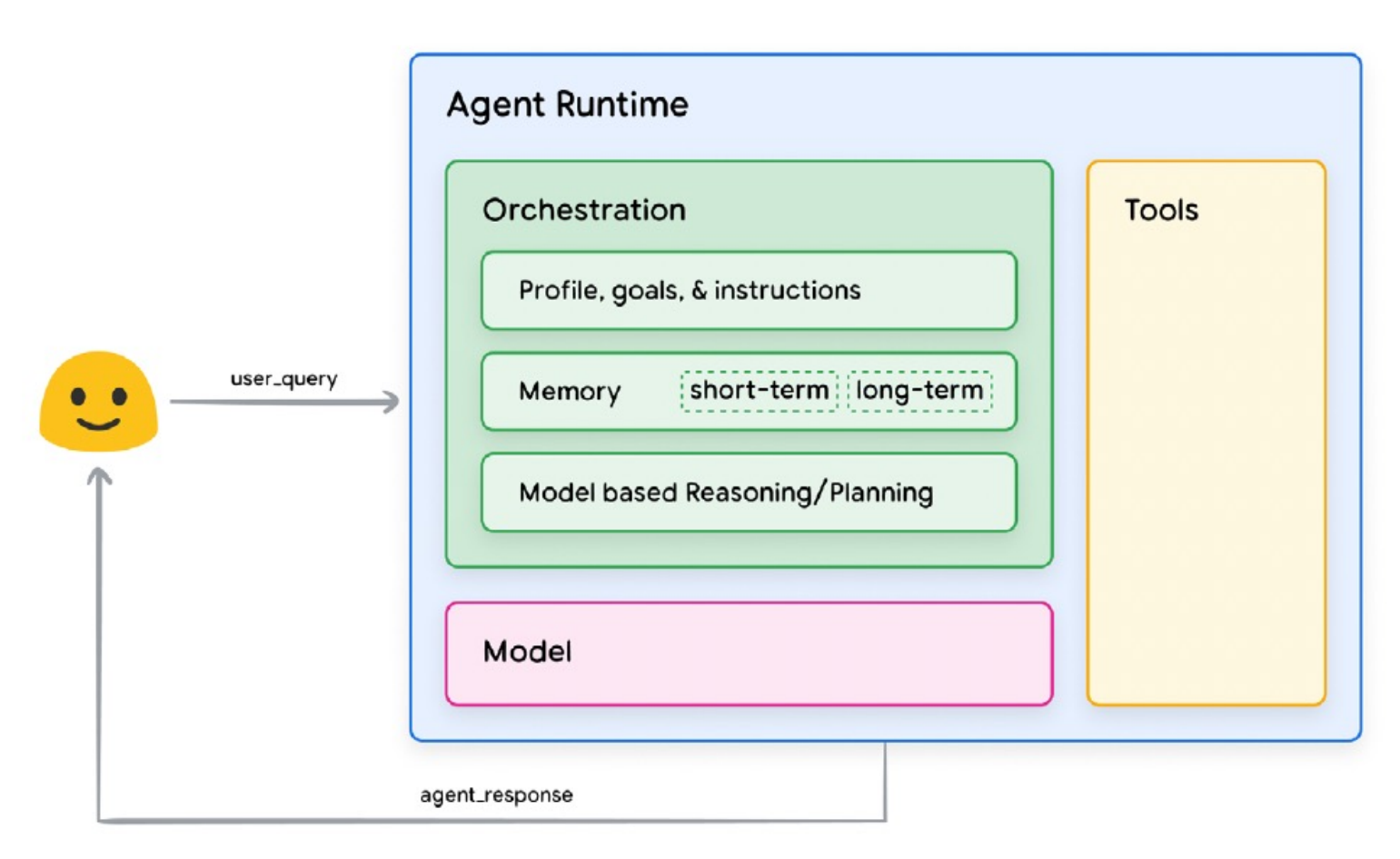
Search & Analyse	Pull & Clean	Develop	Run	Write	Peer
Search & Analyse Psychology Papers 12-16 hours	Pull & Clean Market Data 8-10 hours	Develop Event Study Methodology 6-8 hours	Run Statistical Analysis 4-6 hours	Write Manuscript 10-14 hours	Peer Review & Rigor Check 4-6 hours

Task Breakdown: Cancel Culture & Stock Impact Research

Search & Analyse	Pull & Clean	Develop	Run	Write	Peer
- Semantic Scholar API + keyword clustering (automated) - Human reviews summaries and selects 10-15 key papers - Human synthesizes concepts	- Python script with *yfinance* API, list of "events" from *Tiktok* - Human validates data quality - Human defines event windows and adjust data	- Python/R event study library - Human writes custom methodological justification	- Manual coding in R/Python - Debugging errors - Running multiple model specifications	- AI-assisted drafting of sections - Human writes original insights - Formatting with template - Final polish	- Automated statistical assumption tests - Human critical read-through

Core Architecture

- Model
- Tools
- Orchestration layer
- Runtime services (deployment)



Model: The Reasoning Core

- An agent can utilize one or **multiple language models (LMs)** of any size, ranging from small to large.
- Must be capable of following **instruction-based reasoning and logic frameworks**
- **Versatility:** models can be
  - general-purpose,
  - multimodal, or
  - fine-tuned for specialized tasks.

Model responsibilities:

- Goal interpretation
- Planning
- Tool selection
- Synthesis

Considerations:

Intelligence

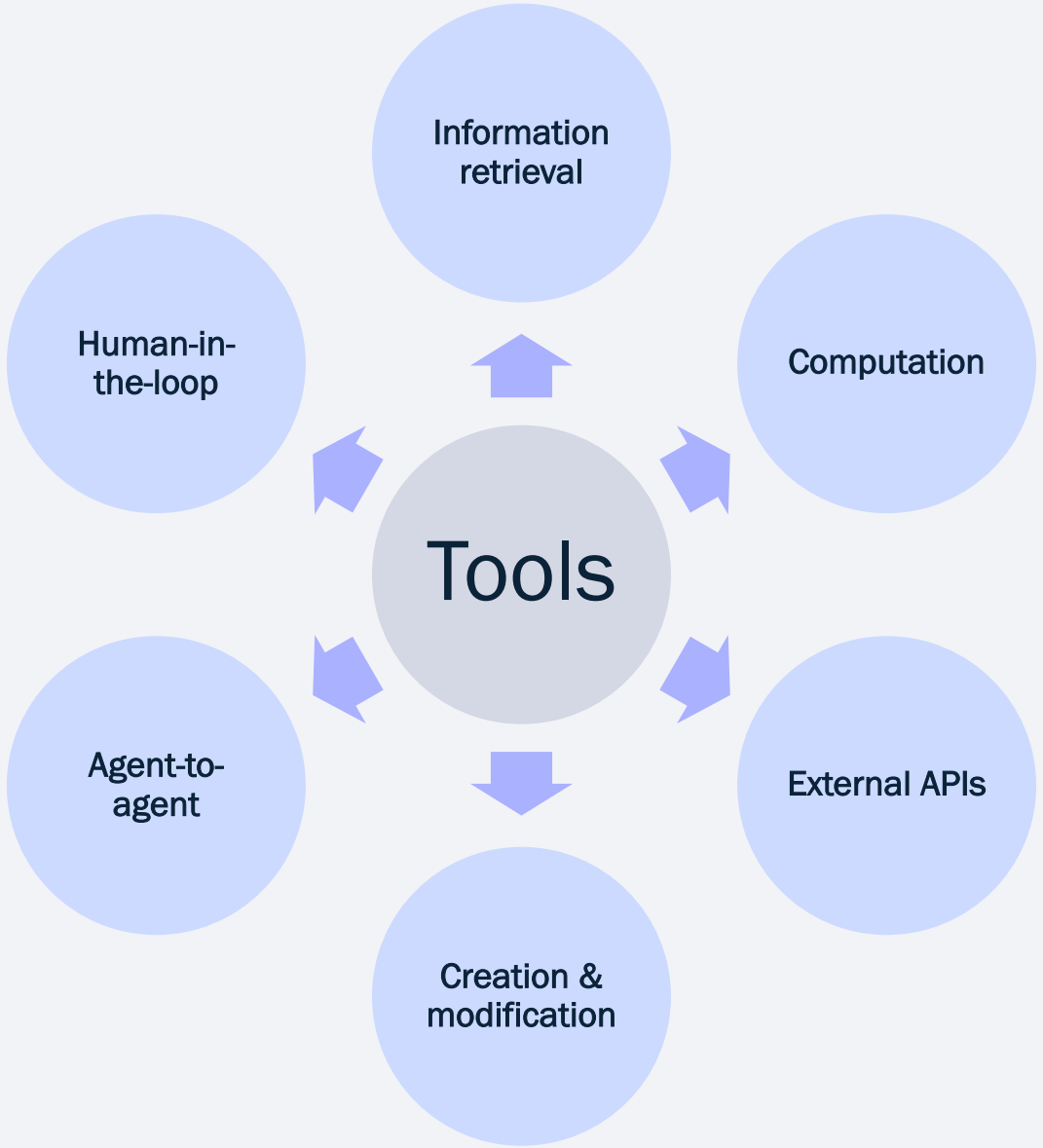
Context Window

Cost

Speed/Latency

Tools

- External capabilities that allow an autonomous agent to interact with the real world
- Agents can have 5–10 tools (simple) or 50–200+



Main Approaches to Tool Use

Function Calling / Tool Calling (Native LLM Feature)

Text-based Tool Use

Model Context Protocol (MCP)

Orchestration: memory

Memory enables agents to

- maintain context,
- learn from experience,
- avoid repetition,
- build on prior knowledge

Types of Memory

- **Short-Term Memory (STM) — Working / Conversation Memory**
  - Keeps track of the **current session/task** — recent messages, user inputs, agent thoughts, tool results, and intermediate reasoning steps.
- **Long-Term Memory (LTM) — Persistent / Cross-Session Memory**
  - Stores knowledge the agent should remember **indefinitely** or across many interactions — user preferences, past experiences, domain facts, learned procedures.

  Can be organised as:
  - Vector Stores
  - Knowledge Graphs
  - Baked-in (model's knowledge for high-confidence, stable facts)

Key Reasoning & Planning Patterns

- Chain-of-Thought (CoT)
- ReAct (Reason + Act)
- Reflection / Self-Critique / Reflexion
- Plan-and-Execute
- Tree-of-Thoughts (ToT)
- Graph-of-Thoughts (GoT)

Beyond single-pass generation
Structure thought,
Explore alternatives,
Incorporate feedback,
Simulate outcomes,
Iterating toward reliable solutions for tasks

All three are **the same cognitive loop:**
Reason → Decide → Evaluate → Update

Chain-of-Thought (CoT)

- Explicit “think step by step” prompting.
- **Strength**: Dramatically improves multi-step logic, arithmetic, common sense, and symbolic reasoning.
- **Limitation**: Linear, no branching/backtracking, prone to hallucinated intermediates.
- **Role**: Foundational building block for nearly all advanced patterns.

Loop is fully internal and implicit
- Reasoning
- Self-check (weak, implicit)
- Answer

No environment. No feedback except the model’s own text.
CoT = one-pass internal simulation of the loop
Same loop, no external grounding, no persistence.

ReAct (Reason + Act)

- Interleaved loop: Thought → Action (tool call) → Observation → repeat.
- **Strength**: Highly adaptive, grounded by real-world feedback, excellent for dynamic/exploratory tasks.
- **Limitation**: Myopic, token/latency heavy in long traces, context drift risk.
- **Role**: Most widely deployed base pattern for interactive agents.

Loop is externalized at the "Act" step
•Reason internally
•Act in the world (tool / API / search)
•Observe real output
•Reason again
ReAct = the same loop, but "Act" and "Observe" are real
Nothing new cognitively — just grounding the loop in reality.

Reflection / Self-Critique / Reflexion

- Agent critiques its own output/step, generates feedback, then refines.
- **Strength**: Major reliability gains (15–30%+ on hard tasks), acts as verbal self-improvement.
- **Limitation**: 2–3× cost/latency, risk of over-critique loops.
- **Role**: Essential quality layer added to ReAct or Plan-and-Execute in production.

Loop is externalized at the "Evaluate / Update" step
•Generate answer or trajectory
•Explicitly critique it
•Extract failure modes
•Modify future reasoning
Reflection = the same loop, but "Evaluate" is explicit and persistent
Still the same loop — now with memory or policy updates.

Tool invocation

- **Tool invocation** is the mechanism by which a language model **requests an external capability** (API, database, search, code execution) and receives the result back into its context.

Text-based Tool **Function Calling**

Text-based Tool Use

Initial Setup (Before the Loop Starts):

Define tools: A dictionary mapping names to functions (e.g., {"web_search": lambda query: search_api(query)}).

Prompt template: "You are an agent. Tools: web_search (searches the web), calculator (runs math). Format: Thought: [reasoning]. Action: [tool_name] [args]. Observation: [result]. Final Answer: [answer]."

1. Build Prompt: Combine system instructions + history + user query. Prompt: "User: What's the weather like today in Melbourne, and should I wear a jacket? Think step by step. Use tools if needed."

2. Call LLM: Send to model

LLM output (raw text):
"**Thought**: I need current Melbourne weather to decide on jacket. Location is Melbourne, AU, time is Feb X, 20XX ~10 PM AEDT, so summer evening.
Action: web_search current weather Melbourne Victoria Australia February X 20XX"

3. Parse Output:
Runtime scans text. Look for "Action:" prefix. Extract tool: "web_search" Extract args: "current weather Melbourne Victoria Australia February X 20XX" (simple string parse). Pseudocode: XXXXX

4. Execute Tool: Call the real function. Input: args from parse. **Output**: Result (e.g., "28°C, clear skies, light breeze 5 km/h"). Handle errors: If API fails, return "Error: Search timed out." Implications: Sandbox here (e.g., rate limits, auth) to prevent abuse.

Function Calling

Initial Setup (Before the Loop Starts):

Define tools: A dictionary with a list of tool definitions (name + description + JSON schema of parameters)

1. Build Prompt: Combine system instructions + history + user query. Prompt: "User: What's the weather like today in Melbourne, and should I wear a jacket? Think step by step. Use tools if needed."

2. Call the LLM with tool-calling mode enabled

3. Model decides to call a tool

Instead of writing free text, the model returns a **structured tool call** object.

4. Runtime handles the structured output (very clean)

Checks the schema, makes a call

Maria Aise

Tool Invocation

Step	Raw Text Prompting (ReAct-style)	Function Calling (structured)
Model output format	Free text with special tags / keywords	Structured JSON object (tool_calls array)
How runtime knows it's a tool	Regex / string search / custom parser	API response field tool_calls is present & typed
Parsing reliability	Medium–low (hallucinated formats common)	Very high (native, schema-validated)
Parameter extraction	Manual splitting / regex	Already a Python dict / JSON object
Error when format is wrong	ParseError → retry / fail / human escalation	Model usually refuses bad format or API rejects
Conversation history growth	Text keeps growing (Thought + Action + Observation)	Structured messages (tool call + tool response)
Debugging	Need to inspect messy text logs	Clean structured logs (tool name + exact args + result)

Tools: Search (Web / Semantic / Specialized)

- Fetch up-to-date, external, or factual information not in the model's knowledge or context.

Subtypes:

- General web search,
- semantic/vector search,
- X/Twitter search,
- news/academic.

Typical use: Current events, weather, prices, fact-checking, research background.

Key traits: Fast, medium cost, medium risk (SEO noise, outdated info, prompt injection via pages).

Common guardrails: Reflection, cross-verification, query rephrasing

Tools: APIs (External Service Integrations)

- Perform real-world actions or fetch structured data from third-party services.

Typical use:

- Calendar (create events),
- email (send/read),
- CRM (update records),
- payments (Stripe),
- maps

Key traits: Fast–medium speed, medium–high cost, **very high risk** (real money, real emails, real data changes).

Common guardrails: Scoped OAuth, human-in-the-loop for writes, rate-limit handling.

Tools: Databases

Databases (SQL / NoSQL / Vector DB Access)

Query structured or semi-structured enterprise/personal data.

Typical use:

- Business analytics,
- personal knowledge retrieval,
- vector/RAG search over documents.

Key traits: Fast–medium speed, medium cost, **very high risk** (data leaks, accidental deletes/updates).

Common guardrails: Read-only default, query validation, result pagination/summarization.

Tools comparison

Category	Frequency in agents	Risk Level	Speed	Most universal combo
Search	Very high	Medium	Fast	Almost every agent
Code Interpreter	Very high	High	Medium	Paired with search
APIs	High (enterprise)	Very High	Fast–Medium	Scoped + human gates
Databases	Medium–High	Very High	Fast–Medium	Enterprise only
File System	Medium	High	Fast	Document/code agents
Browser	Medium–Low	Medium	Slow	When JS needed

Agentic Frameworks

Framework	Best For	Style	When to Choose
LangGraph	Complex control flow, state machines	Graph-based orchestration	Production-grade agents, custom loops
CrewAI	Role-based multi-agent teams	Crew + role + task metaphor	Collaborative / division-of-labor use cases
LlamaIndex	Agents heavily using documents/RAG	Data-centric agents	Research, knowledge-heavy agents
AutoGen	Conversational multi-agent	Chat-based agents	Simulation, research prototypes
Google Agents SDK	Clean, minimal abstraction	Lightweight	Quick prototypes, Gemini ecosystem

Evaluation, Debugging & Reliability

- Agents produce **long trajectories** (multi-step reasoning + tool calls + observations), not single outputs.
- **Non-determinism** (temperature, tool variance, branching) → same input can yield very different paths.
- **Compounding errors** over long horizons → early mistake cascades → total failure.
- **Open-ended success** — no simple "right/wrong" label; goal completion is nuanced/subjective.
- **Real-world dependencies** (API uptime, data freshness, external changes) → hard to mock perfectly.
- **Cost & scale** — full trajectory evals are 5–20× more expensive than classifier batches.
- **Implication**: Under-investing here leads to agents that "work in demos" but fail reliably in production.

Unit Tests vs. End-to-End Trajectory Evaluation

Unit tests (modular, isolated):

- Test single components (prompt templates, tool parsers, one LLM call, error handling).
- Fast, cheap, repeatable, great for debugging regressions.
- Limitation: Misses interactions and compounding failures.

End-to-End (E2E) trajectory evaluation:

- Run full agent episodes on realistic tasks → measure goal completion.
- Captures real system behavior, emergent failures, long-horizon dynamics.
- Limitation: Expensive, slow, high variance → requires multiple runs per sample.

Key Metrics

- **Success rate** (% of tasks completed correctly) — primary gold-standard metric.
- **Step efficiency** (average steps / tokens / LLM calls per successful task) — measures cost & speed.
- **Cost per task** (USD or tokens) — critical for scaling and ROI.
- **Hallucination rate** (factually incorrect statements per trace) — often human- or LLM-judged.
- Secondary: tool call accuracy, parse failure rate, loop/exit condition correctness, human preference score (Elo-style).

Common Failure Modes + Mitigation Patterns

1. **Early wrong tool call / hallucinated parameters** → Mitigation: Strong tool descriptions, reflection before action, router models.
2. **Infinite loops / stuck cycles** → Mitigation: Max iterations, progress checks (e.g., "did observation advance goal?"), reflection stopping rules.
3. **Context drift / needle-in-haystack loss** → Mitigation: Rolling summarization of history, external memory (vector/graph), selective context injection.
4. **Observation misinterpretation** → Mitigation: Reflection/self-critique after tool use, structured observation parsing.
5. **Tool / API failures cascading** → Mitigation: Robust error handling (retry logic, fallback tools), graceful degradation.
6. **Overconfidence / no escalation** → Mitigation: Confidence scoring, human-in-the-loop gates for high-stakes steps.
7. **Cost / latency blowouts** → Mitigation: Fast/cheap model routing, caching repeated calls, early termination heuristics.

Practice