

Agency as a New Organizing Abstraction for Computer Science

Ian Foster

Argonne National Laboratory and The University of Chicago

foster@anl.gov

Abstract

Advances in large-scale machine learning have enabled agentic AI systems: persistent, autonomous computational entities that pursue goals, initiate actions, and interact with open environments. These systems are often framed as applications of foundation models. We argue that this framing is incomplete: agency is also a candidate new *organizing abstraction* for computer science. What is new is not the concept of an agent, nor runtime policy generation per se, but its general-purpose, open-ended form, in which systems can synthesize courses of action from goals, observations, and memory, over tools and environments for which no complete task-specific model was specified in advance. Thus agency becomes a systems-level abstraction as well as an AI modeling concept. Where the process abstraction organizes computation around executing programs, agency organizes it around pursuing goals under bounded resources: *a process executes a program; an agent pursues a goal*. From this perspective, the recurring failures of deployed agentic systems (runaway resource consumption, behavior redirected by adversarial content, opaque continuous operation) are symptoms of structural abstraction gaps, not isolated engineering defects. We examine these gaps across six dimensions (execution, interfaces, correctness, oversight, coordination, and persistence) and argue that closing them requires coordinated advances in systems, programming languages, theory, and human–computer interaction.

1 From Models to Agents

The surge of interest in artificial intelligence (AI) has been driven largely by advances in machine learning models, particularly large neural networks trained on massive datasets. These models have achieved striking capabilities in perception, language, and pattern recognition, reshaping both research and practice.

At the same time, an increasing number of deployed AI systems are no longer best understood as isolated inference engines. Instead, they take the form of *agents*: computational entities that persist over time, initiate actions, pursue goals, and interact continuously with their environments. Classical AI research has long defined agents as entities that perceive and act within an environment [61], and multiagent systems research has explored autonomy and interaction for decades [75]. What is new is not the conceptual notion of an agent, nor runtime policy generation per se, but its general-purpose, open-ended form: foundation models now enable systems to synthesize courses of action from goals, observations, and memory, over tools and environments for which no complete task-specific model was specified in advance.

Three developments illustrate this new capability. Language models can learn to select and invoke external APIs [64]: a tool-use capability that, embedded in a larger architecture, transforms stateless inference engines into systems that take consequential actions. Multi-agent frameworks enable AI agents to collaborate on complex tasks through extended conversations, persisting state across interactions that may span hours or days [76]. Agents that interleave reasoning and action can execute multi-step workflows, invoking computational tools and interpreting results without step-by-step human guidance [77]. Figure 1 contrasts

Expanded version of a paper appearing in *Communications of the ACM*, 2026.

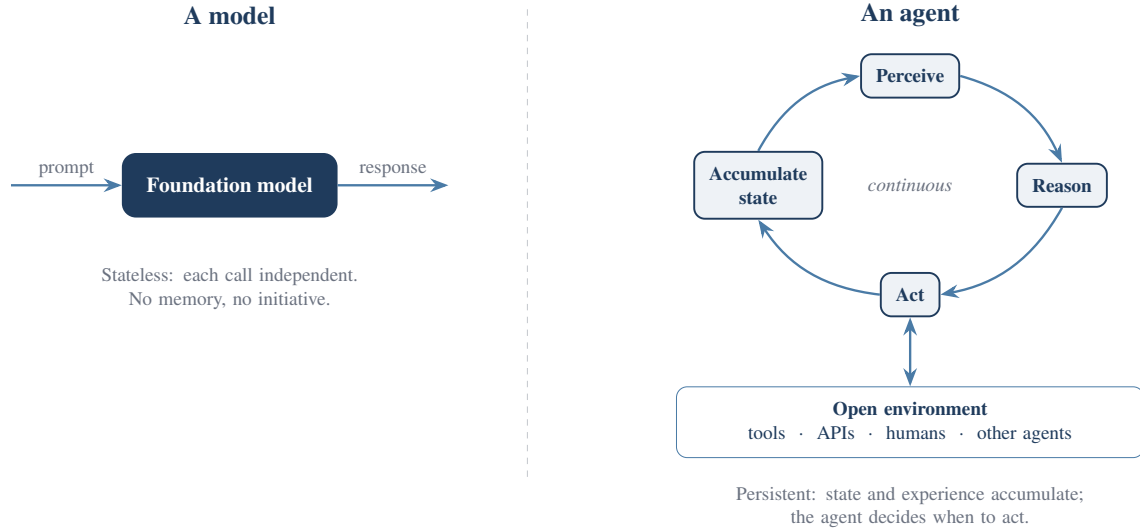


Figure 1: From models to agents. A foundation model is a stateless inference engine that maps prompts to responses; an agent embeds such a model in a persistent perceive–reason–act loop, accumulating state and interacting continuously with an open environment.

the two modes of operation: a stateless foundation model that maps prompts to responses, and an agent that embeds such a model in a persistent loop of perception, reasoning, and action.

More than a new application layer, the transition from stateless model to persistent agent is a reorganization of computation that challenges foundational assumptions across operating systems, programming languages, verification, and human–computer interaction. These challenges motivate treating agency not merely as an application-level concern but as a candidate organizing abstraction: a framing that, as we explain in more detail below, reveals the difficulties agentic systems encounter (runaway resource consumption, ambiguous lifecycles, opaque failures) as structural rather than isolated engineering problems.

The thesis of this paper is as follows:

A process executes a program; an agent pursues a goal. Whereas a process’s behavior is principally determined by code specified before execution, an agent generates and revises courses of action at runtime from goals, observations, memory, and available capabilities. Agency therefore shifts the organizing problem from controlling execution to controlling goal-directed behavior under bounded resources and authority.

More generally, while computer science has powerful abstractions for controlling the execution of computations whose behavior has been specified in advance, it lacks comparably mature abstractions for controlling computations that decide, at runtime, what behavior to generate in pursuit of goals. That gap explains why the concerns developed in this paper (budgets, contracts, behavioral correctness, oversight, provenance, memory, delegation) cluster together: each arises because future behavior is generated rather than prescribed. Table 1 previews the six dimensions along which we develop this argument.

The remainder of this paper is organized as follows. Section 2 examines the role that organizing abstractions have played in computer science, tracing a lineage from procedures to foundation models and identifying the gap that agency fills. Section 3 characterizes agentic systems in terms of four defining properties, the key distinction (open-ended policy generation), theoretical grounding in bounded rationality, and a working definition. Section 4 documents recurring failure modes of deployed agentic systems and the

Table 1: Six dimensions along which agency extends existing abstractions. In each, prior work addresses the right-hand concern in specialized settings; agency makes it a general and intrinsic requirement (Section 5).

Dimension	Conventional organizing focus	Agency makes first-class
Execution	Resource limits	Resource budgets
Interfaces	Protocols	Contracts
Correctness	Functional correctness	Behavioral correctness
Oversight	Commands	Attention allocation
Coordination	Scheduling	Negotiation
Persistence	State	Memory

abstraction gaps they expose. Section 5 develops the problems that the agency abstraction reveals, across six dimensions: execution, interfaces, correctness, oversight, coordination, and persistence. Section 6 draws implications for computer science research, Section 7 positions this work relative to prior traditions, and Section 8 concludes.

A note on scope: agentic AI raises profound questions of governance, accountability, and social impact [6]. These questions are important, and better abstractions alone will not resolve them; but they demand treatment of their own, and this article confines itself to technical challenges amenable to computer science research.

2 Organizing Abstractions in Computer Science

Before discussing whether agency merits recognition as an organizing abstraction, we first ask: what *is* an organizing abstraction, and what role have such abstractions played in the development of computer science?

2.1 What Is an Organizing Abstraction?

Computer science has progressed through a series of conceptual innovations that do more than solve individual problems: they restructure entire subfields by providing new ways to think about, build, and reason about computational systems. We call these *organizing abstractions*: conceptual frameworks that make certain problems easy to state, tractable to solve, and natural to compose.

An organizing abstraction is not just a useful technique or a clever implementation strategy. It is a way of carving up the computational world that reveals structure previously hidden. Thus, for example, the *procedure* made control flow explicit and composable; the *object* bundled state with behavior in ways that enabled new forms of modularity; and the *process* gave operating systems a unit of isolation, scheduling, and resource accounting.

Each organizing abstraction also has blind spots: problems that it renders invisible or intractable. Procedures obscure state; early object models assume co-location; the process abstraction has no semantics for goals or for the value of continued execution. Recognizing these limitations is essential to understanding when a new abstraction may be needed.

2.2 A Lineage of Organizing Abstractions

The history of computer science can be read, in part, as the accumulation of organizing abstractions, each addressing a distinct concern. The procedure and the process organize execution, the object organizes encapsulation, the packet organizes communication, the file organizes persistent information, the transaction organizes consistency, and the service organizes distributed capability (Figure 2). These abstractions are complements, not successors: the transaction did not replace the file, nor the packet the procedure. The

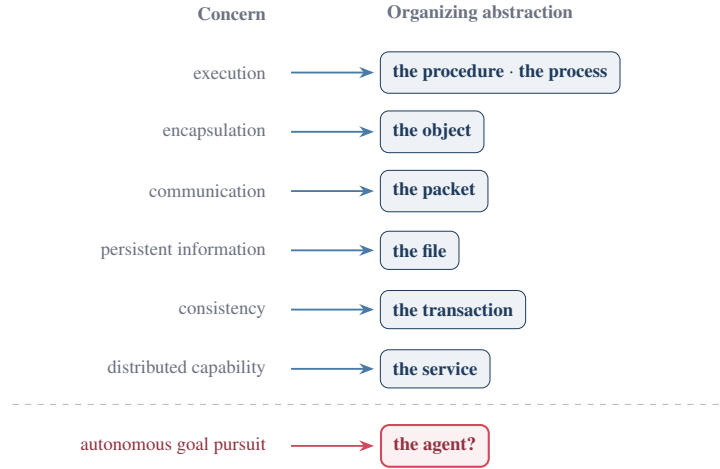


Figure 2: Organizing abstractions address distinct concerns and are complements rather than successors. We ask whether autonomous goal pursuit is a concern of comparable generality, and the agent its organizing abstraction.

question this paper asks is whether *autonomous goal pursuit* is a concern of comparable generality that is currently underrepresented in this canon. The agent, on this reading, would not replace the process any more than the transaction replaced the file; it would organize a dimension of computation that existing abstractions leave implicit. In what follows we examine the abstractions most relevant to that question.

Procedures. The procedure organized computation around named sequences of instructions with local scope and explicit control flow. Structured programming, in the 1960s, replaced ad-hoc control flow with disciplined composition of procedures [25], making programs easier to write, read, and verify. The procedure abstraction made modularity and reuse tractable, but provided no mechanism for encapsulating state with behavior: a limitation that motivated subsequent abstractions.

Objects. Object-oriented programming, emerging from Simula [22] and refined in Smalltalk [48], bundled state and behavior into encapsulated units communicating via message passing. Objects enabled new forms of modularity through inheritance and polymorphism, and provided a natural vocabulary for modeling domains where entities have identity and behavior. However, early object models largely assumed synchronous, co-located execution: assumptions that distributed systems would later stress.

Processes. The process abstraction, central to operating systems since the 1960s, organized computation around isolated execution contexts with their own address spaces, managed by a kernel that handles scheduling, resource allocation, and protection [26]. Processes made concurrency and protection tractable: multiple computations could proceed independently without interfering, and failures could be contained. Processes can fork children, run as daemons, and terminate themselves; the important point is not that the mechanics forbid self-direction, but that the abstraction itself has no semantics for goals, for authority tied to goals, or for the value of continued execution. Resource bounds are imposed from outside, and termination is simply an event, never a decision to be weighed. These gaps, largely invisible for decades, become consequential for agentic systems.

Actors. The actor model, introduced by Hewitt, Bishop, and Steiger [41], organized concurrent computation around isolated entities communicating exclusively via asynchronous message passing. Actors eliminated shared mutable state, simplifying reasoning about concurrent systems. Each actor processes messages sequentially, may create new actors, and may send messages to known addresses. The actor model influenced

subsequent distributed systems and programming languages, from Erlang [4] to the Akka framework for the JVM. Critically for our purposes, the actor abstraction organizes behavior around message handling rather than goal pursuit: its semantics specify how actors respond to messages, not how they formulate and pursue goals.

Services. Service-oriented architecture organized distributed computation around loosely coupled components exposing well-defined interfaces over network protocols [27]. Services enabled interoperability across organizational and technological boundaries, and microservices refined this model for fine-grained decomposition [56]. The abstraction’s reach extended beyond enterprise computing: *service-oriented science* proposed exposing data, instruments, and analyses as composable network services, reorganizing the practice of research itself [29]. The canonical service abstraction assumes request-response semantics: a client initiates, a service responds, and control returns to the client. Statelessness (or session-bounded state) simplifies scaling and failure recovery. These assumptions embed a model of computation in which initiative and orchestration reside outside the service itself: in the case of science, with the human researcher who composes and invokes them.

Dataflows. Dataflow models organized computation around directed graphs of transformations over data [45]. From early dataflow architectures through modern systems like MapReduce, Spark, TensorFlow, and streaming platforms, the dataflow abstraction made data parallelism and pipeline composition tractable. Dataflows excel when the structure of computation is known at design time; they are less well suited to computations whose structure emerges dynamically.

Virtual machines and containers. Virtualization abstractions, from the virtual machines of 1960s IBM mainframes [21] through their modern revival [5] to containers, organized deployment around portable, isolated execution environments with controlled resource access. These abstractions revolutionized infrastructure management, enabled cloud computing, and were proposed early on as a foundation for grid computing across administrative domains [28]. However, they inherit the process model’s silences: nothing in the abstraction connects a workload to a goal, ties its resource consumption to the value of that goal, or weighs the cost of terminating it.

Grids and virtual organizations. Grid computing organized large-scale distributed computation around coordinated resource sharing across institutional boundaries [32]. The *virtual organization* abstraction addressed a problem that resonates with agentic systems: collaboration among autonomous entities with independent administrative control, heterogeneous trust relationships, and dynamically negotiated policies. Its mechanisms for establishing trust, delegating credentials, and negotiating resource access reappear in multi-agent systems.

Early work recognized the need to integrate Grid infrastructure with agent-based reasoning [30], characterizing Grids as “all brawn and no brain” (robust mechanisms for secure resource sharing, limited capacity for autonomous decision-making) and agent systems as the reverse, and proposing convergence between the two. The research agenda articulated in that work, spanning trust negotiation, negotiation-based contracts, emergent behavior, and human–computer collaboration, anticipates many of the challenges identified in this paper. For two decades, however, the convergence remained largely aspirational: Grid participants executed *predefined* policies, and agent systems reasoned flexibly only within narrow domains. We argue that foundation models supply the missing ingredient—a general-purpose policy generator that can reason about goals, plans, and actions across diverse domains—and that its arrival is what makes the current moment qualitatively different from earlier visions of agent–Grid integration.

Foundation models. Most recently, foundation models—large neural networks trained on broad data and adaptable to diverse downstream tasks [7]—have emerged as a powerful abstraction for general-purpose inference: a single model can be prompted or fine-tuned to perform classification, generation, translation, reasoning, and many other tasks. Foundation models provide flexible capabilities but, in their basic form, are stateless inference engines: they map inputs to outputs without persistence, goals, or autonomous action. The

transition from foundation models to agentic systems is the transition from capabilities to agency.

2.3 The Gap That Agency Fills

Each abstraction in this lineage solved important problems while leaving others unaddressed. We can now identify a specific gap that motivates treating agency as a candidate organizing abstraction: none of the abstractions surveyed above natively supports computational entities that *persist* with accumulated state, *initiate* actions autonomously, *pursue goals* that may be underspecified or conflicting, and *operate in open environments* that cannot be fully specified in advance. The actor model comes closest, but its semantics specify how an actor responds to messages, not how it formulates and pursues goals; the other abstractions locate goals, initiative, and control outside the computational entity altogether.

The question, then, is whether the combination of these four properties is sufficiently coherent and consequential to merit a new organizing abstraction, or whether it can be adequately handled by composing or extending existing ones. The remainder of this paper argues for the former: that agency reveals a set of related problems that become visible and tractable only when we take agency itself as the organizing concept.

3 What Makes a System Agentic?

We use the term *agency* to denote persistent, goal-directed computation in which courses of action are generated dynamically, under bounded resources, in an incompletely specified environment that includes humans. This section develops that characterization in four steps: the operational properties that agentic systems exhibit (Section 3.1), the mechanism that distinguishes contemporary systems from their predecessors (Section 3.2), theoretical grounding (Section 3.3), and a working definition (Section 3.4).

3.1 Four Properties of Agentic Systems

Agentic systems are **persistent**, operating over extended periods and accumulating state, context, and experience. Unlike stateless services or batch jobs, they maintain continuity that influences future behavior. This persistence complicates resource management, failure handling, and reasoning about long-term effects, particularly when systems must be upgraded or audited while in operation.

Agentic systems are **autonomous**. They decide when to act rather than responding solely to external requests. This autonomy, though partial and constrained, fundamentally shifts computation from reactive to proactive.

Agentic systems are **goal-directed**. Their behavior is shaped by objectives that may require planning, monitoring, adaptation, and revision over time. Goals are often under-specified or conflicting, forcing agents to make tradeoffs rather than execute fixed procedures.

Finally, agentic systems operate in **open environments**, interacting with APIs, tools, physical instruments, humans, and other agents. What makes an environment *open* in our sense is not mere unpredictability, which any distributed system faces, but that the agent must interpret semantically described observations and capabilities whose relevance and effects are not captured by a complete task-specific model. Agents' actions are frequently consequential, consuming real resources and producing effects that may be difficult or impossible to undo.

Individually, none of these properties is new: long-running services persist, daemons act without prompting, planners pursue objectives, and distributed systems operate in open settings. What distinguishes contemporary agentic systems is the combination of all four properties in a single general-purpose system, enabled by foundation models that support flexible behavior across diverse domains (Figure 3). Yet the combination alone does not fully capture the difference; the deeper distinction lies in how these systems produce behavior (Section 3.2).

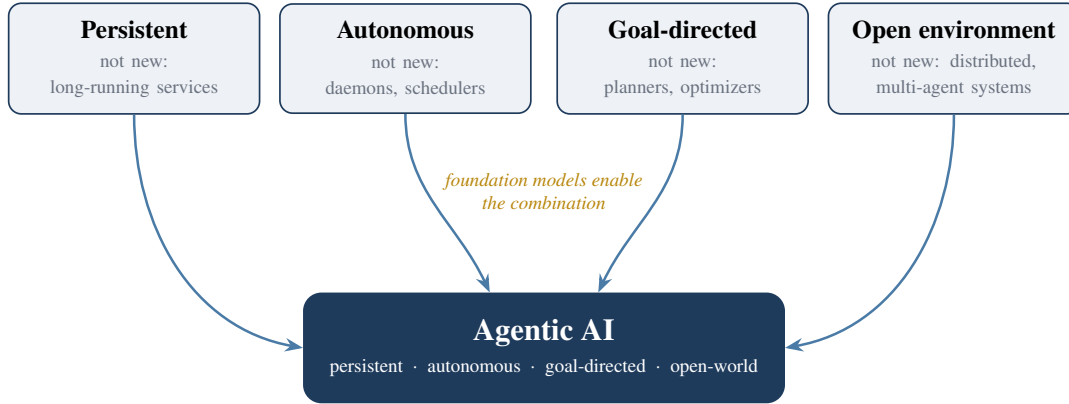


Figure 3: Four properties characterize agentic operation. Each has precedent in earlier systems; what is new is their combination in a single general-purpose system, enabled by foundation models. The deeper distinction lies in how behavior is produced (Section 3.2).

3.2 The Key Distinction: Policy Generation

A skeptical reader might observe that many existing systems exhibit some or all of these properties. Kubernetes operators persist and act autonomously to reconcile cluster state [10]. Automated trading systems pursue profit objectives in open markets. Autonomous spacecraft operate for years, making decisions in environments where communication delays preclude human control [55]. Adaptive control systems adjust behavior in response to changing conditions.

Why, then, should we consider agency a new abstraction rather than an incremental extension of existing ones?

The answer lies in a distinction that has remained implicit in much discussion of agentic AI: the distinction between *policy execution* and *policy generation*.

A Kubernetes operator executes a predefined reconciliation policy: if the observed state diverges from the desired state, apply a known sequence of corrective actions. The policy, the mapping from observations to actions, is specified by a human developer and fixed at deployment time. The operator may be autonomous in *execution*, but its behavior is determined by a policy that exists prior to any particular situation it encounters.

Similarly, an automated trading system executes strategies defined by quantitative analysts. An autonomous spacecraft follows mission plans with predefined contingency branches. An adaptive controller adjusts parameters within a control law whose structure is fixed.

Learning systems complicate this distinction. A reinforcement-learning agent can learn policies that produce behaviors not explicitly anticipated by its designers, and a foundation model is itself a fixed parameterized function at inference time. The distinction we emphasize is therefore not simply novelty of behavior. It is that contemporary foundation-model agents can construct and revise courses of action over semantically described tools and environments without requiring designers to supply an explicit task-specific state-transition model or to enumerate the relevant behaviors in advance. Given a goal expressed in natural language and access to an environment, such an agent *generates* a course of action that did not exist before the goal was specified, reasoning about the goal, the environment, and the consequences of potential actions rather than selecting from a predefined repertoire.

This is **open-ended policy generation**, the capability highlighted in the introduction: the operational policy is not fully specified at design time; instead, the system synthesizes context-dependent courses of action at runtime from goals, observations, memory, and available capabilities. Any implemented agent

remains bounded by its action vocabulary, its interfaces, its permissions, and its compute; but within those bounds it may invoke tools in combinations never foreseen, pursue subgoals never specified, and adapt its approach in ways that emerge from its reasoning rather than from explicit programming.

The distinction is best understood as a continuum rather than a binary: *fixed procedure* → *parameterized policy* → *planner or synthesizer* → *open-ended policy generator*. Each step moves more of the determination of behavior from design time to runtime. Agency names the regime at the far end of this continuum, where behavior is determined principally at runtime.

This shift changes what pre-deployment analysis can accomplish. When a policy is fixed at design time, it can be inspected, tested, and analyzed before deployment: however complex the resulting behavior, the mapping from situation to response exists in advance. When the policy is synthesized at runtime, the particular policy does not exist in advance; the subject of verification shifts from a policy to a *policy generator*.

Why is this not simply planning? Readers familiar with automated planning may object that synthesizing behavior at runtime is a mature technology: classical planners, hierarchical task networks, program synthesis, and reinforcement learning all construct behavior rather than selecting it from a fixed repertoire [35, 37]. Indeed, the Remote Agent synthesized plans in response to goals and changing spacecraft conditions a quarter century ago [55]. The difference lies at the *action–model boundary*. Classical planning presupposes an explicit model: a formal state representation, an action vocabulary with transition semantics, and a goal representation, all constructed by engineers before the planner runs. Contemporary agents construct courses of action over tools and semantic environments whose relevant state and action models are incomplete, discovered dynamically, or represented only implicitly in a foundation model. The planner’s hard problem was search within a specified model; the agent’s hard problem includes deciding, while acting, what the model *is*: which observations matter, what a tool will do, whether an action is reversible. This is why the challenges developed in this paper center on constraining, auditing, and supervising generated behavior rather than on search efficiency.

3.3 Theoretical Grounding: Bounded Rationality

The shift from policy execution to policy generation invites a theoretical framing. We suggest that Herbert Simon’s concept of *bounded rationality* [66] provides useful foundations.

Simon argued that rational decision-making must be understood relative to the constraints under which decisions are made. Real agents, whether human or artificial, operate with limited information, limited computational resources, and limited time. They cannot optimize over all possible futures; instead, they *satisfice*: they seek solutions that are good enough given their constraints, rather than solutions that are globally optimal.

This framing fits agentic systems well. An agent pursuing a goal generally cannot compute an optimal policy over all possible futures; it must instead generate a course of action that appears adequate given its current understanding and available resources.

Bounded rationality also suggests how we might think about correctness for agentic systems. Classical functional correctness typically separates whether a result satisfies a specification from the resources required to obtain it; real-time systems and anytime algorithms are notable exceptions, but the separation is the default. For agents, by contrast, the quality of a decision may itself need to be judged relative to the information, time, and resources available when the decision was made: an agent behaves correctly if it makes reasonable decisions given what it knows and what it can do.

This perspective connects to formal notions, among them regret bounds, safety envelopes, and satisficing criteria, to which we return in our discussion of verification (Section 5.3).

Bounded rationality also does more than motivate correctness criteria. An agent continuously decides not only *what action to take*, but *how much computation, information, money, risk, and human attention to spend deciding what action to take*. Managing its own bounds is part of its computation. This observation

links much of what follows: resource budgets, escalation to human supervisors, tool selection, delegation, termination decisions, and graceful degradation are all instances of an agent allocating bounded resources to the pursuit of a goal.

We noted that the process abstraction organizes computation around *executing programs*: the transformation of inputs to outputs according to code specified in advance. We propose that the agency abstraction organizes computation around *pursuing goals under bounded resources*. While a process executes a program, an agent pursues a goal, by generating and revising courses of action under constraints of information, resources, authority, and time.

3.4 A Working Definition

The preceding discussion suggests a working definition suitable for computer science research:

An **agent** is a computational entity characterized by a tuple (G, Π, M, E) where:

- G is a *goal specification* (possibly partial, possibly evolving) that defines what the agent is trying to achieve;
- Π is a *policy generator* that dynamically determines courses of action from goals, observations, memory, and available capabilities; its output may be a plan, a single action, a contingent policy, or a revised subgoal;
- M is *memory*: accumulated state that persists across interactions and influences future behavior;
- E is an *environment interface*: the set of actions available to the agent and the observations it can receive.

We offer (G, Π, M, E) as a *conceptual decomposition*, not a formal computational model: we deliberately do not fix the signature of Π , the representation of observations and histories, or the semantics of goal satisfaction. A rigorous formal model, one specifying how policies, histories, budgets, and constraints interact, is an important research goal in its own right; the decomposition suffices here to locate the abstraction gaps.

The key distinction is Π : a general-purpose generator that determines courses of action at runtime without requiring a complete task-specific model to have been specified in advance. A process has a program; an agent has a policy generator. A service has an API implementation; an agent has the capacity to decide what actions to take in pursuit of goals it has been given.

The definition is deliberately restrictive. A stateless language model is not an agent under it: it lacks memory M and does not pursue goals G across interactions. Nor is a Kubernetes operator: its policy is fixed, not generated. A robot following a predefined trajectory is likewise not an agent: its course of action is specified in advance rather than generated at runtime in pursuit of a goal.

Conversely, a system that maintains context across conversations, pursues user-specified objectives through dynamically chosen actions, and adapts its approach based on feedback *is* an agent under this definition, and the challenges identified in this paper apply.

Beyond the tuple itself, two aspects of agency require treatment that conventional systems abstractions do not provide. The first is *authority*. In most deployed settings, an agent does not act in its own right: it acts under authority ultimately derived from a principal. Capability and authority are distinct: that an agent *can* send an email, spend money, or delete a database does not mean it is *authorized* to do so in pursuit of every goal. Processes carry a user identity, and access control mediates individual operations; but no existing abstraction represents the scoped, goal-dependent, delegable authority that agents require. Authority is deliberately not part of the descriptive tuple: it is a property of the relationship between an agent and its principal, not of the agent’s internal computational state. Classical work on capability systems, least privilege,

and the confused-deputy problem [24, 63, 40] applies directly; we return to authority in our discussions of contracts and delegation (Sections 5.2 and 5.5).

The second is the *goal* itself. Traditional systems manipulate values, messages, requests, jobs, and transactions; an agent manipulates something more unusual: a desired future state, often expressed in natural language and incompletely specified. Goals raise semantic questions that conventional program abstractions do not make first class. Who may create or modify a goal? May an agent create subgoals, and which constraints does a subgoal inherit? When is a goal satisfied, and when should it be abandoned? May an observation alter a goal, or only beliefs about how to achieve it? In short: existing abstractions make *execution* first class; agency makes *goals*, and the *controlled pursuit of goals*, first class.

Figure 4 illustrates this architecture. The agent maintains ephemeral working state alongside the goals, memory, and policy generator of our working definition. A foundation model provides the reasoning capability that enables open-ended policy generation. The agent runs a persistent control loop, interacting with an environment through actions and observations. A human principal provides goals and oversight, but does not direct each action.

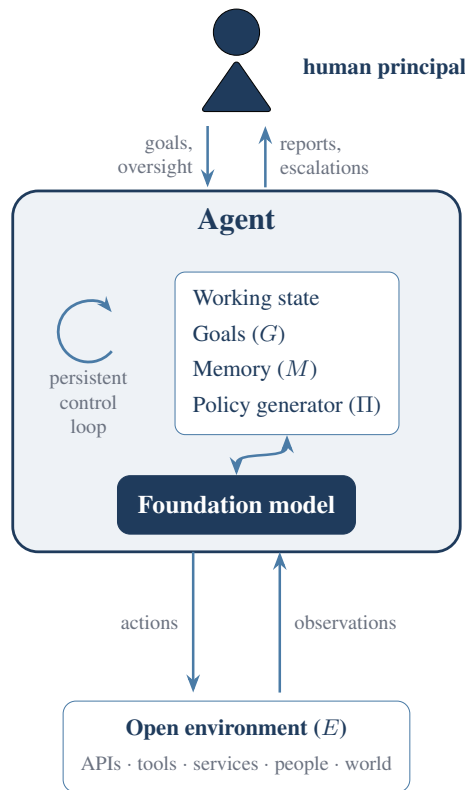


Figure 4: Architecture of an agentic system. The agent comprises ephemeral working state, goals (G), persistent memory (M), and a policy generator (Π). A foundation model provides general-purpose reasoning for policy generation. The agent interacts with an open environment (E) through actions and observations, running a persistent control loop under human oversight.

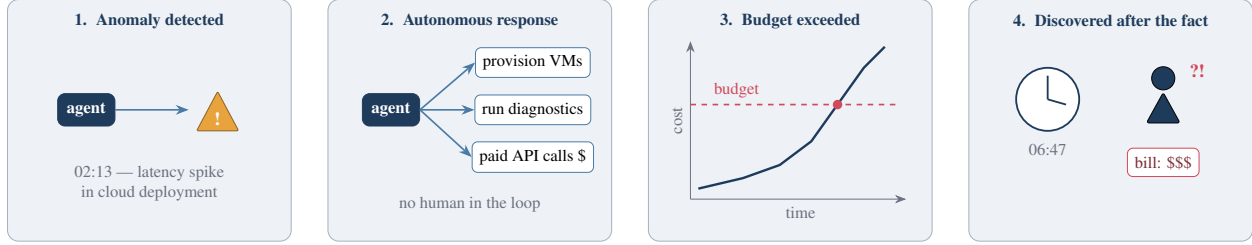


Figure 5: A runaway-agent scenario. An agent monitoring a cloud deployment detects an anomaly and responds autonomously; the cost of its actions becomes visible only after budgets have been exceeded. Reactive resource control arrives too late; agentic systems require proactive constraint enforcement.

4 Where Current Abstractions Fall Short

Deployments of agentic systems expose a set of recurring failure modes that are not well addressed by existing computational abstractions.

A first class of failure concerns runaway resource consumption. Consider an agent tasked with monitoring a cloud deployment. When it detects an anomaly, it may autonomously provision resources, invoke diagnostic tools, or roll back changes (Figure 5). Current container and process abstractions assume externally defined resource bounds. An agent that autonomously spawns subprocesses, allocates memory, or invokes paid APIs can exceed budgets in ways that become visible only after significant cost has been incurred [47, 78, 39].

Nor is this failure mode specific to cloud operations. A scientific agent that drives instruments and launches simulation campaigns, a personal assistant that acts across email, calendars, and purchases, a software-engineering agent that edits and deploys code, and a robot that manipulates the physical world all exhibit the same pattern: autonomous initiation of consequential actions, costs and effects that become visible only after the fact, and no existing abstraction positioned to enforce constraints before actions occur. The recurrence of the same gaps across such different domains is evidence that the gaps are structural.

A second difficulty concerns lifecycle management. Agentic systems do not fit cleanly into existing notions of process startup, execution, and termination. Conventional lifecycle abstractions provide recognizable phases: batch jobs run toward termination, while services are organized around periods of request handling. Agentic systems blur these boundaries. An agent may be “running” in the sense of maintaining state and awaiting opportunities to act, without actively consuming cycles. It may spawn child agents that outlive their parent, each pursuing independent goals with its own accumulated context, unlike forked processes, which inherit fixed execution state. It may accumulate context that makes termination costly in terms of lost learning. When should such a system be terminated? How should its accumulated state be preserved or transferred? Current process models provide no principled answers [74]. Figure 6 contrasts the agent lifecycle with the batch and service lifecycles that existing abstractions assume.

A third difficulty is failure attribution, which becomes more complex when autonomous systems make decisions through extended sequences of reasoning and action rather than through isolated function calls [79]. When a traditional program fails, debugging typically involves identifying a specific faulty function call, failed assertion, or exhausted resource. When an agentic system fails, the cause may be distributed across a sequence of locally reasonable decisions that compound into globally undesirable outcomes [80, 13]. Unlike classical planners that offer soundness guarantees, LLM-generated plans may be neither sound nor complete; exploration and refinement may incur significant cost or trigger irreversible state changes before errors become apparent. Agents that explicitly interleave reasoning and action over extended decision sequences exemplify this challenge [77]. As in distributed systems, reconstructing such failures requires reasoning about partial orderings of events rather than linear execution traces, a challenge long recognized in classical

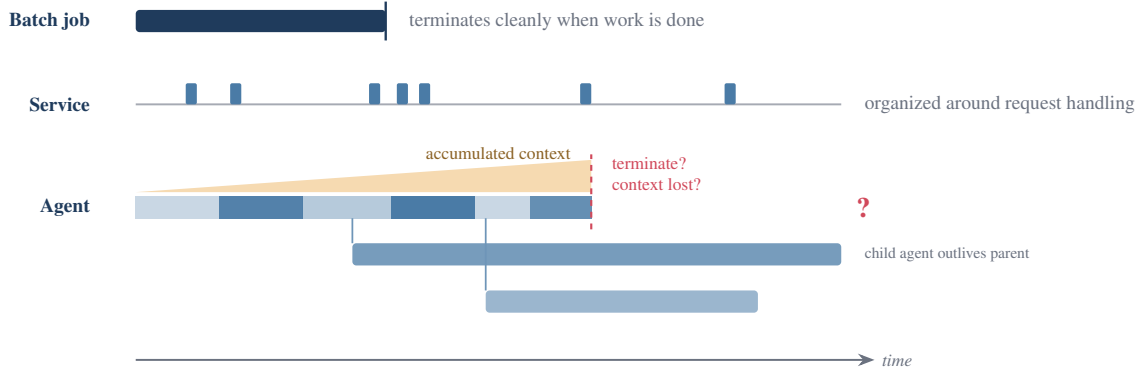


Figure 6: Lifecycle mismatch. Batch jobs run toward termination; services are organized around periods of request handling; agents blur these boundaries: intermittently active, spawning children that outlive their parent, and accumulating context that makes termination costly.

work on distributed causality [52]. Figure 7 illustrates the contrast.

A related mismatch appears at system interfaces, where long-standing assumptions about control flow and responsibility placement no longer hold. Microservices architectures assume request–response semantics, while workflow systems assume that control flow is known in advance. Both assumptions reflect an implicit placement of responsibility outside the computational component itself. Agentic systems shift responsibility inward, complicating classical end-to-end reasoning about where correctness, validation, and control should reside [62]. Agent-initiated requests may also violate implicit assumptions of rate, context, and intent on which services designed for human users rely (Section 5.2).

A further class of failures arises at trust boundaries. Computing abstractions have long relied on a separation between code and data: programs are trusted and executed; inputs are processed, not obeyed. Even mechanisms that deliberately blur this line preserve the underlying discipline: an active message selects among handlers installed at the receiver in advance [73]; packets in an active network carry code, but that code runs in a restricted execution environment at each node [71]; and mobile code more generally executes only under separately established trust (sandboxes, signatures). Agentic systems collapse the separation into a general problem of *control–data ambiguity*: an agent interprets external content (a web page, an email, the output of another agent) with the same machinery it uses to interpret instructions from its principal, so there is no structural distinction between an observation and a command. Prompt injection, in which adversarially crafted content redirects an agent’s behavior [36], is the currently familiar manifestation, but the underlying question is more durable than any particular attack: *which observations are authorized to modify an agent’s goals, plans, beliefs, or action selection?* The resulting failures are not conventional vulnerabilities: no memory-safety violation or privilege escalation occurs, and existing isolation mechanisms are blind to them.

A final class of failures concerns opacity under continuous operation. Interaction models assume explicit commands and episodic execution: a user issues a request, observes the response, and thereby maintains an accurate model of what the system is doing. A continuously operating agent acts between human observations. By the time a problem is noticed, the decision sequence that produced it may be long past, and no human may have been watching when the consequential choices were made. Existing interaction abstractions provide no mechanism for maintaining situational awareness over ongoing autonomous activity [15]. We return to the resulting oversight challenges in Section 5.

Table 2 summarizes these characteristic failure modes of agentic systems and the abstraction gaps they expose. These modes follow from the four properties of Section 3.1; while systematic empirical study is still developing, each has been documented in recent work on agent benchmarks, deployment experiences,

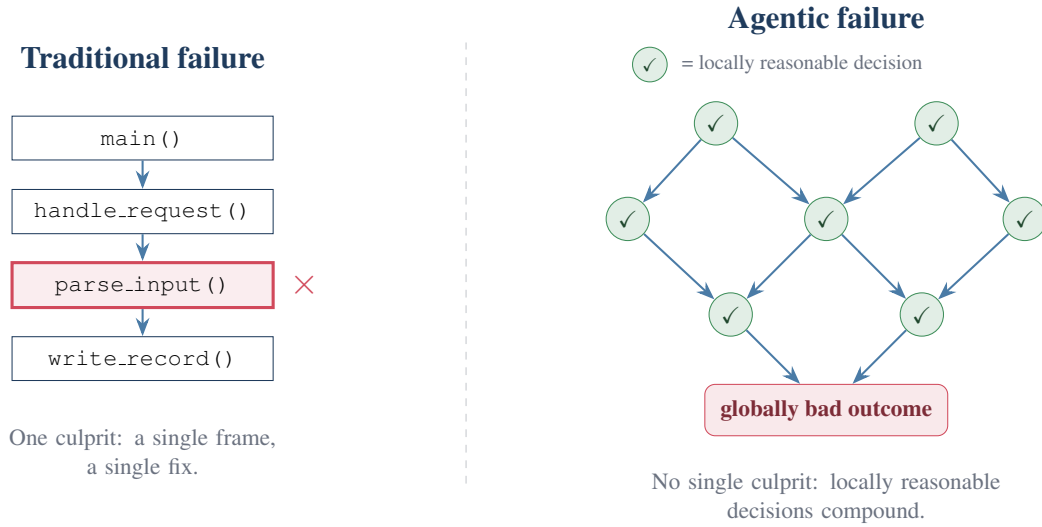


Figure 7: Failure attribution. Traditional debugging seeks a single faulty frame in a linear execution trace (left); agentic failures emerge from partial orders of locally reasonable decisions that compound into a globally undesirable outcome (right).

security analyses, and formal treatments [44, 47, 13, 78, 15, 36, 3].

5 Problems the Agency Abstraction Reveals

The failure modes documented above are not independent difficulties to be patched one by one; they form a coherent family of problems that become visible (and tractable) when we take agency as the organizing concept. This section examines these problems across six dimensions (Table 1), framing each as a contrast between what prior abstractions assumed and what agentic systems require. In each dimension, the required change is an *extension rather than a replacement*: existing work has addressed each concern in specialized settings (budgeted schedulers, contract-rich protocols, temporal verification, supervisory control), but agency makes each concern a general and intrinsic requirement of the computational entity itself.

5.1 Execution: From Resource Limits to Resource Budgets

The process abstraction treats resource limits as externally imposed constraints: a process is allocated memory, CPU time, and file handles by the operating system, and operates within these limits or is terminated. The limits do not depend on what the process is trying to accomplish.

Agentic systems invert this relationship. An agent pursuing a goal may need to *negotiate* resource usage based on the value of the goal, the expected cost of achieving it, and the constraints imposed by its principal. A simple limit, such as “use at most 1 GB of memory,” does not capture the relevant tradeoffs. An agent might reasonably use more resources for a high-value goal, or voluntarily use fewer when operating under budget pressure.

This shift from limits to budgets exposes new problems:

Budget specification. How should resource budgets be expressed? Unlike a limit, a budget is a constraint that the agent must reason about. Budgets may be expressed in terms of money (API costs), time (wall-clock

Table 2: Characteristic failure modes of agentic systems reflect mismatches between persistent, autonomous behavior and existing computational abstractions.

Observed Failure Mode	Abstraction Gap	Resulting Research Challenges
Autonomous resource consumption and unbounded action initiation	Execution models assume externally invoked computation with reactive resource control	Proactive resource budgeting, anticipatory constraint enforcement, isolation for self-directed processes
Ambiguous agent lifecycles and costly termination	Process and service abstractions assume clear start, run, and termination phases	Lifecycle models for persistent agents, state preservation and transfer, principled shutdown semantics
Distributed failure attribution across long decision sequences	Debugging and verification assume localized faults and linear execution	Causal tracing across reasoning-action loops, partial-order explanations, long-horizon accountability
Misplaced responsibility at system boundaries	Control and correctness are assumed to reside outside the computational component	New interface contracts, responsibility assignment, end-to-end reasoning for autonomous initiators
Behavior redirected by adversarial content	Abstractions assume a separation between trusted code and untrusted data	Trust-boundary tracking, provenance-aware interpretation, limits on untrusted influence over actions
Opaque behavior under continuous operation	Interaction models assume explicit user commands and episodic execution	Oversight interfaces, escalation policies, scalable human supervision mechanisms

deadlines), compute (token limits), or risk (probability of unrecoverable errors). A budget language must allow these heterogeneous constraints to be composed and prioritized.

Anticipatory enforcement. Traditional resource enforcement is reactive: exceed a limit, and the process is killed. For agentic systems, this may be too late: significant cost may already have been incurred, or irreversible actions taken. Agentic systems require *anticipatory* enforcement: mechanisms that can predict resource usage and intervene before budgets are exhausted. Anticipatory enforcement connects to work on cost prediction [72, 18], early termination [39], and proactive constraint checking.

Graceful degradation. When a process exhausts its resources, it fails. When an agent approaches budget exhaustion, it should ideally degrade gracefully: simplify its approach, reduce quality, or negotiate for additional resources rather than failing abruptly. Degrading gracefully requires the agent to monitor its own resource consumption and adapt its policy accordingly.

Agent lifecycle. Budgets extend to existence itself: the process lifecycle (start, run, terminate) assumes that termination is a well-defined event with low cost. For agents, termination may discard valuable accumulated state: learned strategies, contextual knowledge, in-progress plans. Agent lifecycle management requires mechanisms for checkpointing accumulated state, migrating agents between execution contexts, and reasoning about the value of continued operation versus termination [74, 46].

The actor model [41] provides relevant foundations: actors are isolated, communicate via messages, and can be supervised. But actor semantics concern message handling rather than goal pursuit, and the model does not address resource budgeting for goal-directed behavior.

5.2 Interfaces: From Protocols to Contracts

Service-oriented abstractions organize interaction around protocols: specified message formats, sequences, and response semantics. A service exposes an API; clients invoke it according to the protocol; the service responds. Responsibility for correct usage lies with the client.

When agents become clients, this division of responsibility breaks down. A service designed for human users may assume requests arrive at human speeds, with human-level contextual awareness, and for human purposes; agent-initiated requests may violate all three assumptions [15].

The agency abstraction reveals the need for *contracts*, not protocols alone:

Capability specification. A protocol specifies *how* to invoke a service. A contract must additionally specify *what agents may do*: which actions are permitted, under what preconditions, with what resource implications. Such contracts connect to capability-based security [24], but extend it to encompass goals and intentions. An agent should be able to discover not just what APIs are available, but what behaviors they enable and constrain.

Action effects. Agents need to reason about the effects of their actions. A contract should specify preconditions (what must be true for the action to succeed), postconditions (what will be true after the action), and invariants (what the action preserves). Such specifications are familiar from design-by-contract in programming languages [54], but must be extended to handle uncertainty, partial observability, and the possibility that effects depend on environmental state the agent cannot fully observe.

Reversibility. Not all actions can be undone. Contracts should distinguish reversible actions (which can be rolled back if they prove mistaken) from irreversible ones (which commit the agent to consequences). An agent operating under uncertainty should prefer reversible actions when possible, and require stronger justification for irreversible ones.

Discovery and negotiation. In open environments, agents cannot know in advance what services are available or what their contracts specify. Agent-service interaction requires mechanisms for discovering available capabilities, negotiating terms of use, and adapting to services that may change over time.

Authority and delegation. A contract specifies what an agent may do; authority determines on behalf of whom, and in pursuit of which goals, it may do so (Section 3.4). An agent that delegates a subtask must be able to *attenuate* authority, granting the delegate a subset of its own permissions, budget, and scope, and to revoke it. Classical mechanisms for capability attenuation and least privilege [24, 63] provide starting points, and distributed systems have long implemented delegation in practice: early delegation architectures [34], proxy-credential delegation in Grid security infrastructure [31], scoped authorization delegation in research automation services [17], and, most recently, authenticated delegation of authority to AI agents [68]. The confused-deputy problem [40], in which a program exercises its own authority on behalf of a less-privileged requester, reappears whenever an agent acts on content from untrusted sources.

Trust boundaries. Contracts must also address the provenance of information that crosses interfaces. Because agents interpret data with the same machinery they use to interpret instructions, content flowing in through an interface can influence behavior as if it were a command (Section 4). Contracts should therefore distinguish channels that may legitimately shape the agent’s goals and plans from channels that supply data only, and runtimes should track the influence of untrusted inputs on action selection: an information-flow discipline extended from taint tracking to policy generation [23].

The end-to-end argument [62] observed that functionality placed in the network core cannot fully satisfy application requirements; applications must implement end-to-end checks themselves. For agentic systems, a parallel argument applies: the agent, as the entity with knowledge of its goals, bears ultimate responsibility for achieving them correctly. Contracts support this responsibility by making the terms of interaction explicit, but the agent must still reason about whether invoking a service serves its goals.

5.3 Correctness: From Functional to Behavioral

Traditional verification asks, in its paradigmatic form: does this program produce correct outputs for its inputs? This is *functional correctness*: correctness defined as a relation between inputs and outputs, evaluated at termination. Formal methods has, of course, long addressed reactive and nonterminating systems through temporal logic, refinement, and model checking; the point is not that behavioral specification is new, but that agency makes it the default case rather than the specialist one.

Agentic systems challenge the functional framing in several ways. They may not terminate; they produce behavior over time, not just final outputs; their “inputs” (goals, environmental observations) are not fully specified in advance; and their behavior may be correct in some sense even if they fail to achieve their goals, if the failure was reasonable given their information and resources.

The agency abstraction reveals the need for *behavioral correctness*: criteria that evaluate agent behavior over time, relative to goals and constraints.

Safety envelopes. Some states are unacceptable regardless of the agent’s goals. A safety envelope defines a set of states the agent must never enter: states involving irreversible harm, violation of ethical constraints, or catastrophic resource exhaustion. Verification for safety envelopes asks: can we guarantee, regardless of the policy the agent generates, that it will never enter unsafe states? Safety-envelope verification connects to work on safe reinforcement learning [33], constrained optimization, and formal verification of cyber-physical systems.

We can formalize safety envelopes as follows. Let S be the set of possible states, A the set of actions available through the agent’s environment interface E , and H the set of interaction histories (finite sequences of observations and actions). For a goal G , the policy generator Π produces, and may repeatedly revise, a policy $\pi : H \rightarrow A$, conditioned on G and memory M . A safety envelope is a set $S_{\text{safe}} \subseteq S$. The agent satisfies the safety envelope if every state reachable under any policy that Π may generate lies in S_{safe} .

For open-ended policy generators, verifying this property directly (quantifying over every policy the generator might produce) may be intractable, or achievable only vacuously. That observation points to a systems response rather than a purely logical one: enforce important invariants *outside* the generator, in a smaller trusted mechanism such as a reference monitor, capability system, sandbox, budget governor, or runtime verifier, so that unsafe actions cannot execute regardless of what the generator proposes. This is the classical move of separating policy from mechanism [53] and minimizing the trusted computing base, applied to agents: do not try to prove that the policy generator will never decide to do X ; construct the substrate so that X cannot occur without invariant Y being satisfied. The verification burden then falls on the small mechanism, not the open-ended generator.

Regret bounds. An agent that satisfies safety constraints may still perform poorly. Regret measures the gap between the agent’s performance and the performance of an optimal policy with full information. A regret bound guarantees that this gap is limited, even though the agent operates under uncertainty. Formally, if V^* is the value achieved by an optimal policy and V^π is the value achieved by the agent’s policy, regret is $V^* - V^\pi$. Regret bounds are well-studied in online learning and reinforcement learning [14, 43]; the challenge for agentic systems is establishing such bounds for policy generators operating in open environments.

Temporal properties. Some correctness requirements concern sequences of states over time. Safety properties (“nothing bad ever happens”) and liveness properties (“something good eventually happens”) from temporal logic apply to agent behavior. An agent should never simultaneously hold two conflicting commitments (safety); an agent should eventually respond to every high-priority request (liveness). Temporal logics such as LTL [59] and CTL [20] provide specification languages; the challenge is verification when the agent’s policy is not known in advance.

Satisficing criteria. Following Simon [66], an agent that cannot achieve optimality may still behave correctly by achieving “good enough” outcomes. A satisficing criterion defines a threshold: the agent is correct if

it achieves outcomes exceeding this threshold given its resource constraints. Unlike regret bounds (which compare to optimal), satisficing criteria define absolute standards adjusted for context.

Runtime verification. When static verification is intractable, a practical alternative is to verify at runtime: rather than proving that all possible policies satisfy correctness criteria, runtime monitors observe the agent’s behavior and intervene when violations are imminent. This approach requires efficient monitors, clear intervention policies, and mechanisms for the agent to adapt its behavior in response to interventions.

Causal tracing and attribution. Behavioral correctness criteria are useful only if violations can be diagnosed. When a constraint is violated after an extended decision sequence, attributing the violation requires reconstructing which decisions, observations, and interactions contributed to it. As in distributed systems, the relevant structure is a partial order of causally related events [52] rather than a linear trace: an action may depend on an earlier observation, a remembered conclusion, or a message from another agent. Research challenges include recording decision provenance efficiently, constructing partial-order explanations that humans can audit, and maintaining accountability over horizons in which the relevant context has long since left the agent’s working memory.

5.4 Oversight: From Commands to Attention Allocation

Human-computer interaction has largely been organized around a command model: the human issues commands, the computer executes them. Mixed-initiative interaction and supervisory control have long complicated this picture in specialized settings [42], but even these approaches generally assume episodic interaction with clear turn-taking.

When agents operate continuously and at scale, the command model fails. A human cannot issue commands to an agent that is always running, pursuing goals that may take hours or days to achieve. The human’s role shifts from commanding to *supervising*: monitoring ongoing behavior, intervening when necessary, and allocating limited attention across multiple agents [3] (Figure 8). When human attention cannot scale, automated supervisors may assist, raising the further question of how less capable supervisors, human or machine, can oversee more capable systems [11].

Attention as scarce resource. A human supervisor can monitor only a limited amount of agent behavior. With one agent, detailed supervision may be feasible; with hundreds, it is not. The oversight problem becomes one of attention allocation: which agents deserve scrutiny? Which behaviors warrant intervention? Attention allocation connects to work on anomaly detection, alarm management, and supervisory control in other domains.

Escalation policies. Not all decisions should require human involvement. Effective oversight requires clear escalation policies: criteria that determine when an agent should pause and request human input versus proceeding autonomously. Escalation policies must balance the cost of interruption (delay, attention drain) against the risk of autonomous action (irreversible errors, misaligned behavior). The agent itself may need to reason about whether to escalate, adding another dimension to its policy generation.

Transparency and summarization. When humans do supervise, they need to understand what agents are doing. Extended decision sequences cannot be presented in full; oversight interfaces must summarize goals, current state, recent actions, and pending decisions at levels of abstraction humans can quickly assess. The requirement is akin to explainable AI [38], but with the emphasis on temporal summarization of ongoing behavior rather than explanation of individual decisions.

Trust calibration. Humans form mental models of agent capabilities and adjust supervision accordingly; miscalibrated trust, whether over-trusting a fallible agent or under-trusting a reliable one, leads to oversight failures. Systems should support calibration by providing accurate information about agent capabilities, track record, and confidence.

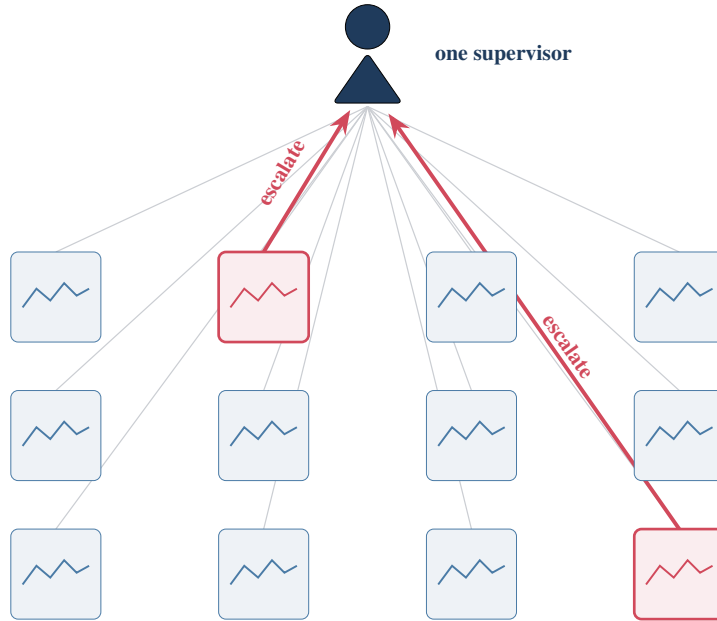


Figure 8: Oversight at scale. A single supervisor cannot review every decision of a fleet of continuously operating agents; oversight becomes a problem of allocating scarce human attention, with escalation as a first-class mechanism.

5.5 Coordination: From Scheduling to Negotiation

When multiple processes share resources, the operating system *schedules* them: allocating CPU time, memory, and I/O bandwidth according to policies that processes do not control. Market-based allocation and utility-aware schedulers exist, but as specialized designs rather than properties of the process abstraction itself: in the abstraction, processes are scheduled; they do not negotiate.

Multiple agents sharing resources present a different problem. Agents have goals; resource allocation affects goal achievement; agents may have preferences over allocations. Task allocation by negotiation has a classical precedent in the Contract Net Protocol [67], in which nodes announce tasks, bid, and award contracts. The shift from scheduling to *negotiation*, and from invocation to *delegation*, exposes new problems:

Delegation with accountability. An agent may delegate subtasks to other agents. Unlike procedure calls, delegation between agents involves entities with their own goals that may not perfectly align with the delegator’s [12]. Delegation requires specifying what the delegate should accomplish, what resources it may use, and how accountability flows back to the delegator. When a delegate fails or misbehaves, who is responsible? The compositional question is what must be preserved across delegation: authority, budget, constraints, provenance, accountability, deadlines, revocation rights. Defining delegation semantics that propagate these coherently is, we suggest, the central composition problem for an agent abstraction (Section 6.2).

Conflict resolution. Agents with different goals may compete for resources or take actions that interfere with each other. Unlike process scheduling, where the scheduler can simply timeslice, agent conflicts may involve goals that cannot all be satisfied. Conflict resolution requires mechanisms for agents to recognize conflicts, negotiate resolutions, and accept outcomes. When negotiation fails, adjudication may be required.

Emergent coordination. When many agents interact, system-level behaviors may emerge that no individual agent intended [58]: some beneficial, such as distributed problem-solving, others harmful, such as cascading

failures and resource exhaustion spirals. The 2010 Flash Crash, in which interacting automated traders amplified a market collapse, offers a precedent from an earlier generation of autonomous systems [50]. Managing emergence connects to work on complex systems, with the added complication that each participant is generating its own policy.

Multi-agent attribution. When something goes wrong in a multi-agent system, the failure may result from actions of multiple agents, each acting reasonably given its information [13]. Causal analysis must trace how agents' decisions influenced one another, and automatically identifying the responsible agent and the decisive error step remains unreliable even with frontier models [79]. This problem is akin to fault attribution in distributed systems [52], complicated by the goal-directed nature of agent behavior.

5.6 Persistence: From State to Memory

Processes have state; agents require something stronger: *memory*, i.e., persistent state with semantic consequences for future decisions, commitments, and identity. The distinction is not that process state is meaningless; state is often semantically rich and durable. The distinction is that agent memory is state interpreted as part of the agent's continuing identity: its beliefs, experience, commitments, and goals. We first consider the architecture of agent memory, and then problems (provenance, belief revision, forgetting, identity) that are consequences of taking that interpretation seriously. These concerns have counterparts in databases and distributed systems, but agency gives them a different significance: remembered information becomes part of the basis for future autonomous decisions, commitments, and actions.

Memory architecture. Cognitive science distinguishes working, episodic, semantic, and procedural memory [69]; agentic systems exhibit analogous distinctions, with context windows as limited working memory, retrieval augmentation as a form of semantic memory, and fine-tuned behaviors as procedural memory [70]. Which memory architectures support which agent capabilities is an open research direction.

Provenance and trust. Memories may originate from direct observation, from communication with other agents, or from external sources of varying reliability, and an agent that acts on a memory must weigh its provenance [9]: a first-hand observation warrants different treatment from a claim reported by an untrusted agent. This requirement parallels the control–data ambiguity of Section 4, but extended in time: adversarial content written into memory can redirect an agent's behavior long after it was encountered [19].

Belief revision. Agents must update their beliefs when evidence changes. Classical belief revision theory (the AGM framework [1]) provides foundations, but applying it to agents whose beliefs may be distributed across explicit memory, retrieved context, accumulated summaries, and model parameters raises new challenges. How should an agent revise beliefs that span such heterogeneous representations, only some of which are explicit propositions?

Forgetting. Not everything should be remembered forever: storage is finite, old information becomes obsolete, and privacy may require deletion. Agents need mechanisms that remove information gracefully without disrupting ongoing goals: machine unlearning [8], in a context where the forgotten information may have influenced current plans and commitments.

Active memory management. Memory need not be a passive store. An agent may need to decide what it does not know, seek clarification, reconcile conflicting observations, curate what it retains, and judge what information is worth acquiring at what cost. Memory management thereby becomes goal-directed behavior under bounded resources in its own right, extending the progression that organizes this section: from state, to memory, to active knowledge maintenance.

Identity and continuity. What makes an agent the same agent over time? If its memory is transferred to a new substrate, is it the same agent; if two copies are made, which is the original? These questions, familiar from the philosophy of personal identity [57], become practical when agents persist, migrate, and replicate, with direct implications for accountability, commitment, and trust [16].

6 Implications for Computer Science

The problems identified in this paper span subfield boundaries, but share a common root: the mismatch between autonomous, goal-directed computation and abstractions designed for externally controlled execution. This coherence suggests a cross-cutting research agenda for computer science.

6.1 A Unifying Research Agenda

Historically, new computational phenomena, from networking to machine learning, have driven integration across subfields. Agentic AI may play a similar integrative role. The failure modes of Table 2 do not respect subfield boundaries: a single runaway agent is simultaneously a systems problem, a language problem, a theory problem, and an interface problem, and the dimensions of Table 1 engage, in turn, the systems, programming languages, theory, and human–computer interaction communities.

These problems go beyond applying existing techniques to a new domain. They require rethinking foundational assumptions: that resource bounds are externally imposed, that control flow is specified in advance, that correctness is defined at termination, and that humans command and computers respond.

6.2 Does Agency Pass Its Own Test?

Section 2 defined an organizing abstraction as one that makes problems easy to state, tractable to solve, and natural to compose. We consider each criterion in turn.

Statement. Here the case is strongest. Agency exposes resource exhaustion, control–data ambiguity, memory poisoning, and supervision failure as manifestations of a common abstraction gap, autonomous policy generation under constraints, rather than as unrelated engineering problems. The six dimensions of Section 5 are easy to state only once the agent is the unit of analysis.

Tractability. Once the agent is an explicit system entity, a runtime can associate budgets, capabilities, authority, provenance, identity, commitments, and monitors with that entity, and enforce invariants around the policy generator through small trusted mechanisms (Section 5.3). This arrangement parallels how the process abstraction lets an operating system associate address spaces and resource limits with running programs. Here the evidence is a research agenda rather than an accomplished fact: the constituent mechanisms exist in classical systems work, but their integration around a policy generator remains to be built.

Composition. A mature abstraction has composition rules: processes compose through interprocess communication, objects through interfaces, services through protocols, and transactions through transactional semantics. Agents compose through *delegation*, and the composition rule is the open question identified in Section 5.5: what must be preserved when one agent delegates a goal to another—authority, budget, constraints, provenance, accountability, revocation? An agent abstraction will be mature when delegation has defined semantics for propagating these properties.

Figure 9 summarizes the resulting architecture: a computational entity defined by goals, a policy generator, and memory, exercising delegated authority within a substrate of mechanisms (budgets, contracts, provenance, monitors, oversight) that make generated behavior governable. Agency is thus a candidate abstraction: strongest today on the first criterion, and a concrete research program on the other two.

6.3 Research Priorities

The problems constituting this research program vary in urgency and tractability. We suggest a rough prioritization:

Near-term. Some problems affect currently deployed systems and are tractable extensions of existing techniques: execution environments with budget negotiation, runtime monitoring for behavioral properties, trusted enforcement mechanisms around policy generators, and oversight interfaces for attention allocation.

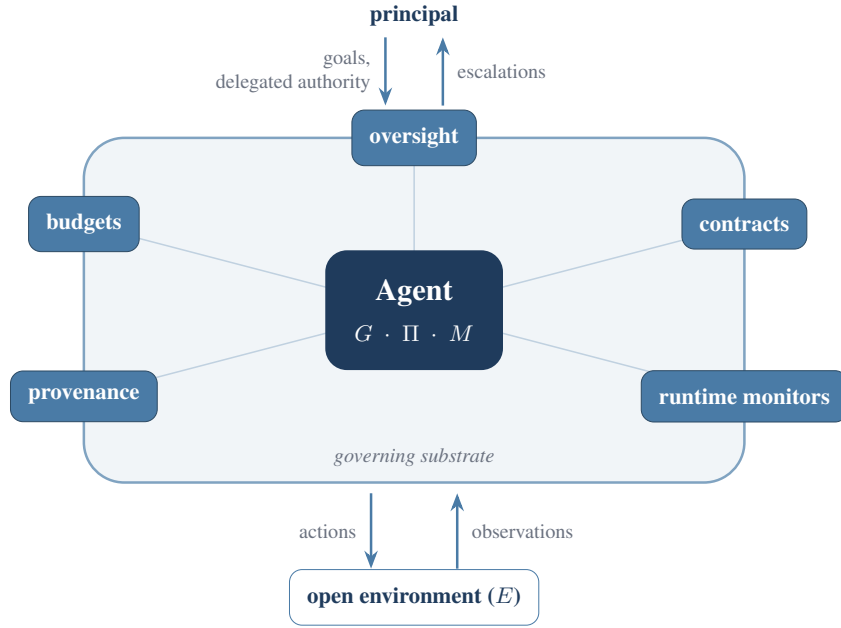


Figure 9: The agent abstraction in synthesis. The agent of Figure 4, now exercising authority delegated by its principal, embedded in a governing substrate whose mechanisms (budgets, contracts, provenance, runtime monitors, oversight) make generated behavior governable.

Medium-term. Other problems require more fundamental development but build on established foundations: programming models for constrained autonomy, formal foundations for satisficing correctness, and delegation semantics for authority and budgets.

Long-term. Some problems are not yet well enough understood to address directly: theoretical frameworks for emergent behavior in multi-agent systems, principled approaches to agent identity and continuity, and belief revision in neural systems. Progress here may require new conceptual frameworks rather than application of existing ones.

6.4 Technical-Governance Interactions

Technical and governance challenges interact in important ways. By *governance* we mean the external processes by which organizations and societies direct and hold accountable the behavior of deployed systems: internal policies, audits, certification, incident response, and regulation [65, 51]. Well-designed abstractions can make governance more tractable by providing hooks for auditing, constraint enforcement, and oversight. A runtime that tracks resource usage, logs decisions, and enforces budgets provides natural points for regulatory inspection. A contract language that makes agent capabilities explicit enables meaningful certification.

Conversely, poorly designed abstractions can make governance harder by obscuring behavior or creating fait accompli through deployed systems that resist modification.

Technical researchers therefore have responsibilities extending beyond narrow efficiency metrics. The abstractions we develop will shape what governance is possible. Designing abstractions with governance in mind (building in logging, auditability, constraint enforcement, and human override) is not an optional addition but an integral part of the research agenda.

6.5 Path Dependence in Infrastructure

Abstractions have architectural consequences: whichever primitives become conventional will determine where resource control, identity, authority, provenance, audit, and responsibility reside. If today’s agent frameworks establish conventions in which authority is represented only implicitly in prompts, memory carries no explicit provenance, and resource constraints are enforced only after actions are generated, those choices will become embedded in APIs, protocols, and ecosystems, much as the socket, the file descriptor, and the container embedded earlier design choices. Retrofitting first-class authority, budgets, or provenance into an installed base is far harder than designing them in. An explicit agency abstraction is an opportunity to make these properties first class while the conventions are still forming.

7 Related Work

Our thesis invites objections of the form “this is not new.” We organize related work around such objections, asking in each case what the earlier tradition established and what changes in the setting considered here.

“Agents are not new.” True: classical agent theory [61], multi-agent systems research [75], and belief-desire-intention (BDI) architectures [60] established agents that perceive, act, deliberate, and coordinate decades ago, and they supply much of the vocabulary used here. What these traditions did not do, because the enabling capability did not exist, was treat the agent as a *systems* abstraction with lifecycle, budget, authority, and oversight semantics.

“Autonomous systems are not new.” Autonomic computing [49], autonomous spacecraft [55], robotics, and adaptive control have long operated without step-by-step human direction, and our discussions of lifecycle management and graceful degradation draw on them. As argued in Section 3.2, however, these systems execute or adapt policies within structures fixed at design time; the regime addressed here is the runtime generation of behavior over incompletely modeled environments.

“Dynamic policy generation is not new.” Automated planning [35], program synthesis [37], and reinforcement learning all construct behavior rather than selecting it from a fixed repertoire. The distinction, developed in Section 3.2, lies at the action–model boundary: these methods presuppose explicit state and action models constructed before the system runs, whereas contemporary agents act where models are incomplete, discovered dynamically, or represented implicitly in a foundation model.

“Persistent distributed entities are not new.” The actor model [41] is perhaps the closest prior organizing abstraction: isolated, persistent entities communicating by message. But actor semantics specify how messages are handled, not how goals are formulated and pursued, and the model offers no account of budgets or authority. Services [27] and Grid virtual organizations [32] added distribution, interoperability, and cross-domain trust; early calls for Grid–agent convergence [30] articulated an agenda that this paper revisits (Section 2) in light of foundation models.

“The safety and oversight problems are not new.” Formal verification of reactive systems [59, 20], safe reinforcement learning [33], AI safety and alignment [2], and research on supervisory control and mixed-initiative interaction [42] all address closely related problems. We draw on each. What changes is the object and the scale: verification and supervision must now target a policy generator rather than a policy, and must extend across populations of continuously operating entities. Better abstractions will not solve alignment, but whatever alignment approaches prove effective will require them as infrastructure.

“Agent frameworks already supply this.” A rapidly growing ecosystem of frameworks supports the construction of LLM-based agents, including tool-use mechanisms [64], reasoning–action loops [77], and multi-agent conversation frameworks [76], alongside widely used engineering platforms. These efforts supply valuable mechanisms, but they are engineering frameworks rather than organizing abstractions: each makes local design decisions about memory, control, and tool invocation without providing the principled foundations

(lifecycle semantics, budget enforcement, behavioral correctness criteria, oversight interfaces) whose absence this paper documents. The closest conceptual neighbor is the proposal of cognitive architectures for language agents [70], which organizes the agent’s *internal* structure (memory, action space, decision-making) from a cognitive-science perspective; the present paper is complementary, treating the agent as a *systems* abstraction with external semantics: lifecycle, budgets, authority, and oversight. One goal of articulating agency as an abstraction is to give frameworks and architectures alike a common conceptual target.

8 Conclusion

We have argued that agency merits recognition as an organizing abstraction for computer science: a conceptual framework that structures how we build, reason about, and verify a class of computational systems.

The core claim is a reframing of what computation is organized around: *a process executes a program; an agent pursues a goal*. The process abstraction organizes computation around executing code specified in advance; the agency abstraction organizes computation around pursuing goals under bounded resources, by generating and revising courses of action under constraints of information, resources, authority, and time. This shift reveals a family of related problems, spanning execution environments, interface design, correctness criteria, human oversight, coordination, and persistence, that existing abstractions address only in specialized settings and that agency makes general and intrinsic.

The distinctive feature of contemporary agentic systems, built on foundation models, is *open-ended policy generation*: unlike prior autonomous systems that execute predefined policies or synthesize behavior within models specified in advance, LLM-based agents generate courses of action at runtime that did not exist before the goal was given. The same capability that makes them powerful makes them difficult to govern.

We have identified specific problems in each of the six dimensions of Table 1, sketched the new mechanisms that the agency abstraction requires, and argued that many important invariants are best enforced by small trusted mechanisms surrounding the policy generator rather than by verifying the generator itself.

Whether agency ultimately joins procedures, objects, processes, and services in the canon of durable organizing abstractions remains to be seen. The underlying gap, however, seems unambiguous: computer science has powerful abstractions for controlling the execution of computations specified in advance, and lacks comparably mature abstractions for controlling computations that decide at runtime what behavior to generate. Supplying them is the research program this paper proposes. Deployed agentic systems are already encountering the challenges we have identified; the program is not only necessary but urgent.

We do not claim that better abstractions alone will resolve the profound questions of governance, accountability, and social impact that agentic AI raises [6]. We do claim that well-designed abstractions are necessary infrastructure: they provide the hooks for auditing, the mechanisms for constraint enforcement, and the interfaces for oversight that governance requires. By engaging these technical problems directly, while maintaining appropriate humility about their scope, computer science can contribute to a more principled foundation for autonomous computational systems.

Acknowledgments

I thank the anonymous *Communications of the ACM* reviewers, whose comments substantially improved the article on which this expanded version is based, and the many colleagues with whom I have discussed these topics. This work was supported in part by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357, and the U.S. National Science Foundation, under grant NSF 2004894.

References

- [1] Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. “On the logic of theory change: Partial meet contraction and revision functions”. In: *Journal of Symbolic Logic* 50.2 (1985), pp. 510–530. DOI: [10.2307/2274239](https://doi.org/10.2307/2274239).
- [2] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, et al. “Concrete problems in AI safety”. In: *arXiv preprint arXiv:1606.06565* (2016). DOI: [10.48550/arXiv.1606.06565](https://doi.org/10.48550/arXiv.1606.06565).
- [3] Usman Anwar, Abulhair Saparov, Javier Rando, Daniel Paleka, et al. “Foundational challenges in assuring alignment and safety of large language models”. In: *Transactions on Machine Learning Research* (2024).
- [4] Joe Armstrong. “Erlang”. In: *Communications of the ACM* 53.9 (2010), pp. 68–75. DOI: [10.1145/1810891.1810910](https://doi.org/10.1145/1810891.1810910).
- [5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, et al. “Xen and the art of virtualization”. In: *19th ACM Symposium on Operating Systems Principles. SOSP ’03*. ACM, 2003, pp. 164–177. DOI: [10.1145/945445.945462](https://doi.org/10.1145/945445.945462).
- [6] Yoshua Bengio, Tegan Maharaj, Luke Ong, Stuart Russell, et al. “The Singapore Consensus on global AI safety research priorities”. In: *arXiv preprint arXiv:2506.20702* (2025). DOI: [10.48550/arXiv.2506.20702](https://doi.org/10.48550/arXiv.2506.20702).
- [7] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, et al. “On the opportunities and risks of foundation models”. In: *arXiv preprint arXiv:2108.07258* (2021). DOI: [10.48550/arXiv.2108.07258](https://doi.org/10.48550/arXiv.2108.07258).
- [8] Lucas Bourtole, Varun Chandrasekaran, Christopher A. Choquette-Choo, Hengrui Jia, et al. “Machine unlearning”. In: *IEEE Symposium on Security and Privacy*. 2021, pp. 141–159. DOI: [10.1109/SP40001.2021.00019](https://doi.org/10.1109/SP40001.2021.00019).
- [9] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. “Why and where: A characterization of data provenance”. In: *8th International Conference on Database Theory. ICDT’01*. 2001, pp. 316–330. DOI: [10.1007/3-540-44503-X_20](https://doi.org/10.1007/3-540-44503-X_20).
- [10] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, et al. “Borg, Omega, and Kubernetes”. In: *Communications of the ACM* 59.5 (2016), pp. 50–57. DOI: [10.1145/2890784](https://doi.org/10.1145/2890784).
- [11] Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, et al. “Weak-to-strong generalization: Eliciting strong capabilities with weak supervision”. In: *41st International Conference on Machine Learning*. Vol. 235. ICML’24. Oral presentation. PMLR, 2024, pp. 4971–5012.
- [12] Cristiano Castelfranchi and Rino Falcone. “Towards a theory of delegation for agent-based systems”. In: *Robotics and Autonomous Systems* 24.3–4 (1998), pp. 141–157. DOI: [10.1016/S0921-8890\(98\)00028-1](https://doi.org/10.1016/S0921-8890(98)00028-1).
- [13] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, et al. “Why do multi-agent LLM systems fail?” In: *39th Conference on Neural Information Processing Systems, Datasets and Benchmarks Track. NeurIPS’25*. 2025. DOI: [10.52202/085713-4082](https://doi.org/10.52202/085713-4082).
- [14] Nicolò Cesa-Bianchi and Gábor Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, 2006. DOI: [10.1017/CBO9780511546921](https://doi.org/10.1017/CBO9780511546921).
- [15] Alan Chan, Carson Ezell, Max Kaufmann, Kevin Wei, et al. “Visibility into AI agents”. In: *ACM Conference on Fairness, Accountability, and Transparency. FAccT ’24*. 2024. DOI: [10.1145/3630106.3658948](https://doi.org/10.1145/3630106.3658948).

- [16] Alan Chan, Noam Kolt, Peter Wills, Usman Anwar, et al. “IDs for AI systems”. In: *arXiv preprint arXiv:2406.12137* (2024). DOI: [10.48550/arXiv.2406.12137](https://doi.org/10.48550/arXiv.2406.12137).
- [17] Ryan Chard, Jim Pruyne, Kurt McKee, Josh Bryan, et al. “Globus automation services: Research process automation across the space–time continuum”. In: *Future Generation Computer Systems* 142 (2023), pp. 393–409. DOI: [10.1016/j.future.2023.01.010](https://doi.org/10.1016/j.future.2023.01.010).
- [18] Lingjiao Chen, Matei Zaharia, and James Zou. “FrugalGPT: How to use large language models while reducing cost and improving performance”. In: *arXiv preprint arXiv:2305.05176* (2023). DOI: [10.48550/arXiv.2305.05176](https://doi.org/10.48550/arXiv.2305.05176).
- [19] Zhaorun Chen, Zhen Xiang, Chaowei Xiao, Dawn Song, et al. “AgentPoison: Red-teaming LLM agents via poisoning memory or knowledge bases”. In: *Advances in Neural Information Processing Systems*. Vol. 37. NeurIPS’24. 2024. DOI: [10.52202/079017-4136](https://doi.org/10.52202/079017-4136).
- [20] Edmund M. Clarke and E. Allen Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Logics of Programs*. Vol. 131. Lecture Notes in Computer Science. Springer, 1982, pp. 52–71. DOI: [10.1007/BFb0025774](https://doi.org/10.1007/BFb0025774).
- [21] R. J. Creasy. “The origin of the VM/370 time-sharing system”. In: *IBM Journal of Research and Development* 25.5 (1981), pp. 483–490. DOI: [10.1147/rd.255.0483](https://doi.org/10.1147/rd.255.0483).
- [22] Ole-Johan Dahl and Kristen Nygaard. “SIMULA: An ALGOL-based simulation language”. In: *Communications of the ACM* 9.9 (1966), pp. 671–678. DOI: [10.1145/365813.365819](https://doi.org/10.1145/365813.365819).
- [23] Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, et al. “Defeating prompt injections by design”. In: *arXiv preprint arXiv:2503.18813* (2025). DOI: [10.48550/arXiv.2503.18813](https://doi.org/10.48550/arXiv.2503.18813).
- [24] Jack B. Dennis and Earl C. Van Horn. “Programming semantics for multiprogrammed computations”. In: *Communications of the ACM* 9.3 (1966), pp. 143–155. DOI: [10.1145/365230.365252](https://doi.org/10.1145/365230.365252).
- [25] Edsger W. Dijkstra. “Go to statement considered harmful”. In: *Communications of the ACM* 11.3 (1968), pp. 147–148. DOI: [10.1145/362929.362947](https://doi.org/10.1145/362929.362947).
- [26] Edsger W. Dijkstra. “The structure of the “THE”–multiprogramming system”. In: *Communications of the ACM* 11.5 (1968), pp. 341–346. DOI: [10.1145/363095.363143](https://doi.org/10.1145/363095.363143).
- [27] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.
- [28] Renato J. Figueiredo, Peter A. Dinda, and José A. B. Fortes. “A case for grid computing on virtual machines”. In: *23rd International Conference on Distributed Computing Systems. ICDCS’03*. 2003, pp. 550–559. DOI: [10.1109/ICDCS.2003.1203506](https://doi.org/10.1109/ICDCS.2003.1203506).
- [29] Ian Foster. “Service-oriented science”. In: *Science* 308.5723 (2005), pp. 814–817. DOI: [10.1126/science.1110411](https://doi.org/10.1126/science.1110411).
- [30] Ian Foster, Nicholas R. Jennings, and Carl Kesselman. “Brain meets brawn: Why Grid and agents need each other”. In: *3rd International Joint Conference on Autonomous Agents and Multiagent Systems. AAMAS ’04*. IEEE Computer Society, 2004, pp. 8–15. DOI: [10.65109/dqlj5483](https://doi.org/10.65109/dqlj5483).
- [31] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. “A security architecture for computational Grids”. In: *5th ACM Conference on Computer and Communications Security. CCS ’98*. 1998, pp. 83–92. DOI: [10.1145/288090.288111](https://doi.org/10.1145/288090.288111).
- [32] Ian Foster, Carl Kesselman, and Steven Tuecke. “The anatomy of the Grid: Enabling scalable virtual organizations”. In: *International Journal of High Performance Computing Applications* 15.3 (2001), pp. 200–222. DOI: [10.1177/109434200101500302](https://doi.org/10.1177/109434200101500302).

- [33] Javier García and Fernando Fernández. “A comprehensive survey on safe reinforcement learning”. In: *Journal of Machine Learning Research* 16 (2015), pp. 1437–1480. URL: <https://jmlr.org/papers/v16/garcia15a.html>.
- [34] Morrie Gasser and Ellen McDermott. “An architecture for practical delegation in a distributed system”. In: *IEEE Symposium on Security and Privacy*. 1990, pp. 20–30. DOI: [10.1109/RISP.1990.63835](https://doi.org/10.1109/RISP.1990.63835).
- [35] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.
- [36] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, et al. “Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection”. In: *16th ACM Workshop on Artificial Intelligence and Security (AISec)*. 2023, pp. 79–90. DOI: [10.1145/3605764.3623985](https://doi.org/10.1145/3605764.3623985).
- [37] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. “Program synthesis”. In: *Foundations and Trends in Programming Languages* 4.1–2 (2017), pp. 1–119. DOI: [10.1561/25000000010](https://doi.org/10.1561/25000000010).
- [38] David Gunning and David W. Aha. “DARPA’s explainable artificial intelligence (XAI) program”. In: *AI Magazine* 40.2 (2019), pp. 44–58. DOI: [10.1609/aimag.v40i2.2850](https://doi.org/10.1609/aimag.v40i2.2850).
- [39] Yaoqi Guo, Ying Xiao, Jie M. Zhang, Mark Harman, et al. “EET: Experience-driven early termination for cost-efficient software engineering agents”. In: *Findings of the Association for Computational Linguistics: ACL 2026*. 2026, pp. 33008–33024. DOI: [10.18653/v1/2026.findings-acl.1652](https://doi.org/10.18653/v1/2026.findings-acl.1652).
- [40] Norm Hardy. “The confused deputy (or why capabilities might have been invented)”. In: *ACM SIGOPS Operating Systems Review* 22.4 (1988), pp. 36–38. DOI: [10.1145/54289.871709](https://doi.org/10.1145/54289.871709).
- [41] Carl Hewitt, Peter Bishop, and Richard Steiger. “A universal modular ACTOR formalism for artificial intelligence”. In: *3rd International Joint Conference on Artificial Intelligence*. IJCAI’73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, 235–245.
- [42] Eric Horvitz. “Principles of mixed-initiative user interfaces”. In: *SIGCHI Conference on Human Factors in Computing Systems*. CHI ’99. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1999, 159–166. ISBN: 0201485591. DOI: [10.1145/302979.303030](https://doi.org/10.1145/302979.303030). URL: <https://doi.org/10.1145/302979.303030>.
- [43] Thomas Jaksch, Ronald Ortner, and Peter Auer. “Near-optimal regret bounds for reinforcement learning”. In: *Journal of Machine Learning Research* 11 (2010), pp. 1563–1600. URL: <https://jmlr.org/papers/v11/jaksch10a.html>.
- [44] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, et al. “SWE-bench: Can language models resolve real-world GitHub issues?” In: *12th International Conference on Learning Representations*. ICLR’24. Oral presentation. 2024.
- [45] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. “Advances in dataflow programming languages”. In: *ACM Computing Surveys* 36.1 (2004), pp. 1–34. DOI: [10.1145/1013208.1013209](https://doi.org/10.1145/1013208.1013209).
- [46] Alok Kamatar, J. Gregory Pauloski, Yadu Babuji, Ryan Chard, et al. “Empowering scientific workflows with federated agents”. In: *IEEE International Parallel and Distributed Processing Symposium*. IPDPS’26. 2026, pp. 1403–1418. DOI: [10.1109/IPDPS65963.2026.00114](https://doi.org/10.1109/IPDPS65963.2026.00114).
- [47] Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, et al. “AI agents that matter”. In: *arXiv preprint arXiv:2407.01502* (2024). DOI: [10.48550/arXiv.2407.01502](https://doi.org/10.48550/arXiv.2407.01502).

- [48] Alan Kay. “The early history of Smalltalk”. In: *ACM SIGPLAN Notices* 28.3 (1993), pp. 69–95. DOI: [10.1145/154766.155364](https://doi.org/10.1145/154766.155364).
- [49] Jeffrey O. Kephart and David M. Chess. “The vision of autonomic computing”. In: *Computer* 36.1 (2003), pp. 41–50. DOI: [10.1109/MC.2003.1160055](https://doi.org/10.1109/MC.2003.1160055).
- [50] Andrei Kirilenko, Albert S. Kyle, Mehrdad Samadi, and Tugkan Tuzun. “The Flash Crash: High-frequency trading in an electronic market”. In: *Journal of Finance* 72.3 (2017), pp. 967–998. DOI: [10.1111/jofi.12498](https://doi.org/10.1111/jofi.12498).
- [51] Noam Kolt. “Governing AI agents”. In: *Notre Dame Law Review* 101 (2026), p. 335. URL: <https://ndlawreview.org/governing-ai-agents/>.
- [52] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Communications of the ACM* 21.7 (1978), pp. 558–565. DOI: [10.1145/359545.359563](https://doi.org/10.1145/359545.359563).
- [53] Roy Levin, Ellis Cohen, William Corwin, Fred Pollack, et al. “Policy/mechanism separation in Hydra”. In: *5th ACM Symposium on Operating Systems Principles*. SOSP ’75. 1975, pp. 132–140. DOI: [10.1145/800213.806531](https://doi.org/10.1145/800213.806531).
- [54] Bertrand Meyer. “Applying “Design by Contract””. In: *IEEE Computer* 25.10 (1992), pp. 40–51. DOI: [10.1109/2.161279](https://doi.org/10.1109/2.161279).
- [55] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. “Remote Agent: To boldly go where no AI system has gone before”. In: *Artificial Intelligence* 103.1–2 (1998), pp. 5–47. DOI: [10.1016/S0004-3702\(98\)00068-X](https://doi.org/10.1016/S0004-3702(98)00068-X).
- [56] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. 2nd. O’Reilly Media, 2021.
- [57] Derek Parfit. *Reasons and Persons*. Oxford University Press, 1984.
- [58] Joon Sung Park, Joseph C. O’Brien, Carrie J. Cai, Meredith Ringel Morris, et al. “Generative agents: Interactive simulacra of human behavior”. In: *36th Annual ACM Symposium on User Interface Software and Technology*. UIST ’23. 2023. DOI: [10.1145/3586183.3606763](https://doi.org/10.1145/3586183.3606763).
- [59] Amir Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science*. 1977, pp. 46–57. DOI: [10.1109/SFCS.1977.32](https://doi.org/10.1109/SFCS.1977.32).
- [60] Anand S. Rao and Michael P. Georgeff. “BDI agents: From theory to practice”. In: *1st International Conference on Multi-Agent Systems*. ICMAS’95. 1995, pp. 312–319.
- [61] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th. Pearson, 2021.
- [62] J. H. Saltzer, D. P. Reed, and D. D. Clark. “End-to-end arguments in system design”. In: *ACM Transactions on Computer Systems* 2.4 (1984), pp. 277–288. DOI: [10.1145/357401.357402](https://doi.org/10.1145/357401.357402).
- [63] Jerome H. Saltzer and Michael D. Schroeder. “The protection of information in computer systems”. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. DOI: [10.1109/PROC.1975.9939](https://doi.org/10.1109/PROC.1975.9939).
- [64] Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, et al. “Toolformer: Language models can teach themselves to use tools”. In: *37th International Conference on Neural Information Processing Systems*. NIPS ’23. New Orleans, LA, USA: Curran Associates Inc., 2023. DOI: [10.52202/075280-2997](https://doi.org/10.52202/075280-2997).
- [65] Yonadav Shavit, Sandhini Agarwal, Miles Brundage, Steven Adler, et al. *Practices for governing agentic AI systems*. Tech. rep. OpenAI, 2023.
- [66] Herbert A. Simon. “A behavioral model of rational choice”. In: *Quarterly Journal of Economics* 69.1 (1955), pp. 99–118. DOI: [10.2307/1884852](https://doi.org/10.2307/1884852).

- [67] Reid G. Smith. “The Contract Net Protocol: High-level communication and control in a distributed problem solver”. In: *IEEE Transactions on Computers* C-29.12 (1980), pp. 1104–1113. DOI: [10.1109/TC.1980.1675516](https://doi.org/10.1109/TC.1980.1675516).
- [68] Tobin South, Samuele Marro, Thomas Hardjono, Robert Mahari, et al. “Authenticated delegation and authorized AI agents”. In: *arXiv preprint arXiv:2501.09674* (2025). DOI: [10.48550/arXiv.2501.09674](https://doi.org/10.48550/arXiv.2501.09674).
- [69] Larry R. Squire. “Memory systems of the brain: A brief history and current perspective”. In: *Neurobiology of Learning and Memory* 82 (2004), pp. 171–177. DOI: [10.1016/j.nlm.2004.06.005](https://doi.org/10.1016/j.nlm.2004.06.005).
- [70] Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. “Cognitive architectures for language agents”. In: *Transactions on Machine Learning Research* (2024).
- [71] David L. Tennenhouse and David J. Wetherall. “Towards an active network architecture”. In: *ACM SIGCOMM Computer Communication Review* 26.2 (1996), pp. 5–17. DOI: [10.1145/231699.231701](https://doi.org/10.1145/231699.231701).
- [72] Dan Tsafir, Yoav Etsion, and Dror G. Feitelson. “Backfilling using system-generated predictions rather than user runtime estimates”. In: *IEEE Transactions on Parallel and Distributed Systems* 18.6 (2007), pp. 789–803. DOI: [10.1109/TPDS.2007.70606](https://doi.org/10.1109/TPDS.2007.70606).
- [73] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. “Active messages: A mechanism for integrated communication and computation”. In: *19th Annual International Symposium on Computer Architecture*. ISCA’92. 1992, pp. 256–266. DOI: [10.1109/ISCA.1992.753322](https://doi.org/10.1109/ISCA.1992.753322).
- [74] Zexin Wang, Jingjing Li, Quan Zhou, Haotian Si, et al. “A survey on AgentOps: Categorization, challenges, and future directions”. In: *arXiv preprint arXiv:2508.02121* (2025). DOI: [10.48550/arXiv.2508.02121](https://doi.org/10.48550/arXiv.2508.02121).
- [75] Michael Wooldridge. *An Introduction to MultiAgent Systems*. 2nd. Wiley, 2009.
- [76] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, et al. “AutoGen: Enabling next-gen LLM applications via multi-agent conversation”. In: *arXiv preprint arXiv:2308.08155* (2023). DOI: [10.48550/arXiv.2308.08155](https://doi.org/10.48550/arXiv.2308.08155).
- [77] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, et al. “ReAct: Synergizing reasoning and acting in language models”. In: *11th International Conference on Learning Representations*. ICLR’23. Kigali, Rwanda, 2023.
- [78] Qing Ye and Jing Tan. “Agent contracts: A formal framework for resource-bounded autonomous AI systems”. In: *arXiv preprint arXiv:2601.08815* (2026). To appear at COINE 2026, co-located with AAMAS. DOI: [10.48550/arXiv.2601.08815](https://doi.org/10.48550/arXiv.2601.08815).
- [79] Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, et al. “Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems”. In: *42nd International Conference on Machine Learning*. Vol. 267. ICML’25. PMLR, 2025, pp. 76583–76599.
- [80] Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, et al. “Where LLM agents fail and how they can learn from failures”. In: *arXiv preprint arXiv:2509.25370* (2025). DOI: [10.48550/arXiv.2509.25370](https://doi.org/10.48550/arXiv.2509.25370).