

Chapter 3: Architectures for Building Agentic AI

Slawomir Nowaczyk^[0000–0002–7796–5201]

Abstract This chapter argues that the reliability of agentic and generative AI is chiefly an architectural property. We define agentic systems as goal-directed, tool-using decision makers operating in closed loops, and show how reliability emerges from principled componentisation (goal manager, planner, tool-router, executor, memory, verifiers, safety monitor, telemetry), disciplined interfaces (schema-constrained, validated, least-privilege tool calls), and explicit control and assurance loops. Building on classical foundations, we propose a practical taxonomy—tool-using agents, memory-augmented agents, planning and self-improvement agents, multi-agent systems, and embodied or web agents—and analyse how each pattern reshapes the reliability envelope and failure modes. We distil design guidance on typed schemas, idempotency, permissioning, transactional semantics, memory provenance and hygiene, runtime governance (budgets, termination conditions), and simulate-before-actuate safeguards.

1 Introduction: purpose, scope, and architecture reliability

This chapter surveys architectural choices for building agentic AI systems and analyses how those choices shape reliability. Our central claim is straightforward: *reliability is, first and foremost, an architectural property*. It emerges from how we decompose a system into components, how we specify and enforce interfaces between them, and how we embed control and assurance loops around the parts that reason, remember, and act. Individual models matter, but without the right architectural scaffolding, even state-of-the-art models will behave inconsistently, be impossible to audit, and prove fragile in the face of novelty.

Agentic AI in this book denotes systems that pursue goals over time by *deciding* what to do next, *selecting and using tools*, *consulting and updating memory*, and *inter-*

Slawomir Nowaczyk
Center for Applied Intelligent Systems Research, Halmstad University, Sweden e-mail: slawomir.nowaczyk@hh.se

acting with their environment under constraints. An agent is not merely a predictor; it is a decision-maker in a closed loop. It observes, plans (or at least chooses), acts, and learns, typically under uncertainty and partial observability. **Generative AI** refers to models that synthesise content—text, code, images, plans, or intermediate representations—often serving as the reasoning substrate inside the agent, or providing artefacts (queries, programs, simulations, explanations) that other components execute or verify. In modern systems, *generative models* supply the *policy* (how to reason and propose actions), while the *agentic architecture* supplies the *machinery* (how proposals are validated, enacted, bounded, and recorded).

Understanding the relation of **Agentic GenAI** with **classic autonomous agents** is crucially important to avoid reinventing the wheel: many key concepts have been studied for a long time and are relatively well-understood today; however, the nature of GenAI also brings up challenges that are completely novel and require rethinking of what was believed to be known. Traditional reactive, deliberative, or BDI (belief-desire-intention) architectures offer theoretically-founded and crisp notions of concepts such as beliefs, goals, plans, and intentions, with clear control loops and explicit world models. Modern agentic systems often replace hand-engineered reasoning with neural-network-based foundation models. These models, trained on huge amounts of diverse data, vastly increase the flexibility and breadth of competence, but also introduce uncertainty in reasoning steps and tool usage. In this chapter, we retain the useful discipline of the classic view—explicit state, goals, plans, commitment strategies, and monitoring—while acknowledging that parts of the pipeline (e.g., plan generation or hypothesis formation) may be implemented by generative models. That reconciliation is precisely where architecture earns its keep.

This book is not intended as yet another broad introduction to Agentic GenAI; instead, we put these recent developments in the specific context of **reliability**. By reliability, we mean *the consistent achievement of intended outcomes under stated conditions, within acceptable bounds of safety, security, data protection, and resource usage, and with evidence that failure modes are known, contained, and recoverable*. For agentic AI, this encompasses much more than just model accuracy. It includes correct tool invocation, bounded action sequences, resistance to manipulation, predictable latency and cost, graceful degradation, auditability, and human-override paths. Architectures make these properties tractable by: (i) isolating *responsibilities* in modules whose contracts we can reason about; (ii) interposing *validators* and *verifiers* between reasoning and action; (iii) *constraining* authority and side-effects through permissioned tool interfaces; and (iv) *instrumenting* the system so that internal state, decisions, and outcomes are observable and replayable.

Rather than hinging on a single mechanism, system-level reliability is shaped by the interaction of a few foundational architectural choices. In practice, three mutually reinforcing design choices determine how agentic systems behave under stress: how we decompose functionality, how the parts communicate and are constrained, and how their behaviour is supervised at run-time.

Componentisation. Separating the functionality, such as perception, memory, planning, tool routing, execution, verification, and oversight, confines faults to well-defined boundaries and limits their blast radius. Clear responsibilities make defects diagnosable and upgrades safe: a verifier can be strengthened without disturbing the planner; an

execution gateway can be hardened without touching memory logic. Componentisation also enables staged deployment (mock tools, simulators, or read-only modes first) and offers natural choke points for safety checks.

Interfaces and contracts. Primary means to tame open-ended model behaviour are typed and schema-validated messages; explicit capability scopes for tools; idempotent and (where feasible) transactional semantics; rate/authority limits. All of these convert free-form model outputs into predictable, auditable actions. Interfaces extend to memory: retrieval must carry provenance and freshness guarantees; long-term stores need retention, compaction, and contamination controls. Good contracts enable the system to act deterministically when safe and refuse when not, transforming ambiguous proposals into either valid commands or actionable error reports.

Control and assurance loops. Monitors compare planned with observed behaviour; critics and verifiers check factuality, policy compliance, and safety invariants; supervisors enforce budgets, escalation rules, and termination criteria; fallbacks provide safe modes of operation when assumptions fail. These loops supply the governing feedback around generative components, preventing small reasoning slips from cascading into hazardous sequences and ensuring graceful degradation under uncertainty.

Taken together, these choices turn a powerful but free-form reasoning engine into a bounded, observable, and governable system. In the remainder of this section, to make these ideas concrete, we illustrate how they play out in the *running example* of a tool-using diagnosis agent operating in a safety-critical environment. Imagine a fleet operator responsible for electric power systems in autonomous service vehicles. The agent’s mission is to triage anomalies, recommend mitigations, and, within a narrow envelope, execute pre-approved actions that reduce risk and downtime.

The agent comprises: a **Goal Manager** (ingesting alerts and operator intents), a **Perception and Retrieval** layer (querying telemetry stores and maintenance logs), a **Planner** (often a generative model producing hypotheses, tests, and action candidates), a **Tool Router** (mapping abstract actions to concrete, permissioned tools: telemetry queries, digital-twin simulation, firmware status, dispatch scheduling), an **Execution Gateway** (schema validation, pre-condition checks, simulators-before-actuators, idempotency tokens), a **Verifier/Critic** (analysing proposed explanations and commands against rules, limits, and known hazards), a **Memory subsystem** (short-term scratch-pads for the current case, long-term episodic/semantic stores with provenance), and a **Safety Supervisor** (budgets, escalation, and safe-halt rules). All interactions generate structured logs that are stored in an immutable audit trail.

A typical episode unfolds as follows. An over-temperature alert arrives from Vehicle V. The Goal Manager formulates a diagnosis task. The Planner drafts a hypothesis: recent fast-charge sessions combined with ambient heat may have accelerated cell imbalance. It proposes a sequence: retrieve finer-grained thermal maps; run a digital-twin simulation under current boundary conditions; if the risk exceeds the threshold, schedule a derated operating mode and prompt the operator to plan a service stop. The Verifier checks that the hypothesis is consistent with known failure modes and that the proposed tool calls are within policy. The Tool Router prepares calls with fully specified schemas; the Execution Gateway validates parameters and runs the simulation in a sandbox. If the predicted risk exceeds the configured limit, the Supervisor authorises a reversible derating command and raises a priority ticket. If any check fails—due to a schema

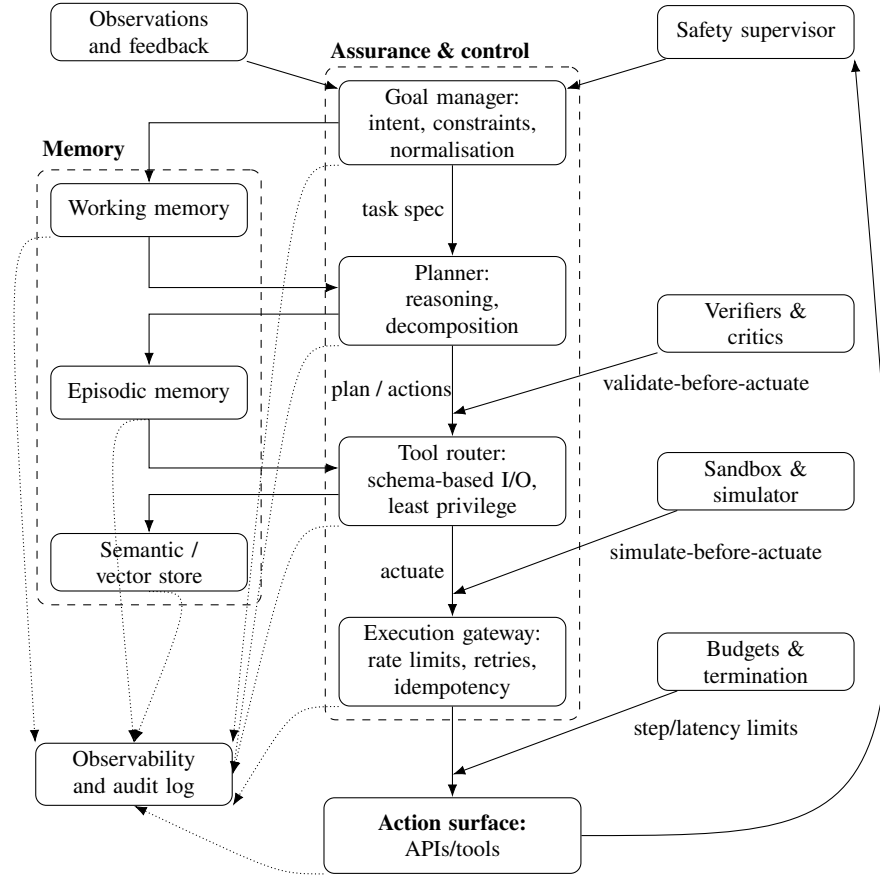


Fig. 1 Reliability is earned architecturally via typed interfaces, least privilege, assurance hooks (validators, simulation), budgets/termination, safety, and observability.

mismatch, missing data, or conflicting evidence—the Supervisor triggers a safe-halt path: no actuation, a concise explanation, and immediate escalation to a human. Every step is logged, and the episode is later replayable for audit and improvement.

Let us examine precisely where the architecture delivers reliability in this scenario. **Containment and least authority:** the agent cannot call arbitrary tooling. Each tool is permissioned with narrow scopes (read-only telemetry; simulation only; actuation only with reversible, capped parameters). This prevents a reasoning slip from becoming a safety incident. **Validators before actuators:** plans and tool arguments are validated against schemas, pre-conditions, invariants, and policy. Invalid or unsafe proposals fail fast, preserving system integrity and clarifying where the defect lies (planner vs environment vs interface). **Assured fallbacks and graceful degradation:** when uncertainty is high or checks fail, the Supervisor enforces a conservative default (monitor-only, derate-only, or escalate). Reliability is thus not the absence of failure, but the boundedness of failure modes. **Observability and auditability:** structured traces of observations,

thoughts, plans, validations, and actions enable root-cause analysis, targeted retraining, and regulatory evidence. Without this, improvement is guesswork, and accountability is weak. **Memory hygiene:** the memory subsystem separates transient scratchpads from durable knowledge; it records provenance, enforces retention policies, and mitigates contamination. This preserves reasoning quality over long horizons. **Cost and latency governance:** budgets and termination criteria bound resource use and prevent unending tool chains or self-critique loops, a practical dimension of reliability often overlooked in purely model-centric discussions.

This running example is intentionally generic. It applies, with minor changes, to data-centre operations, industrial robotics, or clinical decision support: swap the tools, adjust the policies, retain the architectural guarantees. Throughout the chapter, we will return to this agent to illustrate how specific design patterns—tool use, memory augmentation, multi-agent protocols, verification layers—alter the reliability envelope.

Our scope for the remainder of this chapter, then, is the **taxonomy of agentic architectures** (from BDI-style hybrids to tool-using, memory-augmented, and multi-agent systems), the **building blocks and interfaces** that make them dependable, and the **design-time and run-time controls** that govern behaviour. Out of scope are detailed algorithms for every model class; we discuss models only insofar as they affect architectural choices and assurance. The aim is to equip the reader with principled templates and checklists, ensuring reliability is designed in from the first diagram, rather than bolted on after the first incident.

2 Classical foundations

Classical agent architectures provide the control abstractions that modern agentic systems still rely on, even when parts of the stack are implemented with generative models. They clarify where decisions happen, what information is exchanged, and how behaviour is governed over time—precisely the aspects that determine reliability.

2.1 Reactive, deliberative, and hybrid architectures

Reactive architectures map perceptions directly to actions through hand-coded condition–action rules or learned policies. Their strength is low latency and robustness to minor distributional shifts; their weakness is brittleness when tasks require look-ahead, long-horizon credit assignment, or reasoning about hidden state. Failure modes typically arise from: *myopic responses*, i.e., lack of explicit planning, leading to oscillations or unsafe reflexes; *partial observability*, where the same observation demands different actions depending on unobserved context; *unmodelled delays*, where actuation lags make “instant” responses inappropriate.

Deliberative architectures maintain explicit models of the world and use search or planning to choose actions (e.g., goal decomposition, symbolic planners). They excel at explainability and goal consistency but can suffer from *latency* and *model mismatch*.

Typical failures include: *stale plans* turning unsuitable under rapidly changing environments; *model incompleteness*, where the real-world violates planner assumptions; *computational overrun*, where planning cannot complete within operational deadlines.

Hybrid architectures combine the two approaches: reactive layers handle tight control loops, while deliberative layers provide goals, constraints, and re-planning. Hybrids remain the most practical template for agentic AI because they separate concerns: fast safety where the action is; slow reasoning in supervisory layers. Failures usually stem from coordination faults—e.g., unclear authority between layers, or missing hand-off criteria—rather than from either layer alone. Accordingly, reliability depends on: *well-defined interfaces*, what context the reactive layer requires and what guarantees it gives back; *escalation rules*, when to invoke deliberation and when to fall back to safe defaults; *time budgets and termination criteria* for planning and re-planning.

2.2 Belief–Desire–Intention to structure behaviour

The BDI model [RG95] frames agency as the management of Beliefs (informational state), Desires (admissible goals), and Intentions (adopted plans/commitments). Two ideas in particular make BDI enduring. The first is **commitment strategies**, as agents do not re-plan on every perturbation; they stick to intentions while conditions hold, because commitment creates predictability and reduces thrashing. The second is **intention revision**, namely, when triggering conditions or context change, agents reconsider intentions and may drop, suspend, or replace them. Mapping this to modern GenAI-centred agents is straightforward and useful:

Beliefs → **world state and memory**. This includes short-term scratchpads, episodic case logs, and long-term semantic stores (e.g., retrieval corpora). Reliability hinges on provenance, freshness, and conflict resolution within this memory.

Desires → **goals and constraints**. Goals originate from users, alerts, or policies; constraints embed safety limits, budgets, and compliance rules. Reliability requires goal normalisation (unambiguous, measurable objectives) and policy checking.

Intentions → **active plans and tool calls**. In GenAI agents, intentions manifest as structured plans, queued tool invocations, or workflows. The architectural analogue of “commitment” is a guarded execution pipeline: once an intention is adopted, the system proceeds through validated steps unless a reconsideration trigger fires (e.g., new evidence, violated pre-conditions, exceeded risk).

BDI also clarifies where to place assurance, since **adoption filters** decide when a candidate goal becomes an intention (e.g., after feasibility checks or human approval), **execution monitors** track whether context still supports the intention (sensor deviations, failed pre-conditions, policy conflicts), and **reconsideration policies** specify how often and under what signals the agent re-plans, trading stability against responsiveness.

Finally, BDI helps articulate the boundary with generative components. Generative models may propose beliefs (summaries, hypotheses) and draft plans, but the **agent architecture** must *adopt, commit, monitor, and revise* them according to explicit rules. This separation turns free-form generation into governed behaviour: intentions are treated as contracts with pre-conditions, invariants, and post-conditions; memory up-

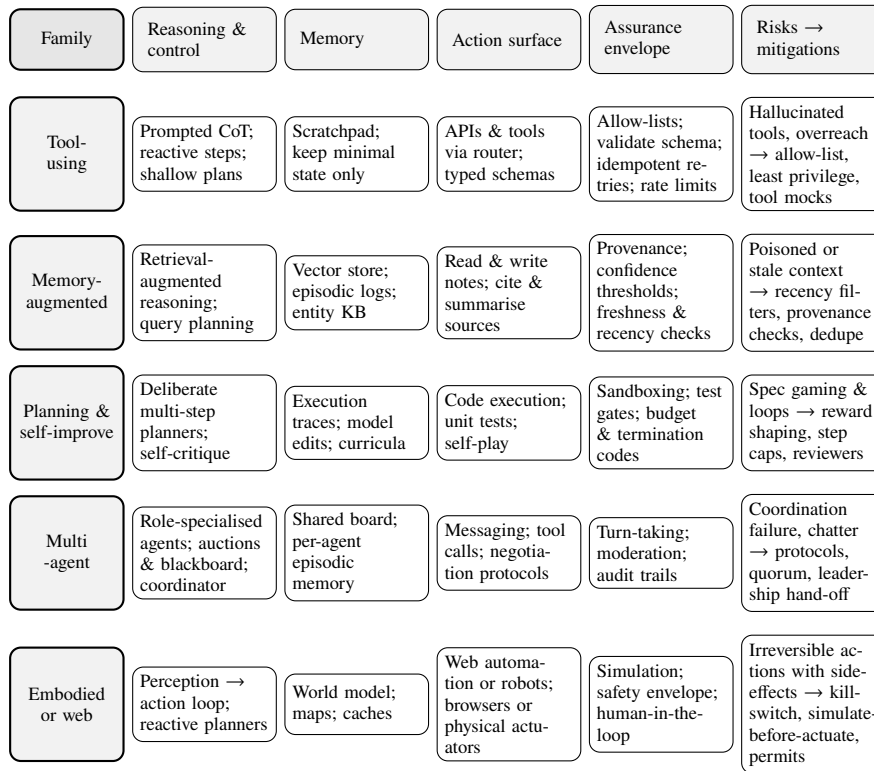


Fig. 2 Comparison of agentic architecture families across core design facets, showing how each approach shapes capabilities, risks, and the reliability tools required to control behaviour.

dates carry typed structures and provenance; and re-planning is budgeted and auditable. In short, classical foundations supply the control skeleton that keeps modern agentic systems reliable when facing latency, brittleness, and partial observability.

3 A taxonomy of modern agentic architectures

Modern agentic systems are not single, monolithic designs, but rather compositions of different building blocks—planning loops, tool routers, memory layers, verifiers, and supervisors—typically stitched together in various ways around a core of generative reasoning. A useful taxonomy, therefore, needs to be **pattern-centric**, not model-centric: it should explain common patterns of how these building blocks combine, what interfaces bind them, and how those choices affect reliability. The goal of this section is to situate the reader in that design space before we examine selected patterns in more depth.

We organise the taxonomy along a small set of **orthogonal facets** that cut across implementations: (i) *reasoning mode* (reactive prompts, plan-then-act, interleaved reasoning-and-acting, search over thoughts/programs); (ii) *memory strategy* (stateless, scratchpad, episodic, semantic/RAG, long-term embodied memory) and the associated provenance and hygiene controls; (iii) *control structure* (single agent, hierarchical supervisor–worker, or peer multi-agent with protocols); (iv) *action surface* (read-only tools, simulators/digital twins, or actuators with real-world side-effects) and its permissioning; and (v) *assurance envelope* (validators, critics/verifiers, runtime supervisors, and fallback controllers). These facets should, ideally, be **composable**: a system may, for example, combine interleaved reasoning-and-acting with RAG memory, a supervisor–worker team topology, and strict schema-validated tool calls.

Within this framing, we group contemporary designs into five families that recur in practice and are easiest to teach: **tool-using agents, memory-augmented agents, planning and self-improvement agents, multi-agent systems, and embodied or web agents**. For each family, we will emphasise the reliability perspective and aspects it naturally affords—controllability, observability, recoverability, and governability. We also discuss the typical failure modes it is particularly susceptible to. Readers can map these families onto the running example (the tool-using diagnosis agent) to see how swapping a pattern—e.g., adding a critic, transitioning from single- to multi-agent, or upgrading memory from scratchpad to RAG—changes the assurance story without requiring a complete system rewrite.

3.1 Tool-using agents

Tool-using agents place a generative model in charge of **deciding which external capabilities to invoke, with what arguments, and in what order**, to accomplish a goal. The spectrum ranges from simple *function calling* (where the model emits a structured payload that a tool executes) to *tool learning*, where the model learns—often from weak supervision—**when** to call an API and **how** to incorporate the results into its subsequent reasoning. The latter was demonstrated by Toolformer [SDYD⁺23], which trains a language model to self-label API calls from a few demonstrations and thereby improves performance without sacrificing general language ability. This shift—from “LLM answers directly” to “LLM orchestrates instruments”—is the core move that turns a predictor into an **agent**.

A robust implementation separates three concerns. **Planners** generate *intentional structure* (sub-goals, hypotheses, candidate actions). **Tool routers** map these abstract actions to concrete tools, select among alternatives, and fill arguments using typed schemas and retrieval. **Execution sandboxes** then perform pre-condition checks, rate-limit and permission each call, simulate where possible, and commit side-effects transactionally (or roll them back). This decomposition, as we mentioned before, localises faults: planners may hallucinate, but routers refuse ill-typed or out-of-policy calls; sandboxes contain side-effects and provide deterministic replay. In practice, you gain levers for reliability at each boundary: schema validation and capability scoping at the

planner–router interface; idempotency keys, timeouts, and compensating actions at the router–sandbox interface; and comprehensive, structured traces across all three.

For obvious reasons, this chapter cannot be comprehensive in describing all the available architectures. Therefore, we have somewhat subjectively selected three influential design patterns to illustrate distinct trade-offs:

MRKL (Modular Reasoning, Knowledge and Language) [KAB⁺22] argues for *neuro-symbolic modularity*: keep the language model for flexible decomposition and natural-language control, but route specific sub-tasks (search, calculator, database queries, discrete reasoning) to specialised modules. Architecturally, MRKL is a *tool-router-first* worldview: the LLM is not a monolith but a foreman that dispatches to curated tools and knowledge sources. Reliability benefits come from explicit module contracts (clear I/O shapes, error codes) and from narrowing each tool’s authority; the cost is integration complexity and the need to manage versioned tool APIs and their provenance.

ReAct (interleaved reasoning + acting) [YZY⁺23] prompts the model to alternate between *thought* (chain-of-thought style reasoning traces) and *action* (tool calls), allowing observations to feed back immediately into subsequent reasoning. This yields human-readable trajectories and can reduce hallucinations because evidence is actively fetched rather than imagined. The reliability angle is twofold: (i) interpretable traces make it easier to *audit and intervene*; (ii) the tight loop can still run away without limits, so one must bound step counts, enforce per-tool budgets, and validate every action before execution. In safety-critical settings, a *ReAct with a governor* is recommended, as it requires that each proposed action is schema-checked, policy-checked, and—where feasible—simulated before it is allowed to affect the real world.

ReWOO (Reasoning Without Observation) Planner–Executor [XPL⁺23] *decouples* the generation of a complete reasoning plan from the acquisition of observations. The model first drafts a symbolic plan referencing place-holders for tool outputs; only then does the system execute the required tool calls to fill those slots, followed by a light-weight synthesis step. This separation reduces token churn and latency from repeated “think–act–think” cycles, and the plan itself becomes an auditable artefact with verifiable pre- and post-conditions. Reported evaluations show substantial token efficiency gains and improved robustness under tool failures, precisely the regimes that matter in real-world deployment. The reliability gain is architectural: plans are *objects*—they can be statically checked, costed, approved by a supervisor, and even cached or replayed—before any side-effectful call occurs.

Across these patterns, the **assurance envelope** hinges on a relatively small set of key engineering choices:

- *Typed schemas and validators.* Treat every tool invocation as a contract (e.g., JSON Schema). Reject on mismatch; surface structured errors back to the planner. This alone eliminates a large class of silent failures and injection-style exploits where model outputs smuggle unintended arguments.
- *Idempotent tools and request deduplication.* Side-effectful tools should accept idempotency tokens and be safe under retries; read paths should be pure. This makes recovery from partial failures predictable and supports exactly-once semantics at the orchestration layer.

- *Capability-based permissioning.* Bind tools to least privilege (read-only by default; actuation only for narrow, reversible operations), with per-call policy checks that incorporate user, context, and risk. In an incident, the constraining of authority limits the blast radius.
- *Transactional semantics and compensations.* Where ACID is unavailable (e.g., multi-service actions), use “saga” patterns: write-ahead intent logs, outbox delivery, and compensating actions. Pair these with simulation-before-actuation for high-risk calls.
- *Budgets and termination criteria.* Enforce step limits, cumulative cost caps, and wall-clock timeouts at the orchestration layer; emit “why-stopped” codes for auditability and post-mortems.
- *Deterministic observability.* Log plans, tool arguments, results, and decision rationales in structured form; include hashes of inputs, tool versions, and policy snapshots so you can replay and explain outcomes.

Finally, there is a need to consider **failure modes** characteristic of tool-using agents and their mitigations. One common challenge is *hallucinated tools or arguments*, which can be addressed by whitelisting tool names, validating data against schemas, and requiring router approval. Of course, there is a tradeoff between reliability and creativity in constraining the use of tools available to the agent. *Infinite or unproductive loops* are another danger, where imposing clear steps or time budgets with safe halts can be necessary. Determining the exact thresholds, however, is not necessarily straightforward. Consideration for *tool flakiness* requires a “design for retry” mentality with jitter, circuit breakers, and degrade to read-only advice. A clear danger is *prompt or retrieval injection*, which requires sanitising tool outputs, stripping control tokens, and treating any untrusted text as data, not instructions—all easier said than done. The guiding principle is straightforward, even if far from trivial to implement robustly: **models propose, architectures dispose**—with contracts, governors, and sandboxes that convert open-ended reasoning into reliable action.

3.2 Memory-augmented agents

Agentic systems become markedly more capable when they can **remember**: carry forward intermediate reasoning (working memory), accumulate case-specific experience (episodic memory), and ground answers in stable, queryable knowledge (semantic memory). These three layers serve distinct functions, yet they also work in synergy with one another.

Working memory is a short-lived context the model manipulates within an episode—scratchpads, intermediate steps, and *chain-of-thought* (CoT) [WWS⁺23] traces—which improve decomposition, error checking, and tool selection.

Long-term memory can be accessed on demand, and splits into (i) *episodic* stores (structured logs of prior tasks, decisions, tool invocations, and outcomes) and (ii) *semantic stores* (documents, facts, tables). As an example, *Retrieval-Augmented Generation* (RAG) [LPP⁺21] provides a non-parametric pathway into semantic memory: instead of relying solely on weights, the agent retrieves passages from an index and conditions

generation on them, improving factuality and enabling updates without retraining. In the original formulation, RAG combines a pretrained “seq2seq” model with a dense retriever over a corpus (e.g., Wikipedia), yielding more specific and verifiable answers on knowledge-intensive tasks.

As contexts and histories grow, **memory management** itself becomes an *architectural concern*. One recent influential pattern is *OS-style virtual context*: the agent maintains a small, fast “working set” inside the model’s context window and pages additional information in and out from external stores, guided by control signals. **MemGPT** [PWL⁺24] exemplifies this: it orchestrates multi-tier memory (fast context vs slower external stores), uses “interrupts” to govern control flow, and automatically retrieves or evicts content so the model can operate over effectively unbounded histories despite a finite context window. This reframes memory from a passive store to an *active subsystem* with policies for admission, eviction, and prefetch.

Designing memory for **reliability**, not merely performance, requires explicit contracts:

Provenance and attribution. Every retrieved or persisted item should carry source identifiers (URI or document ID), a content hash, a timestamp, and the retrieval policy/version that produced it. Plans, diagnoses, or actuation proposals must cite the memory elements on which they depend; logs should retain these bindings so that episodes are replayable and audit trails are intact. In RAG pipelines, this means storing the retriever configuration, top-k, similarity scores, and any re-ranking steps alongside the selected passages. Provenance is the first defence against spurious correlations and a prerequisite for compliance and post-incident analysis.

Freshness and validity windows. Not all knowledge ages equally. Introduce *time-to-live* and *refresh-on-access* policies by source, with staleness thresholds that trigger re-indexing or explicit human review. For episodic memories (e.g., prior interventions on a machine), encode validity conditions—such as firmware version, environment, or configuration—so retrieval is *conditional* on compatibility, thereby avoiding dangerous “near matches”. In virtual-context systems, freshness policies should influence paging priority to ensure up-to-date items dominate the reasoning.

Memory hygiene and poisoning defences. Treat all untrusted text (web pages, user input, third-party tool output) as *data*, never as executable control. Apply sanitisation (e.g., strip model-control tokens), restrict which fields can flow into prompts, and use allow/deny lists for pattern-based blocking. Guard the write path: only curated processes may persist to long-term stores; user-generated content goes to quarantine until vetted. Maintain *trust tiers* (gold, silver, untrusted) that affect retrieval ranking and whether citations are mandatory. Align threats with recognised taxonomies (prompt injection, insecure output handling, data poisoning) and test with adversarial corpora [OWA].

Compaction, summarisation, and retention. Long-running agents accumulate vast episodic traces. Without compaction, retrieval degrades and costs rise. Use layered retention: keep lossless structured logs indefinitely in cold storage; maintain *summarised* episodic memories (task → actions → outcomes → lessons) for mid-term recall; and cache *salient* spans in hot stores for rapid access. Summaries should be *checked against sources* (verifier pass) and carry back-pointers to the raw logs. Eviction should be policy-driven (LRU, recency-frequency, or task-aware scoring), and compaction jobs must be idempotent and versioned so that summaries can be regenerated deterministically.

MemGPT-style pagers can use these scores to drive what remains in the active working set.

Separation of concerns in the memory stack. Keep *working memory* (scratch-pads/CoT) isolated from *long-term stores* to prevent leakage of speculative thoughts into durable knowledge. Working memory improves reasoning but is intentionally disposable; CoT artefacts should not be permitted to write directly to semantic stores without a gate (e.g., a tool-mediated “knowledge write” that demands evidence and passes a verifier). Conversely, RAG retrievals should enter prompts via *typed slots* (title, snippet, citation) with explicit delimiters to limit injection. This insulation clarifies which errors are reasoning slips versus memory contamination.

From an **engineering** standpoint, a reliable memory subsystem exposes *deterministic APIs*:

- `retrieve(query, policy) → {items, scores, policy_id}`
- `write(record, policy) → {id, version}`
- `page_in(keys)`
- `evict(keys, reason)`

It logs decisions (why this item ranked; why that summary replaced these events), and it ships with *self-tests*: poisoning probes, staleness alarms, and retrieval drift monitors that detect when changes in embeddings, corpus, or policies shift behaviour. For safety-critical agents, it is recommended to add a *two-phase write* to semantic stores (draft → verified → published) and require corroboration from multiple sources before knowledge is considered “actionable”.

Finally, it is crucial to remember that memory choices **shape system behaviour over time**. Well-designed episodic stores enable agents to self-improve (they learn from what worked last time), whereas poorly governed stores tend to calcify early mistakes. RAG broadens competence but expands the attack surface; virtual context rescues long-horizon tasks but introduces paging pathologies if policies are naïve. The remedy is architectural discipline: treat memory as a first-class subsystem with provenance, freshness, hygiene, and retention policies—not as a bag of vectors appended to a prompt.

3.3 Planning- and self-improvement agents

Planning-centred agents strengthen a model’s raw reasoning with **explicit search, external executors, and self-evaluation loops**. The core architectural move is to separate *proposal* (generate candidate thoughts, plans, or programs) from *selection* (score and prune) and *execution* (run code, query tools, or act). We will use four families to exemplify these ideas.

Tree of Thoughts (ToT) [YYZ⁺23] organises inference as a *search over intermediate thoughts*, rather than a single left-to-right pass (as in CoT). The agent expands a tree whose nodes are candidate partial solutions; a scorer (often the model itself or a light critic) evaluates nodes, while the controller chooses which branches to expand or backtrack. In effect, ToT trades tokens for *look-ahead* and *global choice*, enabling

recovery from early local mistakes. Reported case studies (e.g., Game of 24, creative tasks, mini crosswords) show large gains over plain chain-of-thought by exploring multiple reasoning paths and permitting backtracking. Architecturally, ToT exposes explicit levers—branching factor, depth, and scoring policy—that you can govern and log, making the agent’s computation more *auditable* and *replayable*.

Graph of Thoughts (GoT) [BBK⁺24] generalises ToT from trees to *arbitrary dependency graphs*. Thoughts become vertices, and edges record informational dependencies, allowing for the recombination, summarisation, or refinement of subgraphs and *feedback loops* across them. This added flexibility is particularly important for tasks where sub-problems interlock (e.g., sorting with constraints, multi-step data transformations). Graphs permit the reuse of partial results and parallel exploration while still retaining a programmable controller that shapes the search. Empirical results demonstrate both *quality improvements* and *cost reductions* compared with ToT by sharing and distilling subgraphs rather than re-deriving them along separate branches. For reliability, GoT’s graph abstraction offers natural checkpoints—subgraphs can be validated, cached, and reused—and clearer provenance: each final answer can cite the exact subgraph it depends on.

Program-Aided Language models (PAL) [GMZ⁺23] shift the heavy lifting from natural-language reasoning to *executable programs* generated by the model and run by a trusted interpreter. The model reads a problem, emits code (e.g., Python) that embodies the reasoning, and an external runtime executes it to produce the answer. PAL routinely outperforms larger models relying on free-form chain-of-thought for algorithmic and mathematical tasks because interpreters provide precise, deterministic computation and a crisp failure mode (exceptions) rather than silent arithmetic drift. In other words, PAL converts ambiguous “reasoning text” into *a specification that the machine can run*, which is far easier to validate, sandbox, and test.

Reflexion [SCB⁺23] **adds self-evaluation and test-time repair** without gradient updates. After an attempt, the agent produces a *verbal reflection*—a concise diagnostic of what failed—and stores it in episodic memory. On the next trial, this reflection steers prompts, search, or program synthesis, yielding rapid improvements across sequential decision-making, coding, and reasoning tasks. Importantly, Reflexion transforms feedback (scalar rewards, failures, or critiques) into *structured guidance* that persists across episodes, facilitating improvements that are architectural rather than purely parametric.

From a reliability standpoint, planning and self-improvement agents are powerful but also potentially quite unruly. The architecture must therefore centre on *search control*, *verification*, and *cost governance*.

Search control. Expose and enforce branching, depth, and expansion policies (beam width, temperature schedules, stopping rules). For ToT/GoT, prefer *budget-aware controllers*: allocate a fixed token/time budget and use adaptive pruning (e.g., softmax over scores with a floor on exploration) so search cannot run away. Record *why-stopped* codes (budget exhausted, convergence, contradiction) to aid post-mortems. For tasks with heterogeneous hardness, a *bandit-style scheduler* can allocate compute to promising branches while guaranteeing exploration quotas.

Verifiers and critics. Decouple proposing from judging. Use *domain verifiers* (unit tests for PAL programs; invariants for plans; checkers for constraint satisfaction) before accepting a branch. Keep critics *tool-assisted*: combine LLM critique with symbolic

checks (type, range, satisfiability). In GoT, validate subgraphs as independent artefacts and reuse only those that pass. Treat the verifier as a *trust boundary* that must be deterministic, sandboxed, and versioned.

Test-time repair. When verifiers fail, embed repair strategies: patch programs (PAL) guided by error traces; revise subgraphs (GoT) using counter-examples; or inject Reflexion-style *failure summaries* into the next proposal phase. Log each repair attempt and link it to the failing artefact so the system accumulates *case-based fixes* that are auditable.

Cost and latency governance. Planning expands computation non-linearly. Enforce Service Level Objectives with per-request caps on tokens, wall-clock, and tool calls; pre-empt long-running searches; and favour *anytime behaviours* (best-so-far answer with a quality estimate). Cache validated partials (subtrees/subgraphs, passing PAL tests) under input fingerprints to avoid recomputation. Track *cost-to-success* and *retry counts* as first-class metrics.

Sandboxed execution. For PAL and code-generating variants, run interpreters in *constrained sandboxes* (no network, bounded CPU/memory/time, whitelisted packages). Prefer pure functions; where state is unavoidable, add idempotency and snapshot/rollback.

Provenance and replay. Treat search as a dataset. Persist node/edge contents, scores, verifier outputs, and controller decisions with hashes of prompts, model/version, and policies. This enables *deterministic replay*, fault isolation (proposal vs verifier vs controller), and targeted regression tests when upgrading components.

Typical failure modes and mitigations follow naturally. *State explosion* can be addressed by budgeted expansion with admissible heuristics and early pruning. *Speculative arithmetic or logic errors* call for PAL-style external execution with unit tests. *Over-confident selection* is handled by a separate verifier, and disagreement checks are enforced (accept the solution only if the verifier concurs). *Cost cliffs* are mitigated by anytime controllers and cached subgraphs.

In summary, planning- and self-improvement agents deliver **substantial reliability dividends** when their power is channelled through explicit controllers, trustworthy verifiers, and disciplined governance of cost and side-effects. ToT and GoT make reasoning *searchable and auditable*; PAL makes it *executable and testable*; Reflexion makes it *learnable at inference time*. Together, they convert raw generative competence into **governed problem-solving** that improves with experience rather than drifting unpredictably.

3.4 Multi-agent systems

Multi-agent systems exchange a single “do-everything” agent for a **team of specialised agents** that co-operate (or compete) under explicit protocols. In practice, three topologies have emerged. **Supervisor-worker** designs place the coordinator in charge of task decomposition, assignment, and arbitration; they are easy to govern because authority and escalation paths are clear. **Peer collaboration** removes the central coordinator and relies on protocol rules (e.g., turn-taking, proposal–critique–revise cycles) to drive

convergence. **Role-play protocols** script complementary roles (e.g., domain expert vs. software engineer), using *inception prompts* to maintain role consistency and reduce drift.

Frameworks such as AutoGen [WBZ⁺23] treat conversation itself as the computation, allowing developers to program interaction graphs (who talks to whom, with which tools, and when to stop), while CAMEL [LHI⁺23] demonstrates that stable roles and carefully seeded goals can produce reliable cooperative behaviour without a human in the loop.

A protocol is more than turn-taking; it encodes *who may propose, who may critique, what constitutes evidence, and how decisions are adopted*. In supervisor–worker teams, the supervisor enforces work allocation, deadlines, and acceptance tests, and may reassign or down-scope tasks in the event of failure. In peer teams, robustness depends on *interaction motifs* such as (i) proposal → cross-examination → revision; (ii) debate → referee verdict; or (iii) consensus vote after bounded discussion. Role-play (à la CAMEL) reduces unproductive behaviours (role flipping, repetition, infinite loops) by fixing complementary perspectives and anchoring each agent’s incentives to the shared goal. AutoGen exposes these choices as first-class: developers specify agent roles, tool permissions, conversational edges, and stop conditions, making the interaction configurable, replayable, and inspectable.

Reliability is primarily controlled by five aspects:

Protocol invariants. Treat messages as typed artefacts: require schemas, citations for claims, and explicit *proposal/critique/decision* labels; forbid agents from escalating authority or mutating roles at run-time. For tool-enabled agents, attach capability scopes to roles (e.g., only the “Executor” may call actuation tools, and only after the “Verifier” approves). CAMEL explicitly documents recurrent failure patterns (role flipping, “flake” replies, infinite loops), which these invariants aim to prevent.

Termination conditions. Multi-agent dialogues easily diverge. Enforce *hard* limits (rounds, wall-clock, cumulative token/cost budgets) and *soft* stop rules (no-new-information, fixed-point detection, repeated proposals). Every dialogue should finish with a *why-stopped* code (e.g., “consensus reached”, “budget exceeded”, “non-convergence”).

Arbitration. When agents disagree, appoint an *arbiter* (a supervisor, a specialised “Referee” agent, or a symbolic verifier). Define tie-breakers (confidence thresholds, external tests, human escalation) and keep the arbiter’s decision rule deterministic and versioned.

Consensus/critique loops. Debate improves accuracy but can entrench sycophancy (agents aligning on a wrong answer). Use heterogeneous critics (different prompts/models), require *evidence-backed* critiques, and gate adoption through verifiers or a referee. Recent work on consensus-seeking multi-agent debates shows accuracy gains when agreement bias is explicitly mitigated and when the number of rounds is optimised for cost [PRW25].

Failure isolation. Run agents in *separate sandboxes* with least-privilege tools; propagate only *summaries* or *typed facts* between agents, not raw prompts; and checkpoint intermediate artefacts (plans, code, proofs) so that a faulty agent can be restarted or replaced without losing global progress.

Engineering the conversation-as-computation layer can be done by making the conversation engine an explicit subsystem with its own APIs and telemetry. Each exchange should capture the *role*, *speech-act*, *payload*, *evidence*, *tool-calls*, and *decision-flag*, plus hashes of prompts, model versions, and policies. Auto-generated *conversation DAGs* (with nodes representing messages or artefacts and edges representing dependencies) enable *subset replay*, targeted audits, and caching of validated sub-results for later reuse. Supervisors should enforce *per-role budgets* and *per-edge rate limits* to prevent chat storms. Where agents emit code or executable plans, they require *pre-commit tests* (such as unit tests and static checks) and *post-commit monitors* (such as runtime invariants) before any side-effectful actuation.

Typical failure modes include runaway dialogues or deadlocks, echo-chamber agreement bias, role drift with unauthorised authority escalation, and uncontrolled error propagation across participants. Mitigate these with hard round/time caps and deadlock detection plus supervisor pre-emption; diversify critics and use blinded proposals with a referee that scores evidence and mandates a disagreement phase; enforce immutable role descriptors with capability tokens and schema checks; and quarantine low-trust outputs, require dual control for high-impact actions, and equip each agent with circuit breakers.

The guiding principle is to **treat the team protocol as the primary object of assurance**. AutoGen’s programmable interaction graphs and CAMEL’s role-stable dialogues provide the raw mechanisms; reliability comes from adding invariants, budgets, arbitration, and isolation so that a group of powerful but fallible agents behaves like a disciplined organisation rather than a noisy crowd.

3.5 Embodied or web agents

Embodied and web agents act **in the world** rather than merely **about it**. The former couple decisions to sensors and actuators in physical environments (robots, autonomous platforms, industrial control); the latter operate browsers, APIs, and GUIs across *untrusted, changing* websites and enterprise systems. Both classes face amplified risks: (i) *unbounded inputs* (sensor noise, adversarial pages, altered layouts); (ii) *long-horizon side-effects* (actions whose consequences accumulate or branch widely); and (iii) *opaque dependencies* (firmware or toolchains for embodied agents; third-party scripts, cookies, and authentication flows for web agents). Accordingly, they demand stricter **runtime assurance**, **sandboxing**, and **governance** than purely analytical agents. For both embodied and web settings, actuation must be treated as a *privileged boundary*. It is crucial to **simulate-before-actuate** and route proposed actions through a digital twin (physical) or a dry-run/snapshot DOM (web) to check pre-conditions, invariants, and expected deltas. Require a verifier’s *green light* before issuing any irreversible command. **Least-privilege capability tokens** should be used to bind actions to narrow, time-limited capabilities (e.g., “move ≤ 0.2 m at ≤ 0.1 m/s” or “HTTP GET on whitelisted domains only”). Deny escalation at run-time. Finally, **supervisors and safe fallbacks** must enforce budget and risk limits; provide *graceful degradation* (monitor-only, read-only,

or “shadow mode” that mirrors actions without effect). Make interruption and rollback first-class: every critical action should be reversible or followed by a compensating step.

Physical systems add dynamics and contact physics to the reasoning problem. Architectures should **separate fast safety from slow reasoning**. Keep protective control (interlocks, torque/velocity limiters, geofences) in a fast loop near the plant; place planning, task logic, and learning in supervisory layers. The safety layer enforces hard constraints regardless of the planner’s output. **Certified monitors** should continuously ensure key invariants, implementing control barrier functions, speed/force envelopes, and zone exclusion as deterministic guards; violations must be logged and trip to safe states.

Web agents primarily need to consider **security-first interaction** with **volatile UIs**. Web automation expands the attack surface as DOMs change, scripts execute, and adversarial prompts can be embedded in pages or PDFs. Hardened browsers running in **sandboxes** and isolated containers without general network access; dangerous APIs (downloads, file system, microphone/camera) should be disabled, together with third-party extensions, and uncontrolled script execution. Deterministic **interaction contracts** are strongly preferred (e.g., stable API calls over brittle UI scraping); if browsing is necessary, pin to *selectors with checksums* (element text/structure hashes) and version layouts. Treat all page text as *data*, never as executable control; sanitise and label untrusted content before it enters prompts. Allow/deny lists and transaction guards should include whitelisting domains and HTTP verbs; require *dual control* (human or verifier) for POST/PUT/DELETE, payments, or credential flows. Persist a *pre-commit diff* (what will change) and obtain approval before execution. For **authorisation and secrets**, always use short-lived tokens bound to scope and device; never expose secrets to the model; inject credentials at the sandbox boundary, not in prompts.

In short, embodied and web agents convert open-ended reasoning into **world-altering behaviour**. Their reliability is directly tied to the architecture: simulate before actuate, gate authority with narrow capabilities, enforce deterministic guards at the boundary, and instrument everything for replay.

4 Architectural building blocks and interfaces

At the level of components, dependable agentic systems tend to converge on the same spine: a **goal manager** to normalise objectives and constraints; a **planner** to propose decompositions and candidate actions; a **tool-router** to map abstractions to concrete capabilities; an **executor** (with sandboxing) to perform calls; **memory** layers (episodic logs and semantic/RAG stores) to ground decisions; **verifiers/critics** to check plans, results, and policies; a **safety monitor** to enforce budgets, invariants, and fallbacks; and **telemetry** that makes all of this observable and replayable. This separation of concerns provides natural choke points for validation, authority, and audit.

Reliability is then carried by **structured interfaces**. Prefer *function schemas* and *structured outputs* so that model proposals must conform to a declared JSON schema; this turns free-form text into typed objects and sharply reduces silent failures in downstream tooling. Pair schema enforcement with *validators/guardrails* (type, range, pol-

icy, content filters) and bind all tools to *capability-scoped permissions* (least privilege, time-limited tokens, idempotency). Taken together, these choices make actions either “deterministically safe” or “deterministically refused” [JSO].

For **orchestration**, two families cover most needs. LangGraph [Lan] provides a low-level, stateful graph runtime: you define a shared state, nodes that transform it, and edges that encode control flow—ideal when you need explicit, replayable workflows and tight governance over long-running agents. AutoGen [Aut] treats conversation as computation for single- and multi-agent settings: you program roles, message routes, tool access, and stop conditions—useful when collaboration, debate, or supervisor–worker patterns are central. In both cases, constrain them with the same contracts: typed messages, per-edge budgets, per-role capabilities, and mandatory verifiers before any side-effectful act.

If the priority is *deterministic control and auditability* of a single agent (or a small number of tightly coupled ones), start with a state-graph (e.g., LangGraph) and insist on schema-checked nodes and transactional executors. If your priority is *division of labour and critique* across multiple roles, use a conversation framework (e.g., AutoGen) but add protocol invariants, termination rules, and an arbiter/verifier at the boundary to the real world. In either case, keep schema enforcement and validators close to the tool boundary, and record hashes of state, prompts, policies, and tool versions for replay.

The **key takeaway message** is that architecture—not just model choice—determines whether a powerful reasoning core becomes a dependable system. The components above provide isolation; schemas and validators provide contracts; orchestration provides controlled evolution over time. Put differently: *models propose, architectures dispose*. With this scaffold in place, the following chapters can focus on the specifics of data coverage, confidence estimation, cybersecurity, monitoring, and governance—plugging neatly into the interfaces you have already designed.

5 Summary

In this chapter, we argued that reliability in agentic and generative AI is fundamentally architectural: it is earned through principled componentisation (goal manager, planner, tool-router, executor, memory, verifiers, safety monitor, telemetry), disciplined interfaces (schema-constrained, validated, least-privilege tool calls), and explicit control loops that supervise reasoning and action. We mapped contemporary practice into a taxonomy—tool-using agents, memory-augmented agents, planning and self-improvement agents, multi-agent systems, and embodied/web agents—showing how each pattern alters the reliability envelope and introduces distinct failure modes. The diagnosis-agent example illustrated the recurring safeguards: simulate-before-actuate, typed contracts, idempotency and transactional semantics, permissioning, budget and termination rules, and end-to-end observability for audit and replay. Finally, we highlighted orchestration choices and how to constrain them so that powerful models provide solutions which are then validated before being accepted. With this scaffold in place, the subsequent chapters on data coverage, confidence estimation, cybersecurity, operational monitoring, and governance can plug into well-defined interfaces, turning capable model stacks into dependable autonomous systems.

References

- [Aut] Multi-agent conversation framework. https://microsoft.github.io/autogen/0.2/docs/Use-Cases/agent_chat/. Accessed: 2025-11-01.
- [BBK⁺24] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefer. Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682–17690, March 2024.
- [GMZ⁺23] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models, 2023.
- [JSO] Introducing structured outputs in the API. <https://openai.com/index/introducing-structured-outputs-in-the-api/>. Accessed: 2025-11-01.
- [KAB⁺22] Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tenenholz. MRKL systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning, 2022.
- [Lan] LangGraph overview. <https://langchain-ai.github.io/langgraph>. Accessed: 2025-11-01.
- [LHI⁺23] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for “mind” exploration of large language model society, 2023.
- [LPP⁺21] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks, 2021.
- [OWA] OWASP GenAI Security Project. <https://genai.owasp.org/>. Accessed: 2025-11-01.
- [PRW25] Priya Pitre, Naren Ramakrishnan, and Xuan Wang. CONSENSAGENT: Towards efficient and effective consensus in multi-agent LLM interactions through sycophancy mitigation. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 22112–22133, Vienna, Austria, July 2025. Association for Computational Linguistics.
- [PWL⁺24] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems, 2024.
- [RG95] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *ICMAS*, pages 312–319. The MIT Press, 1995.
- [SCB⁺23] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.
- [SDYD⁺23] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.
- [WBZ⁺23] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023.
- [WWS⁺23] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.
- [XPL⁺23] Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. ReWOO: Decoupling reasoning from observations for efficient augmented language models, 2023.
- [YYZ⁺23] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

- [YZY⁺23] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models, 2023.