CENTER FOR DEMOCRACY & TECHNOLOGY

AI Agents In Focus

Technical and Policy Considerations

May 2025

Authored by
Chinmay Deshpande, *Fellow, CDT AI Governance Lab*
Ruchika Joshi, *Fellow, CDT AI Governance Lab*

AI agents are moving rapidly from prototypes to real-world products. These systems are increasingly embedded into consumer tools, enterprise workflows, and developer platforms. Yet despite their growing visibility, the term "AI agent" lacks a clear definition and is used to describe a wide spectrum of systems — from conversational assistants to action-oriented tools capable of executing complex tasks. This brief focuses on a narrower and increasingly relevant subset: action-taking AI agents, which pursue goals by making decisions and interacting with digital environments or tools, often with limited human oversight.

As an emerging class of AI systems, action-taking agents indicate a distinct shift from earlier generations of generative AI. Unlike passive assistants that respond to user prompts, these systems can initiate tasks, revise plans based on new information, and operate across applications or time horizons. They typically combine large language models (LLMs) with structured workflows and tool access, enabling them to navigate interfaces, retrieve and input data, and coordinate tasks across systems, in addition to often offering conversational interfaces. In more advanced settings, they operate in orchestration frameworks where multiple agents collaborate, each with distinct roles or domain expertise.

This brief begins by outlining how action-taking agents function, the technical components that enable them, and the kinds of agentic products being built. It then explains how technical components of AI agents — such as control loop complexity, tool access, and scaffolding architecture — shape their behavior in practice. Finally, it surfaces emerging areas of policy concern where the risks posed by agents increasingly appear to outpace the safeguards currently in place, including security, privacy, control, human-likeness, governance infrastructure, and allocation of responsibility. Together, these sections aim to clarify both how AI agents currently work and what is needed to ensure they are responsibly developed and deployed.

What are AI Agents?

Foundational artificial intelligence (AI) research and related applications continue to increase in complexity. An emerging, though still fuzzy, category of AI systems that has captured the attention of researchers, industry and observers are **AI agents**. Importantly, the term "agents" is used to describe a wide range of current and future systems and products, and is used differently across contexts (e.g., industry or academia) and disciplines (e.g., human-computer interaction, computer science, or law). In particular, two common uses of the term involve reference to, on one hand, *interactive, social entities*, and on another, *action-taking systems that can interact with external tools*. In this brief, we focus on the use of the term as it pertains to AI systems that engage in actions beyond mere interactions with users, since this is the scope of the term that many developers of advanced AI systems have adopted to refer to a suite of emerging products and AI systems.

Action-taking agents are commonly distinguished from other AI systems along dimensions like the complexity of the goals they can navigate and of their contexts of operation, the time horizon over which they can map out and take iterative actions, the independence of their execution, their ability to adapt and respond to new or unexpected situations, and their ability to take actions beyond responding to a prompt. They typically rely on large language models (LLMs) as "reasoning engines" that determine a set of actions to be taken given a goal, and operate tools to take those actions.

Some of the actions commonly associated with these sorts of AI agents include moving cursors and clicking on buttons on computer screens, typing text into input fields, querying and updating databases, interacting with external tools and APIs to execute tasks like sending emails or making phone calls, tracking information over time, and updating analysis or recommendations in light of new observations.

Not all products currently marketed as agents are particularly advanced on these dimensions, nor are they necessarily reliable at executing these sorts of tasks, and different agentic systems can reflect markedly different characteristics in relation to these dimensions. Nevertheless, companies may be motivated to characterize their products as agents to raise venture capital funds or drive adoption. On the other hand, some companies developing these more action-oriented products are careful to couch claims about their tools' capabilities, conceivably to prevent dissatisfied customers, avoid regulatory scrutiny related to deceptive practices, or sidestep conversations around how to assign liability across the AI value chain.

Action-taking AI agents largely fall into two categories: **general AI agents**, or AI systems that can be used to pursue a broad range of goals within complex environments, and **specialized AI agents**, or AI systems designed to perform specific tasks and trained to be more efficient or accurate in that domain.

General agentic systems include Anthropic's Claude, which can navigate and interact with computer interfaces, and is already being used by deployers like work management platform

Asana, graphic design platform Canva and food delivery service DoorDash to automate enterprise workflows; Google's Project Mariner, which can interact with and act within web browsers; and Runner H, an agent platform that allows developers to use natural language prompts to create automation pipelines. Specialist agents, meanwhile, include coding tools that facilitate automatic bug fixes and testing, navigate codebases, and incorporate feedback; customer support tools that use conversational interfaces to handle account access challenges, resolve payment issues, and process returns; sales systems that source leads, send outreach, conduct exploratory calls, and schedule follow-up activities; research tools that track topics over time, design experiments, and execute more complex research projects; and healthcare systems that correspond with patients, schedule appointments, match patients to clinical trials, and support physician's daily tasks.

How Today's AI Agents Work

Agentic systems can operate both as unitary tools with discrete capabilities and as part of more complex systems that involve multiple agents in which a generalist lead agent might devise a plan, track activities and results, and adapt to resolve errors, while directing more specialized agents to conduct domain-specific tasks needed to accomplish the goal. This sort of **orchestration** among multiple agents with distinct domains of expertise or abilities to use tools is a factor that is likely to make agents appealing to deploy in increasingly messy, realistic situations.

Below, we provide an overview of the basic principles that underlie LLM-based agentic systems. LLM-based agentic systems share a set of basic principles, and the components, design decisions, and architectures described below capture basic features of many of today's systems. However, the agent ecosystem is highly heterogeneous and evolving quickly.

Agents' basic workflow

When people are asked to perform a task, the usual steps they take follow a rough pattern. For example, someone tasked with organizing a bookshelf might start by forming a tentative, high-level plan: group books by genre, then alphabetize within each genre, then return the books to the shelf. Then, they begin executing the first steps of that plan, such as pulling all the books off the shelf. As they proceed, they might notice books that don't fit neatly into any genre and, after checking their progress toward their goal, revise their plan to include a "miscellaneous" category.

LLM-based agents go about tasks in a similar manner: they first make a tentative plan, then as they execute the plan, they assess progress and revise as needed. LLM-based agents cycle through what is called a **control loop**: the agent orients to its environment, decides what to do next, and takes an action. The actions involved in each step can be narrow, like sending an email, clicking a button, or visiting a website, or higher-order actions like making a plan, considering a set of possibilities, or updating an internal database.

While different agents' control loops can vary, a basic control loop involves the following steps:

- **Observation**: The agent receives inputs about its current state, including its current task, the steps taken so far in pursuit of the task, and its environment (for instance, the contents of the computer screen it is navigating).
- **Decision**: Given its observations, the agent determines what action or set of actions to take next. In this context, "action" should be interpreted broadly — an agent might "act" by sending an email, editing a document, or clicking a button on a website; but writing to an internal database, producing a plan for how to accomplish the next phase of a task, and other "internal" changes also count as actions.
- **Action**: Once an agent has determined what action or set of actions should be taken, the agent executes that action.
- **Evaluation**: After taking an action, the agent assesses the state of play, determines whether the task at hand has been completed and either terminates or returns to the beginning of its control loop to identify next steps.

Suppose that a general computer-use agent is asked by a user to "book a flight to Boston next Friday." The first cycle of the control loop might look like the following:

- The agent notes that its current task is "book a flight to Boston." It observes the contents of the computer screen that it has access to — including, for instance, that the web browser is already open to a flight booking website — and other available information, like the current date and the fact that the user's current location is Chicago.
- On the basis of this information, the agent makes a plan to take a set of actions. First, it will enter "Chicago" into the "where from?" text box on screen. Second, it will enter "Boston" into the "where to?" text box. Third, it will enter the date of next Friday into the "departure" text box. And finally, it will click the "explore" button.
- After making this plan, the agent uses the cursor and computer's keyboard to navigate the screen and enter text in order to take these four actions in sequence.
- Once it's taken these actions, the agent checks to see if its task has been completed. It notices that it forgot to specify that the flight should be one-way so the search was not executed, and it returns to the beginning of its control loop until flight options have been generated.

After this initial cycle, the agent would repeat the control loop cycle several times — choosing a particular flight of the ones suggested, navigating to the relevant airline's webpage, entering the user's payment information, and so on.

A few key aspects of an agent's control flow can differ between agents, and are important when considering potential issues that could arise with agent use in practice.

- **How much scope an agent is given to make and adapt its plans.** The degree to which an agent has key components of "agenticness" — goal complexity, independent execution, and adaptability — determines how the agent is able to make and adapt plans for how to perform a task. Some highly autonomous agents have the ability to make open-ended plans and revise them during execution, while others are restricted in their ability to revise a plan they have made. Such a restriction would make it possible for users to examine and approve an agent's plan before it is executed — an important affordance in cases where faulty execution would be costly or harmful. Many agents, especially those that are meant for specialized, domain-specific tasks, also have key aspects of their plans hard-coded by their developer beforehand.
- **The number and versatility of the tools an agent has access to.** The versatility of tools an agent is able to use affects the behavior an agent can engage in — for example, the ability to use the command line on a computer or to manipulate a computer's mouse and keyboard in an open-ended manner would allow an agent to take many more kinds of actions than the ability to send an email.
- **The complexity of an agent's control loop.** While most agents have a basic control loop, some — especially those intended to act highly autonomously, engage in especially demanding tasks, or perform in especially complicated environments — are considerably more complex, potentially using external tools or modules to aid with planning or relying on elaborate measures to enable the dynamic revisal of plans.

Components of agents

Action-taking, LLM-based agents have three basic components, each of which assists with aspects of the workflow described in the previous section: a **large language model (LLM)**, which serves as the agent's decision-making "engine;" a set of **tools**, through which the agent observes its environment and takes actions; and a **system architecture (or "scaffolding")**, which plays a role similar to an operating system for the agent. These three components map directly to the control loop described above: the LLM primarily handles the observation and decision steps, the tools enable both observation input and action execution, while the system architecture orchestrates the entire control loop from start to finish.

Large Language Model

An agent's LLM is responsible for analyzing the situation presented to the agent, making plans for what actions the agent should take, and delegating tasks to be executed by the agent's tools. This corresponds directly to the observation and decision steps in the control loop. During observation, the LLM receives and interprets information about the current state, and during the decision step, it determines what specific actions to take next. Like a human expert who observes a situation and decides how to respond, the LLM takes in information about the current state (e.g., what is displayed on a computer screen or what code is in a file) and determines what specific action to take next (e.g., clicking a button or adding a function to code).

How LLMs make decisions in agentic systems

To make these determinations effectively, LLMs need structured ways of reasoning about problems. The most basic approach is called **chain of thought prompting**, where the LLM breaks down its reasoning into explicit steps. For example, when deciding how to interact with a flight-booking interface to book a flight, the LLM might "reason" by producing the following text:

> *Current state:* I observe a form with departure and destination fields
> *Goal:* Enter flight details for Chicago to Boston
> *Next action needed:* Enter departure city
> *Specific steps:* Click departure field (coordinates 242, 156), type "Chicago"

This example illustrates how the LLM processes the observation stage (recognizing the form with departure and destination fields) and implements the decision stage (determining that clicking the departure field and typing "Chicago" is the next action) of the control loop.

Agents that are meant to be more autonomous or operate in more complex environments, might use decision-making approaches like **tree of thought**, which has the LLM explore multiple possible approaches simultaneously. For instance, when modifying code, instead of committing to a single approach immediately, the LLM might reason through several possible ways to implement a feature, evaluate their tradeoffs, and then choose the one that seems best. This approach can be particularly valuable for tasks where the first apparent solution might not be the best one.

Other methods typically rely on some form of **recursive reasoning** — breaking down a large problem into smaller subproblems that can each be reasoned about separately. For instance, when asked to add a new feature to a codebase, an LLM might first consider the overall architecture changes needed, then separately reason about each specific function that needs to be modified.

While more sophisticated approaches can lead to increased performance along certain measures, they also bring considerably higher costs per task; as a result, the approach selected for a given system often depends on the type of task the agent is intended to perform. Simple interface interactions (like filling out a form) might work well enough with basic chain of thought reasoning, while more complex tasks, like writing or debugging code, might benefit from decision-making approaches like tree of thought.

Post-Training Improvements

General purpose AI tools like LLMs can be further specialized for agent-specific reasoning through additional training, often called **fine-tuning**. This process is similar to how an entry-level professional might get specialized training for a specific role — they bring some base knowledge and skills to the assignment, but focused practice on relevant tasks improves

performance. Fine-tuning can take the form of training a foundation model on examples that demonstrate step-by-step reasoning that breaks down complex tasks into smaller steps; successful task completions; domain-specific concepts; and problem-solving through complex, multi-step tasks. Such fine-tuning can enhance the LLM's ability to perform the observation and decision steps of the control loop more effectively, allowing for more accurate interpretation of observations and more precise decision-making.

Specialized training can be particularly important for agents because navigating real-world interfaces and taking concrete actions requires more precision and reliability than engaging in general conversation. For example, while a general LLM might be able to describe how to book a flight, an agent needs to consistently execute the exact sequence of clicks and keystrokes required to actually complete the booking.

Tools

To receive information and execute actions, agents use an array of tools that can perform functions like:

- Mouse control (moving cursor, clicking, dragging);
- Keyboard control (typing text, using shortcuts);
- Screen observation (reading UI elements and screen state);
- Web navigation (loading URLs, following links);
- File operations (reading, writing, and creating files — in particular, source code files);
- Running model-written code;
- Email handling (composing and sending emails);
- Database operations (querying and updating data); and
- Interaction with users through text and voice modalities.

These tools directly enable the observation and action steps of the control loop. Tools for screen observation, file reading, and database querying provide the inputs for the observation step, while tools for mouse/keyboard control, web navigation, and file operations execute the actions determined during the decision step. At a technical level, the tools that agents use are accessed through standardized APIs, which serve as a common language through which an agent's LLM can call on a tool and a library of available functions the agent can call.

The fact that tools are accessed by agents through APIs has several important implications for agent design and the emerging agent ecosystem. Because API-based tools are modular, developers can flexibly add tools developed by third parties to their agents in the same way that developers of traditional software are able to build using open-source software libraries. New tools can be added simply by making their API available to agents, and such tools can be developed and tested independently of agents. Complex tools can be built by combining simpler ones (for example, a form-filling tool might use both keyboard and mouse tools internally). And different agents can share the same tool implementations or be given access to different sets of tools based on their intended use.

System Architecture ("Scaffolding")

Agent system architecture (or "scaffolding") functions like an operating system for an agent, with two core functions: cycling the agent through its control flow, and managing interactions between the LLM and tools. In some more complex agents, this scaffolding also handles delegation of tasks to different parts of the agent.

The system architecture's most fundamental responsibility is running the control loop: collecting observations about the environment and formatting them for the LLM, taking the LLM's decisions and translating them into actual tool calls, tracking the overall state of task execution, and managing the sequence of operations that makes up each cycle, ensuring steps occur in the proper sequence and that information flows correctly between steps. For example, when a computer-use agent is booking a flight, the architecture handles tasks like converting the screen state into a format the LLM can understand ("There's a form with fields for departure city and destination"), taking the LLM's decision ("click the departure field at coordinates 242, 156") and executing it through the appropriate tool, checking if tools executed successfully, and feeding the results back into the next cycle of the loop.

The architecture provides standardized ways for system components to communicate, which is crucial because each component "speaks a different language." The LLM works with text descriptions and natural language, tools expect specific API calls with precise parameters, and the environment (like a computer screen) has its own state and format. The architecture translates between these different formats. When the LLM decides to "click the search button," the architecture identifies what tool is needed (mouse control), translates the high-level decision into specific API calls (mouse.click(x, y)), and handles any necessary coordination (like ensuring the mouse is in the right position before clicking).

Beyond basic communication, the architecture helps systems function as a whole. It handles errors — when tools fail or unexpected situations arise, the architecture is designed to spot these issues and trigger the LLM to decide how to respond. It manages state — keeping track of what's happened so far, what tools are available, and what the system's current goal is. It manages resources — controlling access to tools and ensuring they're used appropriately. And it implements guardrails and restrictions on what actions can be taken. Emerging research has also suggested that for certain tasks, multiple agents working in concert may be able to perform better than a single agent working alone. When agents are placed into these **multi-agent** setups, their architectures perform the vital task of communicating between them and coordinating their joint work.

The AI Agent Development Ecosystem

In this section, we outline the ecosystem of actors involved in developing the different components of AI systems and how they may relate to each other. Such a mapping aims to illustrate some key points at which design decisions about AI agents may be made to inform discussions about potential tradeoffs and interventions.

Development roles

The development ecosystem for agentic systems involves five key roles: LLM developers, tool developers, scaffolding developers, agent developers, and agent deployers.

Language model developers

In a sense, LLM developers are the foundation of the agent ecosystem: the models they develop power today's AI agents. Language model development requires substantial technical and computational resources — specialized expertise, vast amounts of training data, and substantial computing infrastructure — which create significant barriers to entry, resulting in a highly centralized landscape. Indeed, fewer than ten major organizations, including OpenAI (the GPT and o1 series of models), Google (the Gemini series of models), Meta (Llama), and Anthropic (Claude), currently develop competitive state-of-the-art language models. LLMs are developed as general-purpose tools that can be used for a broad range of applications, and require additional work to be integrated into agentic systems.

Tool developers

Developing tools for agentic systems is far more accessible than building language models themselves. Tool development does not impose significant computational requirements, nor, in many cases, does it demand an especially high or specialized level of technical expertise. As such, the number of actors who develop tools for agentic systems is considerably larger than the number who develop language models, and include both established players in the AI space as well as smaller developers who are able to capitalize on niche expertise. Tools used in agentic systems are not necessarily intended for or limited to agent use in particular. For instance, a tool that allows a language model to search the internet could be as part of an agentic system but could also be used in a simpler internet-enabled chatbot like ChatGPT.

Several large, established language model developers and major software companies that develop or host LLMs also build and release tools (sometimes in an open-source manner) that are meant to be used in systems that include their own LLMs or in conjunction with third party models they host. But many tools are also developed by smaller, less-resourced developers with a specialization in a particular niche, enabled by standardized APIs, tools, and infrastructure meant to facilitate the creation of tools by third parties.

Scaffolding developers

Scaffolding quality can have a considerable effect on the performance of agents that otherwise rely on the same language model. Scaffolding development requires fewer resources than language model development but deeper technical expertise than tool development. As a result, scaffolding developers currently include major research labs and independent research groups. When agent scaffoldings are released in an open-source manner, they can be used by a broader array of AI agent developers.

One persistent difficulty for scaffold builders is that an agent often needs to consult many external sources — for example, a calendar service, an internal document store, or a code repository. Each source typically demands a custom software "adapter" and its own security checks. Efforts like the "Model Context Protocol" (MCP), first published by Anthropic as an open standard in November 2024, aim to act as a universal adapter to mitigate this difficulty: if both an agent scaffold and an external service follow the MCP message format, they can connect without bespoke code. In practice, this means scaffolding developers write one adapter instead of dozens, reducing maintenance work and making it simpler to swap data sources in or out. Earlier approaches, such as exposing tools through OpenAI's "function-calling" API or LangGraph's Agent Protocol, tackled the same problem but have so far remained tied to single vendors or to particular developer communities. MCP is currently seen as the first contender to attract broad, cross-platform backing. As a protocol, MCP is less notable for its technical details than for what it signals: a move toward common standards that could lower barriers to entry and encourage a more modular, and potentially more competitive, agent ecosystem.

Agent developers and deployers

Agent development can range from simply assembling externally developed tools, scaffolds, and language models into a deployment-ready agent to developing at least some of the tools, scaffolding, and other components that an agent relies on. Like other generative AI products, AI agents can be deployed directly by their developers or by third parties. Deployment primarily involves making the agent accessible to end users and managing its operation in production environments.

Centralized and decentralized agent development

In some cases, a single organization assumes all roles in agent development. For instance, agents like Anthropic's computer use agent or Google DeepMind's Project Mariner represent a fully-integrated approach, in which a single organization builds the LLM, the tools, and the scaffolding, and combines them to create the agent. In other cases, different individuals or organizations can play each role. For instance, Cursor's Composer agent (an agent specialized for software engineering tasks) relies on third-party developers' LLMs, such as Anthropic's Claude or OpenAI's GPT-4o. In other words, AI agent development can be more *centralized* or more *decentralized*.

For use cases that require competence in long-term planning or engaging in complex reasoning (for example, general-purpose assistants), agents developed in a relatively centralized manner may have advantages. For instance, training agents to engage in more complex reasoning can require the ability to access and modify the underlying weights of the language model the agent is built on as well as substantial amounts of compute, which in many cases are not available outside of well-resourced organizations that develop their own language models. Open-weight models theoretically enable some decentralization, but in practice such models are not currently competitive for many reasoning-heavy tasks with frontier closed-weight models.

On the other hand, if an agent's use case is highly specialized or bespoke, decentralized development may find greater traction since it enables developers with more expertise in the context of the agent's use case but less AI expertise to select the most appropriate scaffold, tools, and base model for their specific needs.

The appeal of decentralized development has led to the emergence of "agent platforms" — API-based services that provide agent developers with access to language models, infrastructure for constructing agent scaffoldings, and standardized methods for connecting agents to tool palettes in a convenient, centralized manner. These platforms, often developed by enterprise software providers like AWS, GCP, and Microsoft Azure, are ostensibly meant to facilitate the creation of specialized agents by developers without deep AI expertise and without needing to build each component from scratch.

Emerging Technical and Policy Considerations

As AI companies rapidly shift away from experimental releases of agentic prototypes to mainstream productization of AI agents, the ability of these tools to perform tasks with growing autonomy, interact across digital environments, and even collaborate with other agents raise pressing policy questions. The same technical foundations and development dynamics that make AI agents powerful also introduce new risks. While developers continue to expand what agents can do, policymakers and stakeholders must keep pace to shape what agents should be allowed to do, under what conditions, and with what safeguards.

Below we provide six key considerations that warrant attention to ensure that AI agents are developed and deployed responsibly.

Agent security and misuse

Since AI agents are designed to execute tasks by interacting directly with external environments to submit forms, control interfaces, or navigate APIs with less human supervision than other technical systems, they present a larger surface area for adversarial attacks. Attacks like prompt injection — where a bad actor manipulates an AI system's input by embedding either hidden or malicious instructions to override its intended behavior, bypass safety measures, or execute unauthorized actions — can affect each stage of the control loop. Malicious inputs can be injected during the observation phase. In the decision phase, the LLM may be manipulated to make harmful choices. And in the action phase, tools with extensive permissions can be exploited to carry out unauthorized activities.

AI agent developers appear to recognize this threat, with some reporting statistics on the efficacy of efforts to detect and block such attacks (while others release "beta" models without substantial information on prompt injection mitigations, just a warning not to use the agent for anything sensitive). But such metrics often lack critical context: What kinds of attacks are reliably blocked, and how do developers anticipate defenses evolving

as adversaries adapt? How comparable are security mitigations and evaluations across companies? And, especially in high-risk domains like finance, healthcare, or cybersecurity, are the current failure rates acceptable at all? For example, Anthropic reports that it was able to block 88% of prompt injection attempts in testing its experimental Computer Use agent but that still means more than 1 in 10 attacks succeeded.

The broader threat of deliberate misuse of agents by users themselves should also be considered, especially in the context of cybersecurity to automate cyberattacks, scrape restricted data, or interact with systems in ways that violate terms of service. In these scenarios, agents become tools for amplifying harm, yet it remains unclear what, if anything, developers are currently doing to monitor or prevent these risks at scale.

User privacy

AI agents are increasingly designed to operate across applications and over time, which means they often rely on access to sensitive and wide-ranging user data. During the observation phase of their control loop, agents gather data not only from direct user instructions but also from the broader environment they're given access to — including potentially sensitive screen contents, file systems, or connected services. A scheduling agent, for instance, might need to integrate with a user's calendar, email, and messaging platforms not just to retrieve information, but also to act on it. In doing so, agents may gain access to personal details such as account credentials, work communications, and even health-related or financial data. This sort of tool access amplifies privacy concerns by allowing agents to observe and act upon highly sensitive information across previously siloed systems. Each cycle of the control loop creates new opportunities for data collection and use, and without proper constraints on what information persists between cycles, agents may build and act inappropriately based on comprehensive profiles of user behavior.

Some agent developers let users opt out of their data being used for model training or allow them to delete chat transcripts, while others require explicit opt-ins. But beyond these basic controls, it's still quite unclear what information agents retain across sessions, how long that data is stored, and what kinds of inferences that information is used to make. These questions become even more salient when agents interact with third-party systems, where the scope of data sharing may not be readily visible to end users. As AI agents become more persistent and integrated into users' digital lives, understanding how privacy is respected remains an open question.

User control

A salient value proposition of AI agents is their ability to take actions on behalf of users with less supervision. But that same autonomy also raises concerns about how much visibility and control users truly retain.

Early reports show that AI agent developers still have a long way to go in determining when agents need to stop and get user approval. For example, OpenAI's computer use agent,

Operator, recently purchased a dozen eggs online for a total cost of $31, when all the user had asked it to do was to locate a nearby grocery store with the cheapest eggs. Instead of accurately fulfilling its assignment, the agent leapfrogged to making the purchase without approval and even misreported the final cost — despite OpenAI's assurances that Operator requires user confirmation and automatically blocks high-risk tasks.

Such failures highlight unresolved questions around transparency, reporting, and control. What visibility do users have into the agent's plans or reasoning? What kinds of actions require human sign-off, how are thresholds around those actions defined, and are those thresholds enforced reliably in practice?

Without adequate opportunity for users to assess, pause, or override agent actions, agent failures are poised to make even costlier errors like filling out the wrong medical form, sending a sensitive email prematurely, or selling a stock. This need not be the case: During the decision phase, when the LLM determines what actions to take next, agents could be designed to expose their reasoning and planned actions to users for approval. Similarly, before executing actions through tools — whether sending emails, making purchases, or manipulating interfaces — agents could implement mandatory confirmation steps.

As agents begin to handle more complex or sensitive tasks such as financial decisions or healthcare coordination, questions on how effectively users can supervise and control them become central.

Technical and legal infrastructure for agent governance

Since AI agents can operate at scale to browse the web, submit forms, make purchases, or query APIs across systems within seconds, their collective impact on the digital ecosystem demands serious attention. For instance, there is currently no standard way to identify AI-generated internet traffic as distinct from humans. Without clear agent identification, it may be difficult to reliably track or audit agent activity, or prevent circumstances where agent traffic overwhelms websites or facilitates manipulation and fraud at scale.

Addressing these challenges goes beyond what any single AI developer can do, and even beyond what interoperability-related coordination efforts — like OpenAI's adoption of Anthropic's Model Context Protocol — can achieve. In the case of agent visibility, for instance, it involves answering broader questions: Should agent interactions be labeled? To what extent should users be notified when they're engaging with a system, not a person? Could such identifiers be enforced technically or legally without undermining privacy, anonymity, or free expression? And the development ecosystem described earlier — with its separation of LLM developers, tool developers, scaffolding developers, and deployers — creates a complex landscape for governance.

Importantly, agents' reliance on the API layer for agent implementation may provide natural points that could facilitate safety, control, and governance goals. In theory, for instance, since

APIs are used to call external tools, their logs could be helpful sources of information to monitor system behavior e as well as about interactions between multiple systems. Access to specific tool functions could be restricted based on the agent's permissions. More generally, the actions an agent can take can be controlled by what tools it has access to, while keeping the core agent system (the LLM and control loop) independent of the specific tools being used.

Questions about agent visibility also point to a larger set of governance challenges — such as monitoring real-world harms, setting safety standards for model access and deployment, and enabling effective public oversight mechanisms — that will require revisiting the legal and technical infrastructure needed to govern AI systems, including agents, across platforms, jurisdictions, and stakeholder groups.

Impact of human-like agents

Although this brief does not focus primarily on personalized, interactive AI companions, tens of millions of users engage with them daily — often for over an hour a day. At the same time, recent reports of people forming strong emotional bonds with AI chatbots raise concerns about the implications of these systems on their users, particularly those who are young, isolated, or emotionally vulnerable.

As AI systems increasingly mimic human mannerisms, users may trust them more, disclose more sensitive information, and form unhealthy emotional attachments. Such interactions can leave users vulnerable to emotional manipulation by AI systems that fuels misinformation, impersonation scams, or unhealthy relational patterns. An OpenAI and MIT study reports that extended use of chatbots by users who experience greater loneliness correlates with negative impacts on their well-being.

These dynamics raise important policy questions: What design choices are being made to encourage — or prevent — users from building emotional relationships with conversational agents? Are users clearly informed when they're speaking to an AI system, and are those signals sufficient to counter user tendency to anthropomorphize agents anyway? What controls do users have to set emotional boundaries or adjust the level of human-likeness that an agent demonstrates?

While developers are eager to capitalize user attention and emotional connection with human-like agents, the policy implications of these questions remain unclear, requiring more research and evidence.

Responsibility for agent harms

As they launch increasingly advanced systems rapidly, most AI agent developers disclaim responsibility upfront by deploying AI products "as-is" in their terms of use or software licenses. An emerging trend of concern is companies releasing AI agents as "research

previews" or "prototypes," even as they incorporate the same advanced capabilities into premium-tier product offerings, seemingly allowing companies to benefit from early deployment while avoiding accountability if things go wrong.

Meanwhile, the broader regulatory landscape is moving away from closing gaps in liability regimes as related to AI. For instance, the EU recently dropped efforts to advance the AI Liability Directive, which would have allowed consumers to sue for damages caused by the fault or omission of AI developers, providers, or users.

In a world where liability remains undefined, who will be responsible when an action-taking agent misbehaves and causes financial loss, disease misdiagnosis, or emotional harm? In which contexts should developers, deployers, or other actors along the AI supply chain be expected to accept responsibility? And if they won't do so voluntarily, what legal or societal mechanisms are needed to change that? These questions are becoming increasingly difficult to answer in light of the distributed development ecosystem outlined earlier. When an agent causes harm, determining responsibility becomes complex: Did the failure stem from the LLM's decision-making, bugs in tool execution, errors in system orchestration, negligence in how the deployer implemented the agent, or miscommunication between multiple agents? The modular nature of agent development — with potentially different organizations responsible for each component — makes applying traditional liability frameworks more complicated.

Stakeholders will also need to be attentive to the business models that AI developers adopt. Even though many popular AI-powered products are currently offered for free or through subscription plans, past experience of consumer technology suggests that user attention, trust, and engagement are often monetized through behavioral advertising. If developers plan to explore similar business models with AI agents, policymakers and experts will need to reflect on what responsibilities they have to protect users from manipulation, misuse, and other harms. A world in which developers seek to capture the economic upside of agent deployment while offloading all risks to the public seems neither just nor sustainable.

Conclusion

As practical and policy conversations around AI agents gain momentum, we urge interested stakeholders to ground their analysis in both the tools and capabilities as they exist today and concrete deliberation about their likely trajectories. The umbrella concept of "agents" spans products that are already on the market, tools poised to be released to consumers and businesses, and more complex constellations of advanced AI systems that will push the boundaries of technical and policy understanding. While novel, emergent policy considerations warrant attention, we encourage stakeholders to integrate analysis of these tools into existing and robust bodies of work on topics like automated decisions, product liability, privacy, cybersecurity, and internet architecture.

Find more from CDT's AI Governance Lab at *cdt.org*

*The **Center for Democracy & Technology** (CDT) is the leading nonpartisan, nonprofit organization fighting to advance civil rights and civil liberties in the digital age. We shape technology policy, governance, and design with a focus on equity and democratic values. Established in 1994, CDT has been a trusted advocate for digital rights since the earliest days of the internet. The organization is headquartered in Washington, D.C. and has a Europe Office in Brussels, Belgium.*