7. Enterprise-Grade Agents

NVIDIA CEO Jensen Huang sees enterprises with "a couple of hundred million digital agents, intelligent agents." Microsoft CEO Satya Nadella says that "agents will replace all software." For sure, there could be a valid debate about the timeframe when we have millions of agents, and whether they will replace traditional software, but there is probably no longer any debate about the fact that a wave of agents is coming.

Clearly enterprises need to be ready. But there is much that could be embedded into an agent architecture that would make it easy to integrate into an enterprise. And if we do this elegantly, we achieve what we call *enterprise-grade* agents - agents that are reliable, explainable, scalable, discoverable, secure, observable, and operable.

MicroAgents (Microservice Agents)

Today's AI workflows are often monolithic, typically implemented as a single Python program running as a single operating system process. This means the LLM handles everything from input parsing, processing logic, through to final output within the same environment, limiting flexibility

and making it challenging to integrate with external services or systems that exist outside the monolithic structure.

So, the fundamental drawback of these monolithic AI workflows is their inability to natively collaborate beyond their own boundaries. Since the single Python program represents the entire workflow, any new functionality—such as a specialized service or a domain-specific tool—must be integrated directly into this codebase. Over time, this leads to brittle designs that are difficult to scale or maintain and hinders the ability to distribute tasks among specialized components.

In contrast, agents (along with tools, but for now we focus on agents) function as a *quantum*, the smallest meaningful unit in an agent-based ecosystem. Agents combine three core elements to perform tasks effectively:

- Large Language Model (the “brain”)
- Suite of tools (the “limbs”)
- Execution framework (for task planning and execution)

This combination enables agents to parse incoming requests, plan out how to fulfill them, and then orchestrate the necessary actions using either internal tools, collaborating agents, or external services. In this way, agents can tackle complex tasks that would otherwise overwhelm a

monolithic AI.

Each agent is packaged as a deployable unit, bundling together its LLM, tools, and execution logic. This packaging allows agents to be managed, scaled, and updated independently. There are several ways to deploy agents with popular frameworks, including microservices, Ray programs, or Dask programs, each offering different approaches to distributed execution and resource management. While Ray and Dask are more specialized for large-scale analytics, microservices have the advantage of being well-established in almost all enterprise environments, making them a natural fit for many organizations.

Microservices make a strong implementation choice because they build on decades of operational experience. To distinguish microservice-based agents from the current monolithic architecture, we are offering a new term: *microagent*.

MicroAgents, as shown in Figure 7-1, are typically packaged as containers and can be orchestrated across distributed environments using platforms like Kubernetes or modern cloud platforms. This model allows each agent to run independently, scale horizontally, and restart gracefully in case of failure—features that align closely with the needs of a robust AI ecosystem.

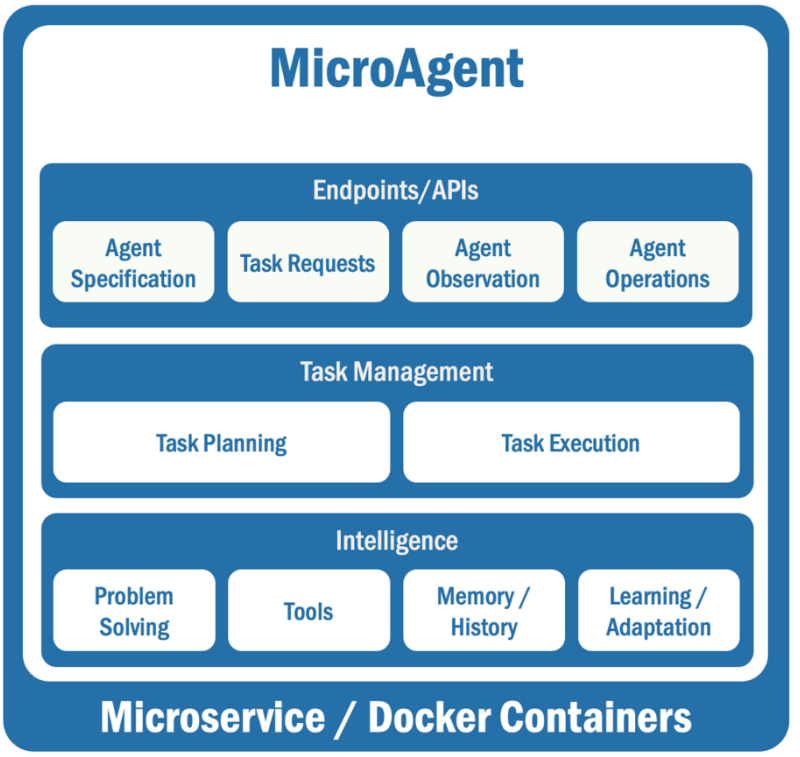


Figure 7-1. MicroAgents

By adopting microservices, organizations can leverage existing best practices for deployment pipelines, monitoring, and logging. Effectively, agents should just become another type of microservice within the broader infrastructure, benefiting from well-understood patterns for resilience, load balancing, and fault tolerance. This not only simplifies the technical aspects of agent deployment but also accelerates time-to-market, since teams can rely on familiar tooling and processes.

One of the major strengths of microservices is their mature security ecosystem. Decades of practice have produced industry-standard mechanisms such as mutual TLS (mTLS) for secure communication between services and OAuth2 for delegated authorization. These protocols can be readily

applied to autonomous agents, ensuring they can securely communicate with each other and with external resources without reinventing the wheel.

Beyond deployment, an agent ecosystem—what we call *agentic mesh*—provides a higher-level framework for discovery and collaboration. Agents register their capabilities and availability within a shared registry, enabling other agents to find and invoke them dynamically. This structure makes it easy to expand or update the ecosystem: as new agents are added, the registry immediately reflects their capabilities. Agentic mesh and most of these agent ecosystem capabilities are explained in Chapter 8.

The fact that each agent is self-contained allows for iterative improvements over time. Developers can upgrade an agent's LLM or expand its tools without disrupting the rest of the ecosystem. And if an error occurs in one agent, it does not bring down the entire system, and troubleshooting is localized to the responsible component.

Importantly, this multi-agent approach fosters collaboration that is difficult to achieve in a monolithic setup. When tasks require specialized skills or domain-specific knowledge, an agent can simply delegate sub-tasks to another agent designed for that role. This model provides a more flexible, scalable, and maintainable approach to AI-driven processes.

Last, and perhaps of significant importance to enterprise operations managers, the use of a microservices foundation in MicroAgents lets enterprises rely on well-established DevSecOps practices—such as continuous integration, container orchestration, and automated security checks—to ensure smooth and secure operation.

Agent Security

Enterprise systems, and by extension enterprise agents, require mature security capabilities: authentication, authorization, identity management, and secure communications. These are foundational to any system operating within sensitive, regulated, or high-value environments.

However, many current agent frameworks neglect these essential elements. Support for standards such as mTLS, OAuth2, and integration with enterprise identity book of record (IBoR) systems is rare or nonexistent. And, unfortunately, developers are often left to patch together ad hoc solutions, which introduces inconsistency, duplication of effort, and potentially dangerous vulnerabilities.

The absence of coherent identity lifecycle management, enforceable access controls, or encrypted transport

between agents leads to material risk by allowing impersonation, data leaks, or unauthorized tool invocation. Addressing these concerns requires adopting security frameworks that have already been proven in distributed enterprise architectures.

microagents, built upon microservices, offer a mature foundation on which to build secure, agent-based systems. This section looks at specific aspects of security that are available to MicroAgents:

- Basic microservices security
- Container security
- Kubernetes security

Basic Microservices Security

mTLS is a communication protocol that enforces two-way authentication between clients and servers. In the context of agents, mTLS ensures that both parties in an interaction can verify each other's identity via digital certificates. Once the handshake is complete, all communication is encrypted end-to-end. This is essential for agent-to-agent communication, especially when agents are distributed across networks. mTLS prevents man-in-the-middle attacks and unauthorized service invocation, offering a robust default posture for

sensitive, real-time agent coordination.

OAuth2 is a widely adopted authorization framework that allows systems to control access to resources based on defined permissions and scopes. It separates the concept of identity (authentication) from the notion of access (authorization), making it ideal for managing complex multi-agent interactions. With OAuth2, agents can request and grant access to tools or data without revealing passwords or persistent credentials. Access tokens can be short-lived and scoped, reducing the attack surface and enabling fine-grained governance over what an agent can do on behalf of another service or user.

Integration with an IBoR system is critical to enforce enterprise-level identity controls. A IBoR maintains a canonical registry of identities, including users, services, and devices, and manages their lifecycle across provisioning, updates, and deactivation. When agents interact with tools or with other agents, their identities must be verified against this system. This allows enterprises to apply the same governance policies to agents as they would to human users or backend services, ensuring alignment with compliance and audit standards.

Secrets management is another key requirement for secure agent operations. Agents must often access APIs, databases, or other tools that require credentials or API keys.

Hardcoding these values into code or environment variables is insecure. A secrets management system (many cloud-native, proprietary, and open source solutions are readily available) can issue, rotate, and revoke credentials as needed. Each MicroAgent can be configured to fetch its secrets securely at runtime, ensuring that credentials are not exposed or leaked.

Audit logging and observability are equally essential. In a distributed agent system, understanding who did what, when, and why is crucial for compliance and forensic investigation. Each agent microservice should log access attempts, tool invocations, authentication events, and errors. Centralized logging systems can correlate these events across agents and alert on anomalous behavior. Without this, unauthorized actions could go unnoticed.

Security policies must also account for dynamic trust boundaries. Agents may need to operate across tenants, virtual networks, or cloud regions. Network segmentation, zero-trust policies, and context-aware access (based on IP, location, or device) become important controls.

Container Security

Of particular note is that the microservices foundation of microagents makes containerization - via common tools such as Docker - not only practical but relatively easy. Using

containers, each agent can be isolated and equipped with its own security boundary. This design ensures that agents are not only logically separated but also technically isolated at the runtime and network level. Containerization provides resource limits and process separation such that any compromise or failure in one agent is less likely to impact others, enhancing the overall fault tolerance and security of the system.

This isolation also simplifies the application of security policies. Each container can be configured with its own network policies, storage encryption, and runtime security profiles. For example, containers can be prevented from running as root, restricted to specific system calls, or monitored with common enterprise intrusion-detection tools. Because these policies are applied per container, they allow for differentiated and quite granular security postures based on the function and sensitivity of each agent.

Container boundaries facilitate continuous deployment and patching, which are essential for maintaining a secure environment. Since MicroAgents are deployed independently, security updates can be rolled out to individual components without requiring system-wide downtime. This modularity also aligns well with DevSecOps practices, where security is integrated into the development and deployment pipeline.

By treating each agent as an isolated, policy-enforced microservice, the MicroAgent model makes enterprise-grade security not just possible but operationally manageable.

Kubernetes Security

Kubernetes is a container orchestration system that is used to run and manage containers at-scale. As such, it is an ideal run-time environment for containerized MicroAgents.

Kubernetes provides native support for certificate rotation, identity injection, secrets mounting, and sidecar-based proxies for secure communication. This modularization enables teams to independently apply security updates and tailor policies without affecting the broader system.

Kubernetes enhances agent security through several built-in features that standardize and automate critical controls. Role-Based Access Control (RBAC) is enforced at the API server level, allowing fine-grained permission assignments for users, services, and agents. Kubernetes service accounts provide identities for pods, and policies define what each pod can access within the cluster. This ensures that agents only access resources explicitly authorized by administrators, reducing lateral movement risks within the system.

Another important capability is Kubernetes' secrets

management. Kubernetes allows the secure and safe use of secrets such as API tokens, passwords, and encryption keys into containers through mounted volumes or environment variables. These secrets are stored in etcd, and access to them is governed by RBAC. While Kubernetes' native secrets management can be integrated with external systems for enhanced encryption, it provides a baseline mechanism for secure credential distribution to agents without embedding secrets in code or images.

Kubernetes also supports network security policies and pod-level isolation (*pods* are the basic scheduling unit in Kubernetes, and a container resides in a pod). Network policies can be used to restrict communication paths between agents, ensuring that only explicitly permitted services can interact. This mitigates the risk of unauthorized agent-to-agent communication. Additionally, the use of service meshes like Istio with Kubernetes enables mTLS between services, traffic encryption, telemetry collection, and policy enforcement. These capabilities strengthen the security posture of each agent without requiring substantial custom implementation effort.

Adopting MicroAgents is not merely an engineering convenience. Rather, we think it is a prerequisite for robust, scalable, and auditable security for the agent ecosystem. Decades of enterprise security best practices can be applied

to agents if they follow the microservice model. This makes agents fit for regulated industries, public sector deployments, and any domain where enhanced levels of security is mandatory.

Agent Reliability

Experience has shown that as LLM capabilities increase, user expectations expand to take full advantage of these new capabilities. But as we race forward, there is a constant balance – every new LLM brings reliability increases (such as reduced inaccuracies and hallucinations), but we ask for more, and sure enough reliability drops. The fundamental truth, at least today, is despite incredible advances, the current crop of LLMs are not up to handling complex workloads, let alone workloads that require a high degree of accuracy. Effectively, this relegates LLMs to simple workloads limits their use in customer-facing situations, and restricts their use in regulated industries such as financial services and healthcare.

Is there a better approach?

Our belief is that today users simply ask far too much of LLMs. No sooner do we get a new, better, LLM and we ask them to do more, and once again we are confronted with

errors and inaccuracies. As the saying goes: rinse and repeat.

Here is the crux of the problem: LLMs are probabilistic, but probabilities multiply and grow – sometimes catastrophically. So, when asked to do small things, LLMs tend to be extremely reliable and accurate. But when asking them to do big things cascades small errors into exponentially growing errors. In a way, small is better.

The Reliability Problem

Anyone who has worked with LLMs knows they generate inaccurate information and hallucinations. This tendency appears manageable for simple, concise tasks, where the model's responses are usually reliable, repeatable, and accurate, but degrades materially as more content is requested (or provided).

Consider LLM-enabled software development which happens to be one of the most common and practical use cases for LLMs today. Developers who have used these models find that short programs or single-function scripts can be generated reliably (for example, by running without requiring human modifications) and with few errors. Yet when the model is asked to produce something more complex – say, a multi-file software project, or a larger, more complex component – its reliability and accuracy drops

sharply. In many instances, these longer code outputs either fail to run on the first attempt or require manual redesigns and fixes.

Such performance gaps render large-scale deployments impractical and, in some cases, untenable for projects demanding a high degree of correctness. Regulated sectors in particular face even greater challenges. Industries like finance, healthcare, and insurance are bound by stringent guidelines that leave little room for the kinds of factual lapses or semantic errors that LLMs introduce.

In fact, a recent study (Li et al., 2023) observed issues in current LLM capabilities, highlighting how lengthier, context-rich inputs exceed what most models can process without compounding their earlier inaccuracies. There is a ". . . notable gap in current LLM capabilities for processing and understanding long, context-rich sequences" and "...long context understanding and reasoning is still a challenging task for the existing LLMs."

This issue reinforces the idea that despite their promise, LLMs still may fall short in scenarios where a high degree of precision is paramount, or when there is a high degree of complexity. Not surprisingly, organizations, aware of the potential for costly mistakes and legal implications, are often reluctant to integrate a technology that displays unpredictability when confronted with complex demands.

Unfortunately, unless these shortcomings are addressed, some of the most compelling market opportunities will remain out of reach, particularly in industries requiring rigorous compliance, such as banking, healthcare, and government.

The Reliability Issue Root Cause

As we have said, LLMs rely on probabilistic methods rather than fixed rules (they are non-deterministic), which explains why they sometimes generate incorrect or contradictory responses. Each output token is chosen based on the previous token, as well as the probabilities learned from vast amounts of training data, meaning there is no guarantee that the final sequence will be free of errors. To make matters even trickier, output may vary between runs despite identical inputs and environmental conditions. This behavior contrasts with a deterministic system, which for a given input would produce the accurate (or at least verifiably accurate) and identical output every time.

The root cause of this behaviour is what we call the *combinatorial explosion of choice*. When a model processes a short prompt, the path from start to finish is brief, and the chance of going astray remains low. However, processing a longer and more intricate prompt involves far more branching possibilities. The more choice, the more the chance of errors, and since errors multiply and cascade,

they grow exponentially.

Simply put, since each token depends on the previous one, a single minor mistake (especially one that happens early in the process) can cascade through the rest of the text leading to potentially catastrophic, or at least unreliable, non-repeatable, and inaccurate, results.

Figure 7-2, demonstrates the cascading of error rates. While error rates are much lower for actual use of LLMs,to simplify the math let's assume a 99 percent accuracy per token for the purposes of illustrating the math behind the combinatorial explosion of choice:

- *100 tokens:* 37% accuracy (0.99^100 is about 0.37)
- *1,000 tokens:* 0.004% accuracy (0.99^1000 is about 0.00004)
- *10,000 tokens:* Very, very low accuracy.

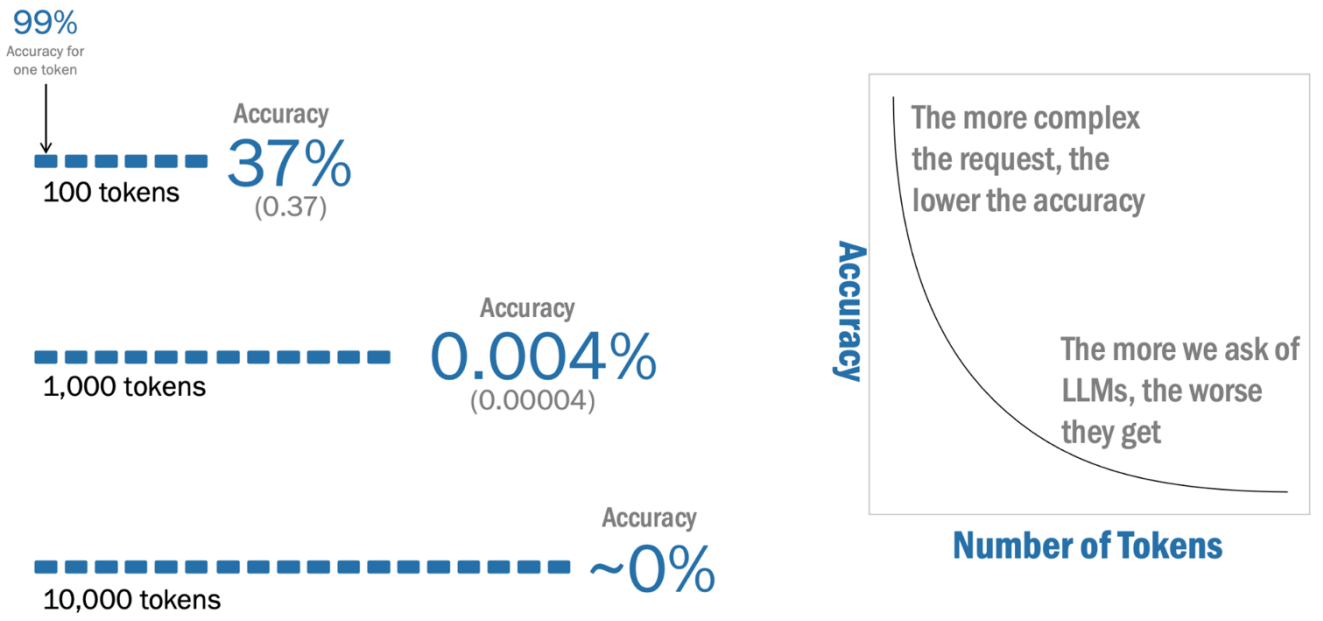


Figure 7-2. Combinatorial explosion of choice

This situation probably underscores why some of the most desirable LLM opportunities – those requiring large-scale outputs or complex reasoning – pose the greatest risk of errors. Today, this trade-off is a material consideration in determining where to apply LLMs.

Potential Solutions

Some practitioners attempt to tackle the accuracy and reliability gap by guiding the LLM through iterative refinement. They begin with a first attempt, then review any inconsistencies, and provide feedback to the model. This approach works reasonably well for smaller outputs, including short snippets of code or concise narratives, and by acknowledging the model's initial flaws and systematically

correcting them, developers manage to extract better final results.

A similar idea involves prompting the model (shown in Figure 7-3) to articulate its logic step by step, sometimes called *chain-of-thought reasoning*. This method tries to reduce inaccuracies and hallucinations by forcing the LLM to outline the intermediate steps between question and answer. This technique prevents the model from jumping to unsupported conclusions, particularly in complex reasoning scenarios, and it has proven to lower the number of stray or irrelevant outputs. Nevertheless, it does not eliminate the fundamental problem of compounding errors when the response grows longer.

Warning

The bottom line is this: as new projects expand in scale or complexity, the underlying mechanics of LLM predictions cause inaccuracies to compound, limiting the effectiveness of even the best prompting or refinement techniques.

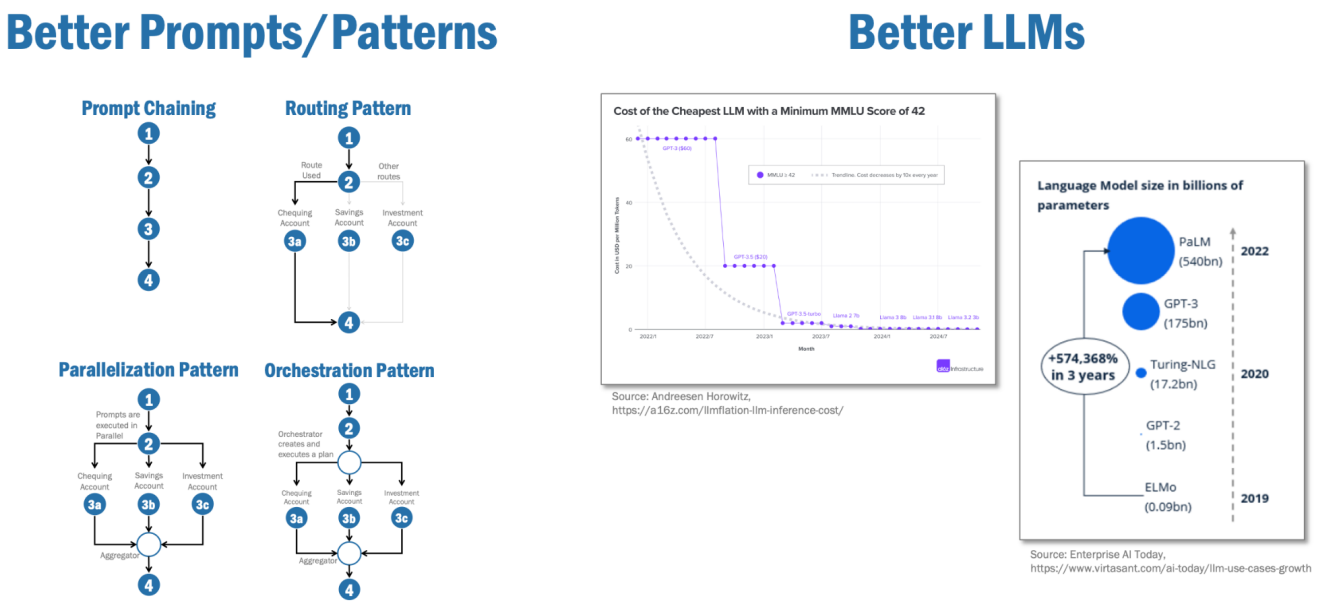


Figure 7-3. Potential solutions

What about better LLMs? Users often place faith in newer and more advanced LLMs as a straightforward remedy for accuracy and reliability challenges. Recent releases push the envelope on model size, training data, and sophistication, allowing them to handle longer input prompts and produce more complex answers without immediately hitting an error threshold where use becomes impractical. This expanded capability extends the threshold at which inaccuracies start to become meaningful, meaning users can request greater detail or breadth before the response quality plummets.

Yet this improvement only postpones the inevitable. Once organizations adopt these higher-capacity models, they quickly realize that the appetite for more extensive outputs grows in tandem. Early successes with moderately intricate

tasks, for example, may encourage teams to push for longer or more nuanced deliverables. Eventually, the same exponential rise in errors – the combinatorial explosion of choice problem – re-emerges, putting us back in the same cycle faced with smaller models (albeit on a larger scale).

Consider, for example, when ChatGPT GPT-3.5 was released, users were amazed by its natural language interactions, but inaccuracies and reliability became very evident quite quickly. Soon enough, GPT-4 came out, which fixed some of the errors, but then we found a new set of things could be done before we hit our error threshold. Still we demanded more, and soon audio and video were added to newer LLMs, and once again, we still found errors.

The fundamental issue is that firms, especially those in regulated industries that rely on highly accurate results, sometimes discover that their newly amplified abilities still fall short when the requests extend beyond a certain scope. Although the window of reliable performance has widened, every added token multiplies the chance of missteps, and the underlying probabilistic nature remains unchanged, so once tasks surpass this improved but finite capacity, the same cascade of mistakes reoccurs.

Unfortunately, this highlights the limits of counting on LLM scaling alone. Even with better models, the combinatorial explosion of choice and the problems it creates continues to

loom.

Task Decomposition

Another approach uses an *orchestrator* LLM to parse and interpret the initial request. This orchestrator, which can be a general-purpose LLM (or an LLM specializing in task planning), determines what needs to be done – a task plan with multiple steps – and then canvases the inventory of available LLMs and assigns specific task steps to whichever specialized models are best suited for them.

By breaking down a larger request into a series of smaller discrete steps - effectively, *task decomposition* - the orchestrator keeps each step's prompt and scope short and well-defined. These specialized LLMs act independently, rarely needing more context than necessary for their specific function. The orchestrator coordinates execution of LLMs, collects the final outputs, and assembles them into a coherent result.

This design results in independence of tasks. And it is *task independence* – the fact that task planning and task execution are handled independently by different LLMs – that leads to a lower overall error rate. Errors in one domain do not cascade through an entire solution because every step has its own boundary and is not forced to rely on the token-by-token path of a single execution thread.

An interesting by-product of specialization is that it also reduces the scope of data required by an LLM to only that needed to fulfill a task. Instead of sharing every piece of data with a general purpose LLM, organizations can restrict access to specialized models that are clearly authorized to handle certain types of information. A financial LLM that processes transaction data would remain separate from a model that writes marketing copy, reducing the risk of unnecessary data exposure.

Also, this type of orchestration also offers considerable deployment flexibility. Teams can plug in new specialized models as they are needed, trained, or made available, and retire older ones that become outdated. Everything revolves around the orchestrator's capacity to pick and choose the best resource for each situation, creating a dynamic environment that continually evolves without requiring a complete overhaul of the end-to-end architecture.

Deterministic Execution

Once a plan is established, the agent shifts into an execution role, calling on relevant tools or other agents that carry out their portions of the work independently. Each component, as illustrated in Figure 7-5, runs with minimal overlap, preserving reliability and reducing the chance of errors migrating across tasks.

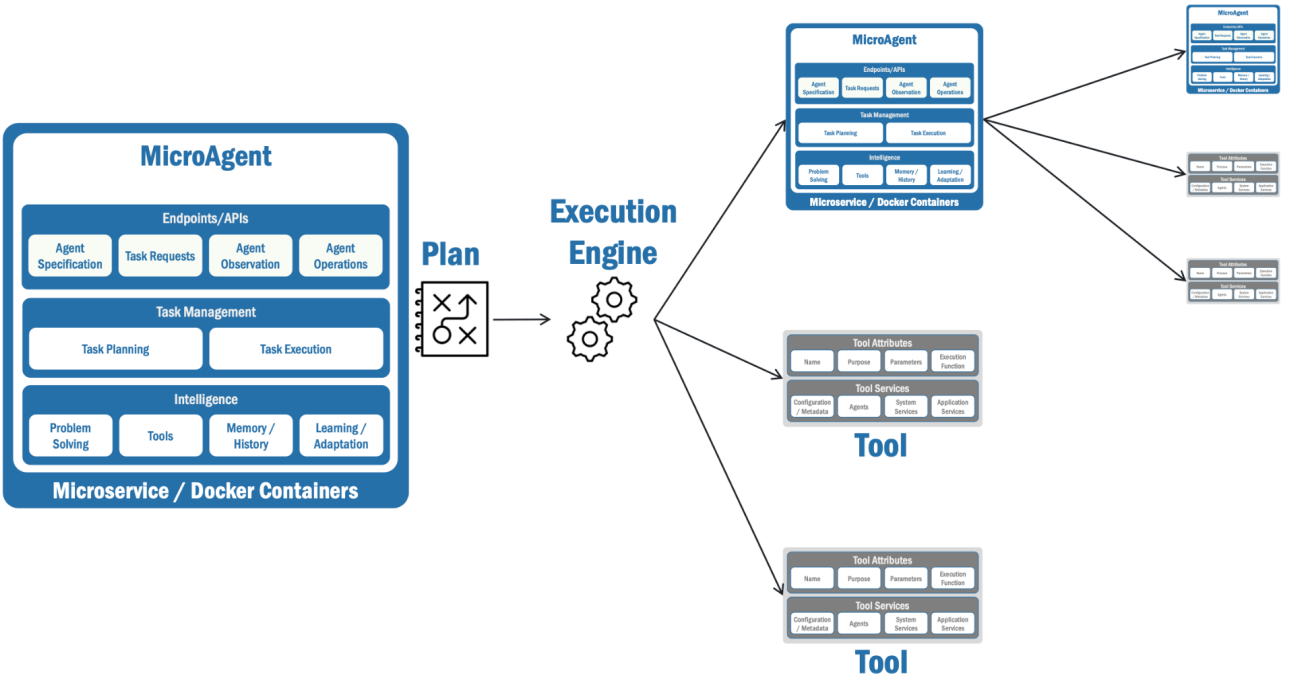


Figure 7-4. Deterministic execution

In this approach, an agent first consults an inventory of available LLMs and tools. It then decides which resource should handle each step of the plan, whether it is a specialized (perhaps domain-specific) model or a general-purpose one. The agent breaks down the request into small, clearly defined actions, ensuring that no single stage becomes unwieldy or error-prone. Once a step finishes, the agent aggregates the output and verifies it against any constraints or acceptance criteria. If needed, it can prompt a tool or model to retry a given action or correct a mistake before proceeding to the next stage.

This agent-based approach can also be integrated with enterprise infrastructure as a series of microservices. Each microservice can be secured, monitored, and scaled

according to the organization's requirements. Operators benefit from a stable environment, since the architecture limits the scope of any given failure and avoids the combinatorial explosion that emerges when a single LLM tries to handle every detail end to end. The microservices model likewise simplifies discovery and deployment, making it easier to roll out new agents with specialized LLMs without disrupting the broader pipeline.

Specialization

The growing sophistication and capabilities of LLMs have been matched – maybe even exceeded – by incredible reductions in their cost. This creates an opportunity for more specialized applications that do not depend on a single, all-encompassing model. Instead of waiting for one general-purpose system to evolve, firms can now afford to deploy multiple, smaller models, tuned and optimized for particular specific tasks. In fact, it appears that the march toward specialist LLMs is not only accelerating but is also inevitable.

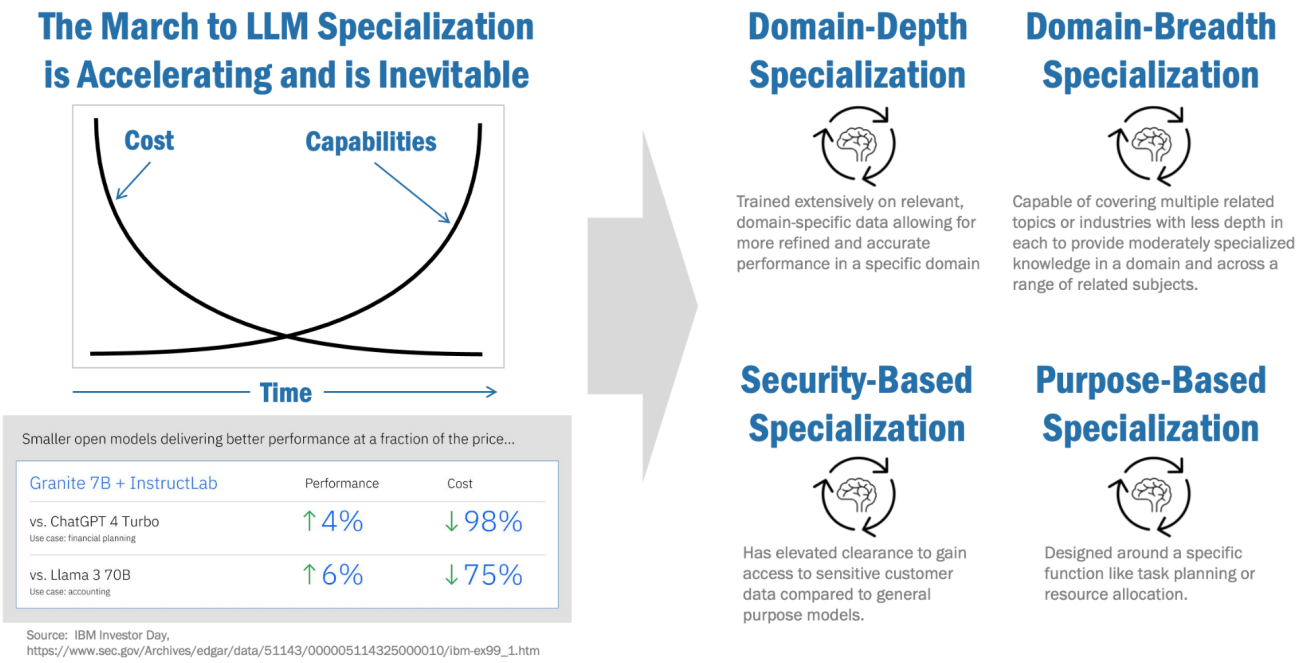


Figure 7-5. LLM specialization

Such specialization mirrors trends seen in other industries. In semiconductor manufacturing, companies once tried to build entire chips on their own, but quickly realized it was more efficient to outsource specialized components. Over time, highly specialized fabrication plants emerged, reducing costs and raising quality. Similar patterns appear in supply-chain networks, where each participant develops a specific competency. The logic is straightforward: when participants concentrate on what they do best, the entire system benefits.

Economists sometimes call this the *theory of comparative advantage*, which states that overall productivity increases if each party focuses on activities where it holds a distinct edge. Applied to LLMs, this suggests that no single model

will dominate every use case. Instead, multiple specialized models can flourish, each specifically trained for a narrower domain.

In practice, specialization means users do not have to rely on one massive LLM. They can implement smaller, dedicated LLMs immediately, covering distinct areas without compromising on performance.

Some organizations have begun to orchestrate multiple language models, combining a general-purpose LLM with one or more specialized models. Each LLM handles a different part of a complex task, ensuring that no single model is overloaded with the entire problem. The aim is to capitalize on the versatility of a general model while leveraging the deep expertise of smaller, more focused LLMs that excel in narrow domains.

A Future with Reliable Agents

LLMs will continue to get better, but their non-deterministic design and the combinatorial explosion of choice problem it creates means their users are still subject to their errors, inaccuracies, and hallucinations.

Rather than wait for the next LLM,but there are other options. Increased capabilities combined with reduced costs are leading to specialization, and new orchestration

techniques allow us to turn use these bigger problems into smaller more tractable parts all the while reducing errors and inaccuracies. And when these techniques are implemented in an agent architecture, especially one built upon microservices, for example, such a system can take advantage of decades of security, observability, and operability experience to address the challenges of a new approach.

And it is this new approach which turns a *forever* problem – that of non-deterministic LLMs and combinatorial explosion of choice – into a decomposition problem with reduced errors and inaccuracies based upon a foundation of solvable, long-studied, and well understood solutions.

Agent Explainability

Widespread agent adoption will only occur when we trust them. But what does it mean to trust an agent? In the simplest sense, *trust* means believing that an agent will do what it is meant to do.

This brings us to a pressing challenge with agents today: agents are driven by LLMs that are both non-deterministic and opaque. This has two major implications: first, LLMs that power agents sometimes produce unreliable, inconsistent,

and error-prone output, and second, when they do cause errors, we cannot figure out why they did what they did.

Agent explainability addresses this gap: making an agent's task plans visible, measurable, and understandable begins to open the agent LLM black box and let trust into the process.

The Trust Gap

Trust, in human relationships, is built over time through observable behavior, consistent actions, and shared understanding. We trust other people not only because of what they do, but because we understand why they do it—we can usually infer motives, judge consistency, and form expectations. And when that trust is broken, we seek explanations.

However, agents don't have the same intuitive understanding as people. As a result, the criteria we use to trust agents must require a focus on transparency and the ability to explain behavior. This matters because it looks like we will soon be delegating more, and more important, responsibilities to agents. And as they become more embedded in business operations, the opportunity and the impact of errors increase.

The problem is the result of two related challenges:

- LLMs that power an agent are non-deterministic—they make mistakes.

- LLMs are black boxes with no view of their internal logic—when they do make a mistake, we cannot figure out why.

Opaqueness means that when an agent acts correctly, we don't know how or why it reached its conclusion. And when and agent fails, we don't have the information to diagnose why it failed. This lack of insight into agent decision making is a fundamental obstacle to trusting agents at scale.

Explainability: Real-world Lessons

Trust is rarely assumed; rather it is earned through transparency, monitoring, and the ability to diagnose failure. In other words, trust is predicated on explainability, and in practice we use explainability all over the place in the real-world.

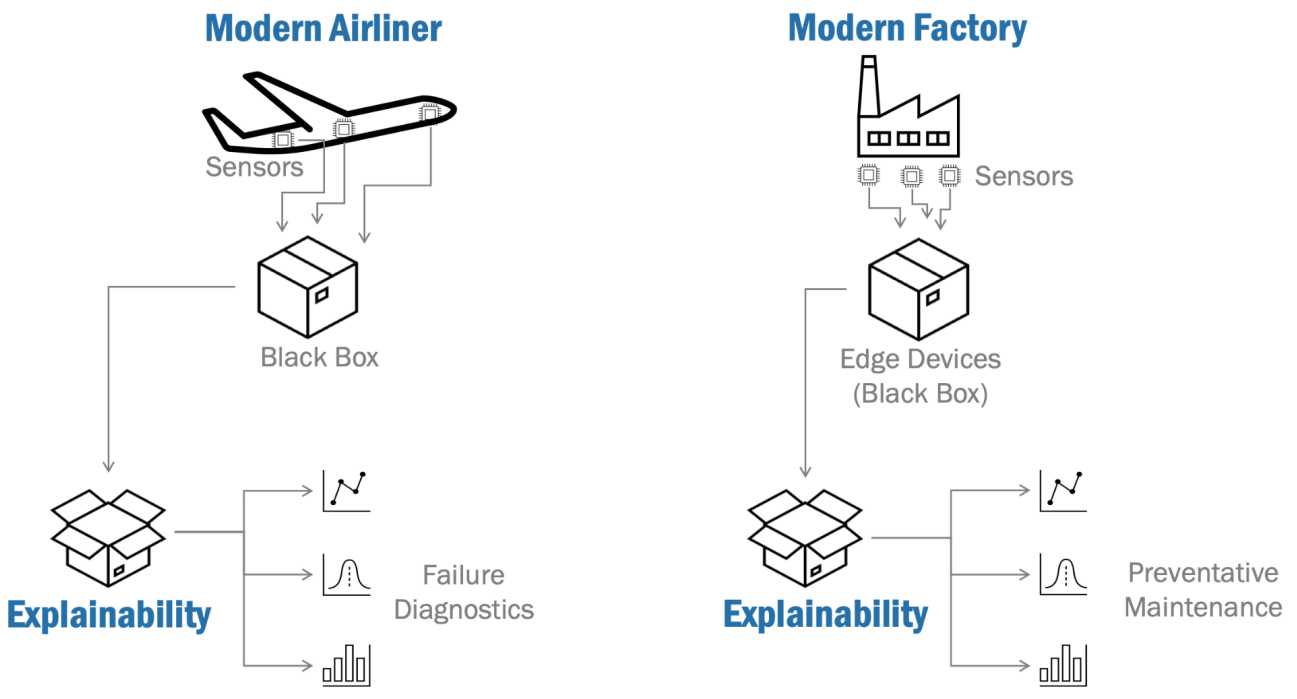


Figure 7-6. Real-world explainability

In aviation, every commercial aircraft is equipped with a flight data recorder and a cockpit voice recorder — together known as the *black box*. These systems run silently in the background, capturing critical data like airspeed, altitude, control inputs, and pilot conversations. When an accident occurs, investigators rely on this data to reconstruct events, determine root causes, and implement corrective actions. The black box doesn't prevent failure, but it ensures that investigators can understand and explain it — and this explainability, in turn, drives systemic improvement and long-term trust in air travel.

In modern factories, the black box takes the form of embedded sensors and monitoring systems. Factory machinery can track diagnostic information like temperature,

vibration, pressure, and just about any metrics necessary to monitor operations as well as detect early signs of failure. The ability to explain why a machine failed — or even better, to prevent it through continuous insight — has become a huge part of modern manufacturing.

Importantly, both of these black boxes are not actually opaque; rather, *they are explainability systems in disguise*. They literally create a trail of evidence of operational behavior that can be inspected, verified, and explained.

This is what we should expect from agents. If we are to trust them with increasingly complex tasks, agents need to do more than produce outputs. We need agents that, like their counterparts in aviation and industry, leave behind a trail of data — an explainable record of what they intended to do, how they did it, and why it turned out the way it did.

Explaining Explainability

At the core of agent explainability is a simple idea: treat task plans as first-class artifacts. In practice, this is composed of several steps, each creating its own explainability artifacts (as illustrated in Figure 7-7):

1. *Task plan:* This includes the detailed structured representation of the steps the agent intends to take. Importantly, since agents can interact with other agents,

this means capturing the task plans, recursively, for each agent in the task plan / execution tree dynamically as they are created.

2. *Collaborating agents and tools:* These identify and explain who the agent is intending to interact with, or which tools the agent is expecting to use.

3. *Parameter substitution log:* This explains how the inputs (also logged) are decomposed into parameters that are provided to collaborating agents or tools.

4. *Task execution log:* This includes instructions used to create the task plan as well as any additional information used to form the task plan.

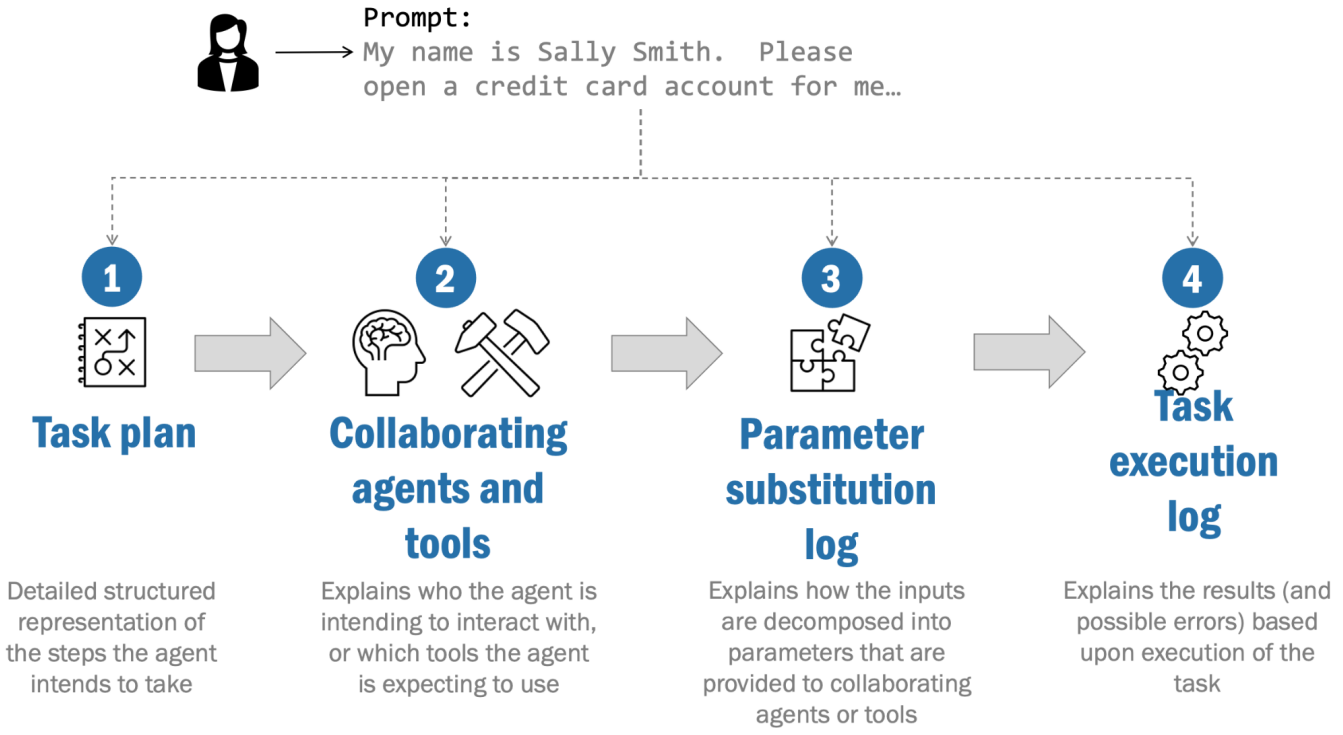


Figure 7-7. Task explainability

As of this writing, AI workflow (in contrast to an agent) is designed to let the LLM drive the end-to-end process: it creates an internal opaque plan, uses hidden logic to ingest prompts and inputs, uses unseen execution capabilities, and finally tosses the plan out once the task is complete. These plans are ephemeral.

In an explainable agent framework, task plans are captured, persisted, and made visible as an immutable historical artifact. However, creating a plan is not the same as following it. That's why explainability also requires monitoring the agent's execution against its declared plan. This involves tracing execution metrics such as task progress, branching decisions, unexpected conditions, and error-handling behavior, as illustrated in Figure 7-8.

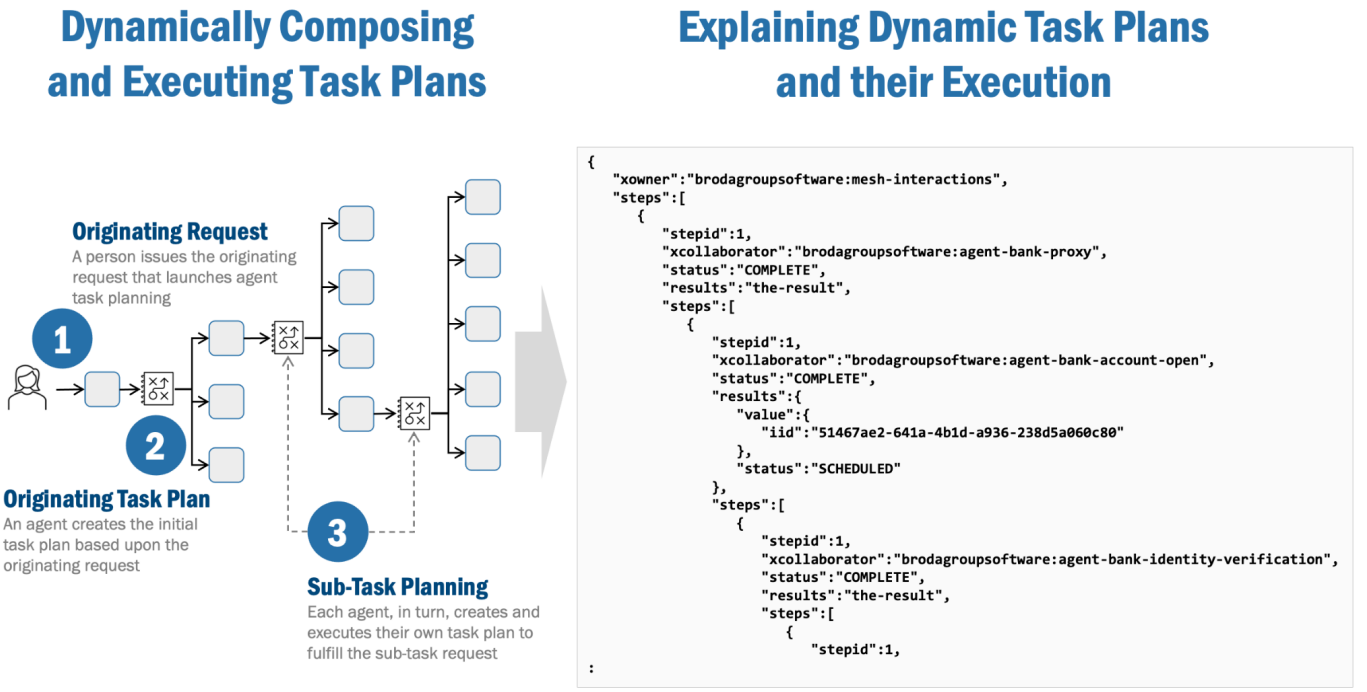


Figure 7-8. Task plan and task execution visibility

These execution traces, combined with the task plan history, form the basis for diagnostic inspection by engineers, auditors, and oversight systems. When behavior appears anomalous or incorrect, these records allow staff (governance professionals, engineers, and so on) to reconstruct the end-to-end task plan and execution path, determine whether the agent deviated from its instructions, and assess whether the deviation was justified.

Towards Explainable Agents

Explainability is often treated as simply a diagnostic tool. But in the agent ecosystem, explainability must be viewed as a fundamental capability—a prerequisite for governance, compliance, and assurance. As agents take on more complex and consequential tasks, stakeholders need to understand not only what an agent did, but why it chose that course of action. This is particularly critical in regulated industries such as finance, healthcare, and critical infrastructure, where transparency is not optional. Without a mechanism to trace decisions back to intentions and plans, agents cannot be verified, audited, or held accountable — and trust remains out of reach.

Explainability also enables a path to certification and oversight. Just as traditional software undergoes rigorous quality assurance before being deployed in high-stakes environments, agents must prove they can reliably operate

within their intended bounds. A transparent task plan — coupled with metrics that track execution fidelity — provides a basis for certifying agent behavior. Whether it's a third-party review, an internal governance audit, or a regulatory filing, explainability enables repeatable, evidence-based validation of how an agent will act. Without this, agents remain unpredictable — and unpredictability leads to a lack of trust.

Finally, explainability supports continuous improvement and operability. It provides visibility into intermediate states, decisions, and deviations from planned behavior. It enables robust debugging, adaptive retraining, and the safe evolution of agent capabilities. At scale, this becomes even more critical: in a world where thousands or millions of agents interact autonomously, trustworthiness is predicated upon explainable design.

An agent ecosystem must include explainability, but not as an optional feature or afterthought, rather as a core design principle. To do this, explainability must be designed in from the start. Agents must be built with transparent task planning, traceable execution, and observable metrics from the ground up.

The challenge is clear: opaque agents will inevitably undermine trust. The solution is equally clear: we must design agents that explain themselves.

Agent Scalability

In an ecosystem of potentially thousands, or millions, of agents, one absolutely clear imperative is to ensure that the emerging agent ecosystem can *scale*. It must scale not just to handle the load, but also so we can build them quickly and efficiently, and it must operationally scale so that we can manage vast ecosystems of agents effectively.

Unfortunately, today's agents, perhaps better described as AI workflows, fall short. According to Anthropic: "LLMs and tools are orchestrated through *predefined* code paths." We believe the architecture as well as implementation approach for AI workflows will have material challenges in scaling to support larger deployments:

- *They will not scale from an execution perspective:* They typically are built in a single Python *main* (albeit importing modules can make it modular), only run in a single operating system process, and are clearly not able to scale beyond simpler execution loads.

- *They will not scale from a development perspective:* AI workflows use a pre-defined execution path which requires an end-to-end knowledge of agent execution as well as edge cases, and these AI workflows are limited to collaborating with components explicitly

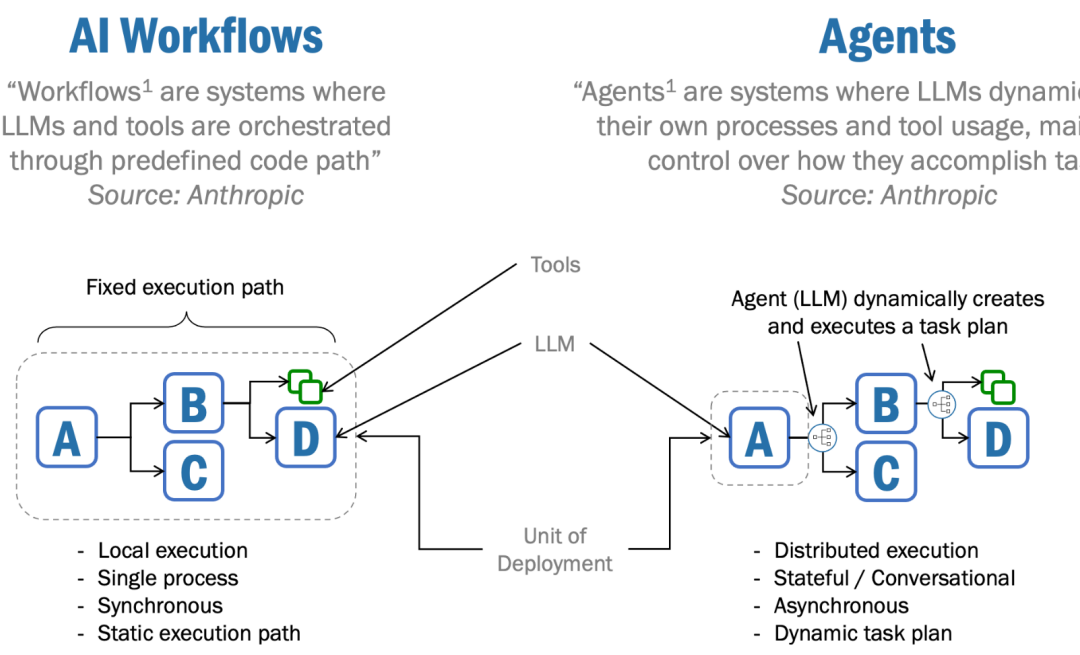
defined in a static program – adding more components to the program becomes impractical very quickly. This is clearly not practical for building agents at scale.

- *They will not scale from an operational perspective:* Although building operational *hooks* – logging, alerts, traceability, and explainability – into AI workflows can be done, without common patterns and tools, usually embedded in toolkits, this is not operationally scalable.

The root cause, as we highlighted earlier, is that today we simply ask far too much—not just of LLMs, but of the current AI workflow architecture. Ultimately, this leads to a compromised architecture - one that cannot scale from either a development, run-time, or operational perspective.

The Scalability Problem

The crux of today's agent implementations is that they are simply single Python "main" function that orchestrates every step of execution flow. While this setup works adequately for demos or small-scale tasks, it inherently lacks the flexibility and robustness needed for large-scale enterprise use cases. As mentioned earlier, developers can still build these monolithic designs in a modular way (through imports and auxiliary functions), but the central logic still ends up funneled through a single process, creating performance bottlenecks, and limiting concurrency (Figure 7-9).



1. Source: Anthropic, https://www.anthropic.com/research/building-effective-agents

Figure 7-9. Design of AI workflows and agents

From an engineering standpoint, the single-process model also poses performance limitations. If one agent needs to parse large volumes of data, interpret user requests, and generate responses in quick succession, a single process could become a throughput bottleneck. Meanwhile, in a multi-agent setting—where agents specialize in different tasks—having them all tied to a single operating system process compounds the risk of crashes, memory leaks, or blocking I/O calls. A single failure can bring down the entire pipeline, harming reliability. Distributed architectures, on the other hand, allow each agent to run in its own environment, preventing localized failures from taking down the system. Figure 7-10 outlines AI workflow scale challenges.

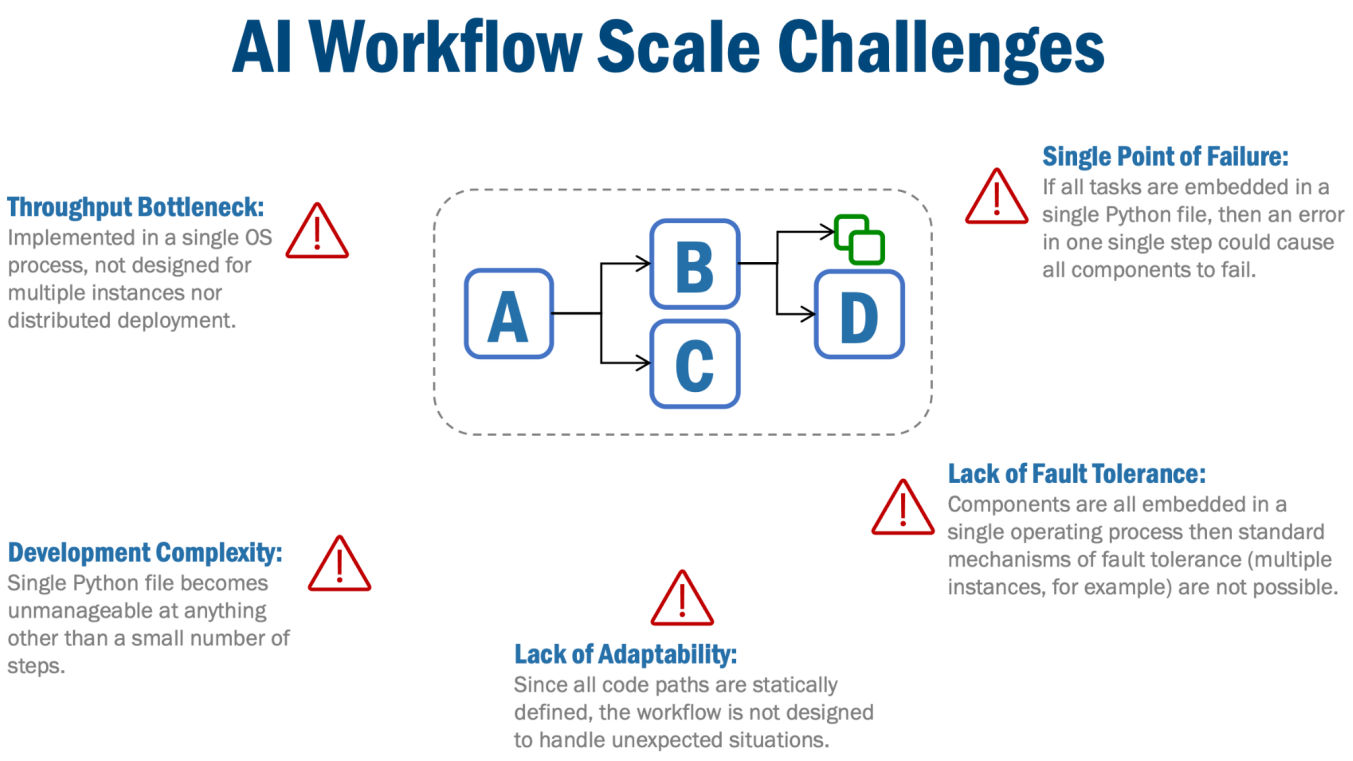


Figure 7-10. AI workflows challenges

Additionally, large enterprises demand robust design principles like fault tolerance, load balancing, and versioning. Monolithic agent frameworks, however, typically lack built-in ways to manage these concerns. Introducing them after-the-fact often resembles patchwork rather than a coherent architecture. In contrast, agents require an architecture that comes with well-established design patterns for service discovery, traffic routing, and high availability, all of which are critical in real production environments.

Additionally, monolithic AI workflows (at least the ones seen in most current agent toolkits) typically lack the capacity for meaningful self-diagnosis or analytics. If the agent encounters an unforeseen failure, it might raise an exception or log an error, but the recovery path typically involves

human intervention to fix the code. Operating agents at-scale requires that individual services detect errors, restart, or reroute themselves, and keep running. This feature is crucial for large-scale enterprise environments where downtime must be minimized, and 24/7 availability is a core requirement.

So far, we have addressed run-time scaling issues, but this monolithic design also introduces build-time (development) scaling challenges. The reliance on predefined workflows forces developers to anticipate each execution branch or edge case the agent might encounter, then manually code the fallback logic. Such an approach is only practical for relatively simple tasks. As complexity grows, the flowchart of possible states and transitions becomes unmanageable.

Complicating matters is the fact that these monolithic AI workflows tend to work in isolation, collaborating (if at all) only with agents explicitly defined in the same codebase. So, the only way to add functionality is to add more and more agents into the monolithic code base. Not only is this approach time-consuming, but it also undermines the notion of autonomy. Instead, truly scalable systems would permit agents to discover and cooperate with one another dynamically, a principle familiar in distributed computing.

There is also a philosophical gap here between *AI workflow*, predominant in monolithic agent designs, and *autonomous*

agents. True autonomy involves dynamic task planning and the ability to gracefully handle novel obstacles. Monolithic AI workflow designs do not support autonomy at scale because their capacity to respond adaptively is limited by predefined flows.

Taken together, these constraints underscore why the current generation of AI workflows—packaged as a single Python main and constrained by pre-defined flows—struggles to scale. To move toward genuinely autonomous and extensible agents, we need an architectural overhaul. As you have seen, microservices provide a blueprint for distributed deployments that can accommodate many specialized agents, each equipped with its own logic and resources. With dynamic discovery, flexible orchestration, and robust fault tolerance, enterprise-grade solutions can finally push past the performance and scalability ceilings of the monolithic agent paradigm. The enterprise needs this shift to unlock the full potential of AI-driven automation and decision-making, especially when the goal is to have hundreds or thousands of agents collaborating on complex tasks.

How do we solve these problems? That is what the next few sections will address.

Distributed Architectures

A distributed agent architecture - made practical with our MicroAgent architecture - leverages commonly used techniques to provide agent execution at-scale.

To recap: today's agents are built upon bottlenecks and constraints inherent in a single program/single process architecture. A distributed architecture solves the single-process bottleneck by spreading the agent workload across multiple computing nodes or environments, making it possible to run many agents in parallel. Rather than funneling every task into a single Python main, each agent can and should operate independently, handling its own tasks and resources. This approach significantly improves throughput, especially in high-demand use cases where large volumes of data must be ingested and processed. Figure 7-11 illustrates several advantages of agent scale.

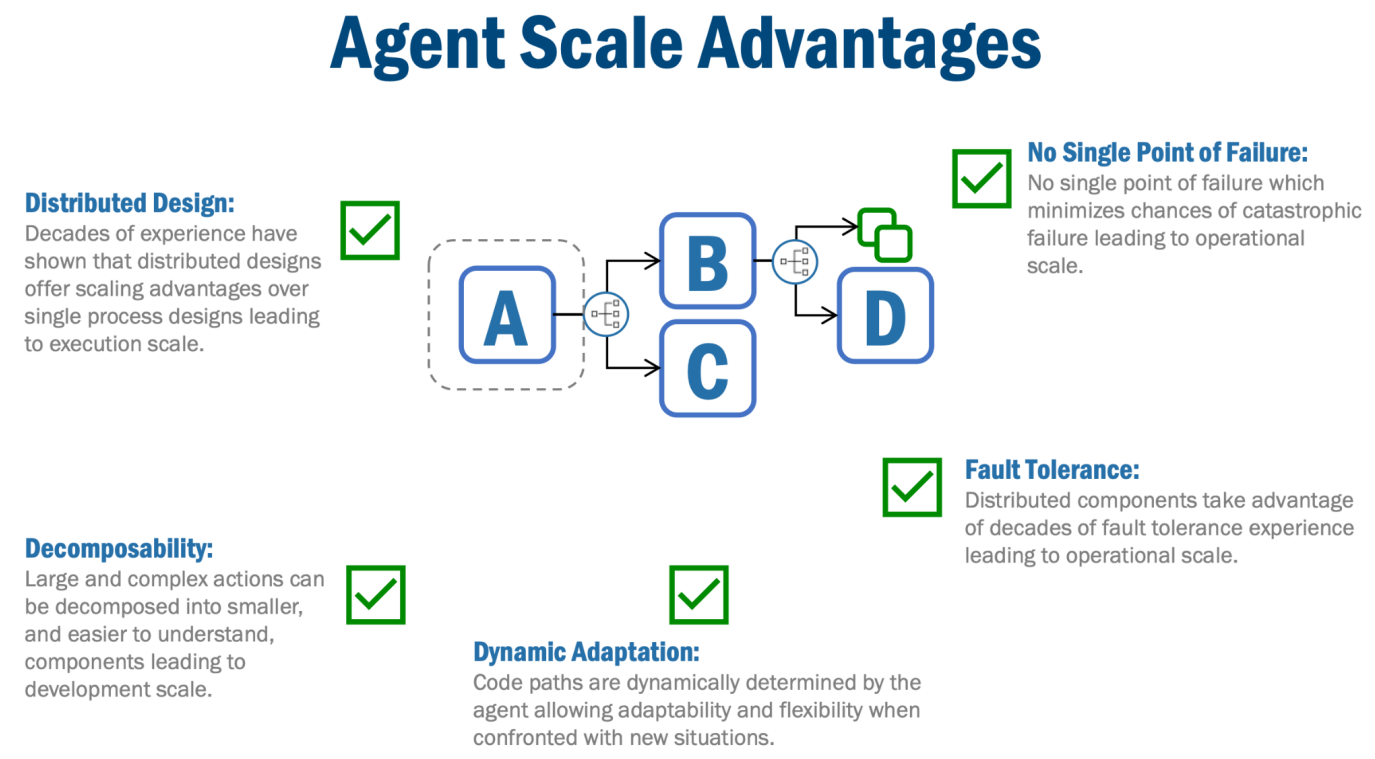


Figure 7-11. Agent scale advantages

Distributed architectures open the door to dynamic scalability, a key requirement for handling large agent ecosystems. By design, these architectures allow new nodes or processes to spin up as needed. This elasticity helps meet spikey workloads, where the need for compute power fluctuates dramatically. In large organizations, it also means different departments can maintain separate agent clusters sized to their own business requirements, yet seamlessly communicate when needed, rather than competing for space in a single, monolithic environment.

Beyond raw scalability, distribution improves deployment flexibility among agents. Suppose each agent can be specialized to a particular domain while still interacting with others over well-defined communication protocols. In this setup, teams can add new specialized agents or retire outdated ones without redeploying the entire codebase. As new tasks or services emerge in an enterprise, the architecture can expand by introducing corresponding agents rather than refactoring a monolithic workflow. This modularity is essential when organizations want to push new features and updates frequently, or experiment with novel AI-driven services in a safe, compartmentalized way.

Finally, a distributed approach facilitates centralized monitoring and analytics at scale – a fundamental

requirement if we expect to see millions of agents. By running agents as discrete entities, engineers can track each agent's health metrics (for example, CPU, memory usage, error rates) and collect logs in a centralized dashboard. This visibility is crucial not only for troubleshooting but also for strategic planning: if certain types of agents are consistently overloaded, organizations can provision more resources accordingly. Industry-standard tools in distributed systems - like distributed tracing, event log aggregators, and monitoring agents - are already well established and can be readily integrated.

Common Collaboration Techniques

Common collaboration techniques provide consistency in the way that agents and tools interact, offering execution, development, and operational scale.

Common collaboration techniques provide a unifying framework for how agents exchange information and coordinate tasks, addressing one of the biggest limitations of monolithic agent designs. To be sure, this field is evolving rapidly, with specifications like Model Context Protocol (MCP) from Anthropic leading the way (see Figure 7-12). However, so far these efforts are in their earliest stages.

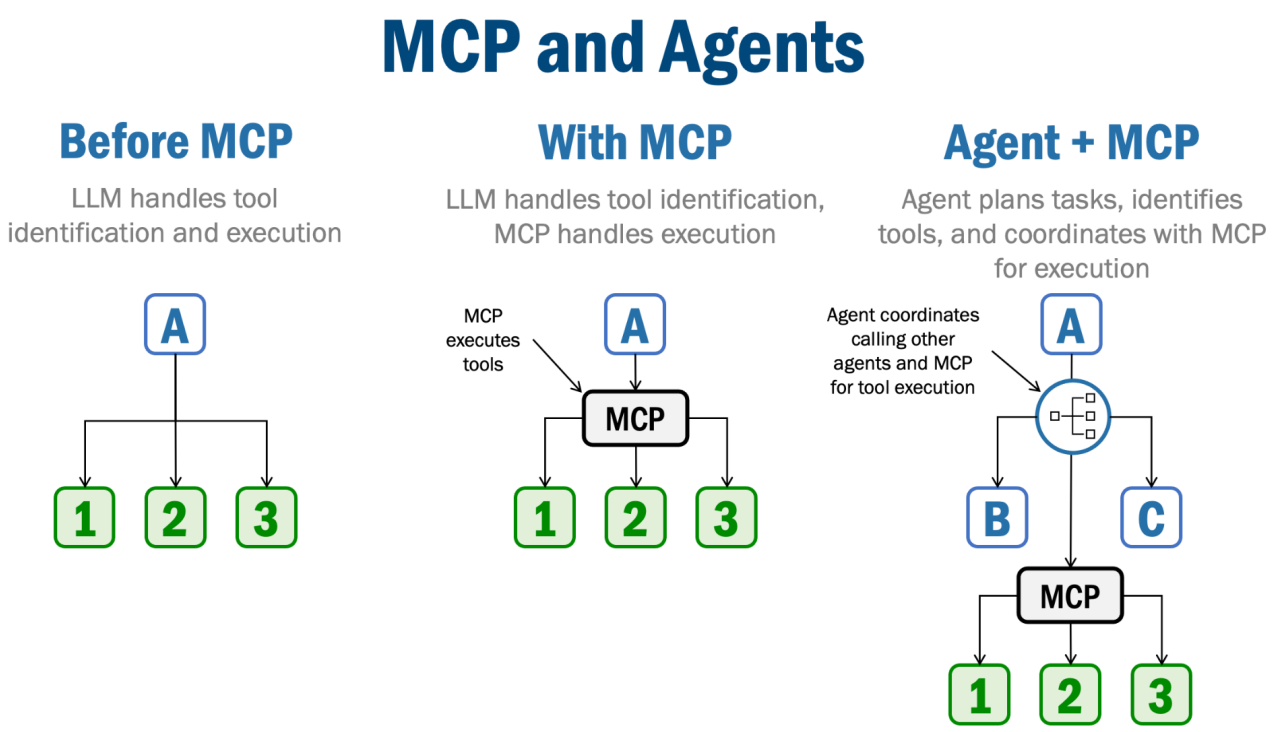


Figure 7-12. MCP and agents

MCP – arguably the most popular and innovative approach for tool interaction – is positioned by Anthropic as "an open protocol that standardizes how applications provide context to LLMs." Anthropic continues: "Think of MCP like a USB-C port for AI applications. Just as USB-C provides a standardized way to connect your devices to various peripherals and accessories, MCP provides a standardized way to connect AI models to different data sources and tools."

That sounds great, but if we carry Anthropic's analogy further, what we get is USB-C cable standards, when what we need is established standards on what goes *over* the cables. So, clearly there is more to be done.

We suggest that the next stage of agent evolution address the standard approach or protocol that runs on top of MCP. Whether it is MCP or any other accepted standard, the approach we envision is similar to how MCP approached this:

- *Messaging standards:* Start by defining clear, lightweight messaging protocols based on widely recognized formats such as JSON, XML, or protocol buffers that let distributed agents communicate with each other.

- *Interaction standards*: Let agents communicate with each other in a coherent manner to complete tasks.

- *Collaboration standards*: Let agents interact on related topics over long periods using *conversations* (we think this term is an appropriate analogy from human interactions, but may be viewed as *sessions* from a technology perspective)

A well-defined collaboration, or conversation, standard typically covers not just the data formats, but also how messages get routed, how acknowledgments work, and the security or authentication required for interactions. It would also address higher-level constructs such as requesting task fulfillment, exchanging information, querying task status, or interacting with people.

Adopting consistent collaboration standards makes the entire agent ecosystem more resilient and maintainable. Instead of debugging a tangle of hard-coded integrations, engineers can diagnose and fix issues in well-defined channels that any agent uses. This is particularly beneficial when scaling to hundreds or thousands of agents, each focused on its own specialized area yet needing to hand off tasks or information to other agents. The reduced friction in coordination translates into faster, safer evolution of the overall system.

Beyond simplifying immediate interactions, collaboration standards also enable higher-level patterns like event-driven architectures and publish-subscribe messaging, where agents can “listen” for relevant events and react in real time. This opens the door to advanced features – such as emergent behavior and more dynamic task allocation – since agents can spontaneously form new workflows by subscribing to events produced by other agents. Ultimately, collaboration standards are the key enabler that help transform monolithic AI workflows into truly autonomous, networked agents that can flexibly scale and adapt as organizational needs evolve.

Conversation/State Management

Conversation and state management allow agents to manage long-running interactions in a reliable way, offering

operational scale.

Conversation management lets agents maintain context over extended periods. Although a single-shot or short-lived conversation might suffice for simple tasks, more complex enterprise workflows can unfold over minutes, hours, days, or longer. A robust conversation management layer ensures that each agent retains the relevant context of every exchange, enabling the agent to revisit prior steps, incorporate new developments, and handle deviations without requiring an entirely new workflow script. This approach, illustrated in Figure 7-13, allows for fluid, multi-turn interactions that can adapt to changing goals and unforeseen obstacles.

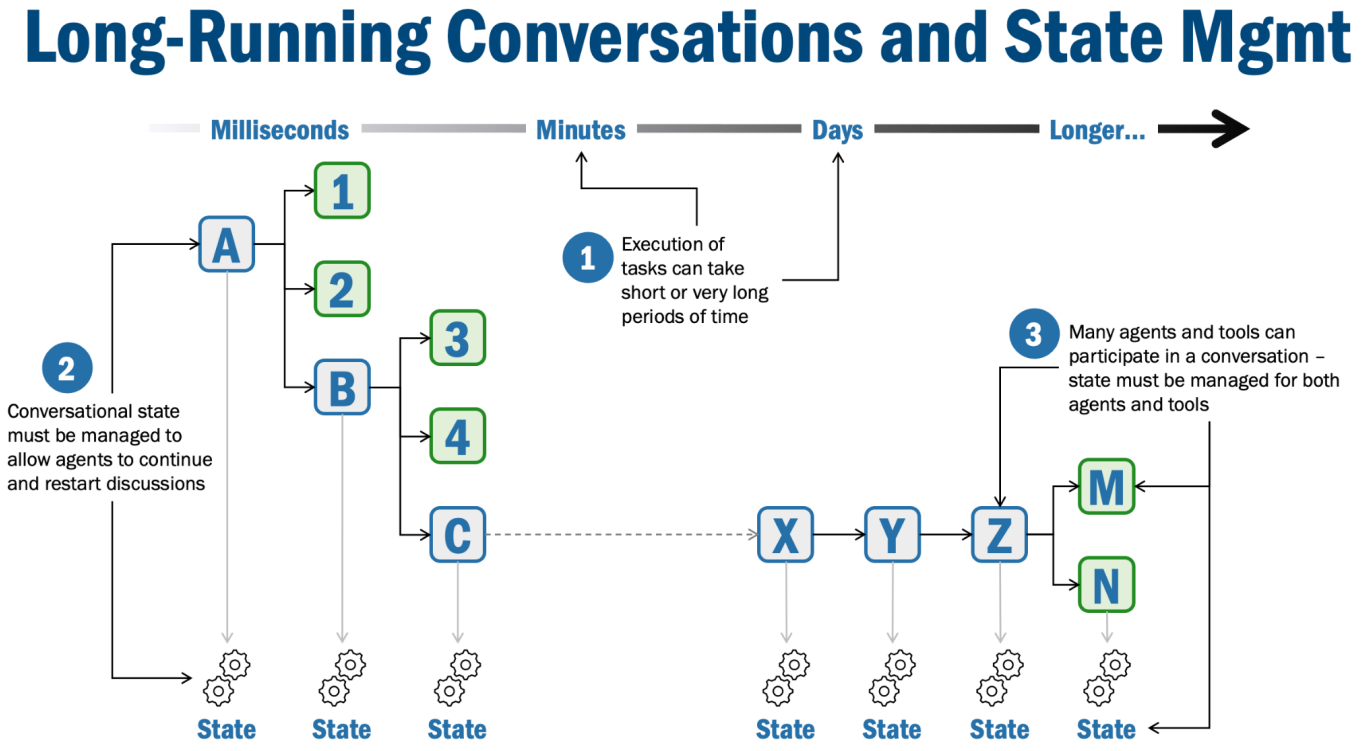


Figure 7-13. Long-running conversations and state management

Now, if agent conversations can span longer periods of time, and consequently the likelihood or potential of an agent failing increases, then perhaps obviously agents must be able to recover gracefully from failures. That is where state management comes in: it allows agents to remember what they've done, who they interacted with, and what transpired in their interactions – what we call *conversational state* – and use that information to recover from failures.

However, when an agent relies on naïve or simple techniques – for example, storing state in local environment variables – then a crash or restart can wipe out all knowledge of what came before. With robust state management, though, the agent's memory can survive crashes and reboots, and the agent can continue from the exact point it left off. This allows for more advanced workflows where tasks can be interrupted, rescheduled, or handed off to another agent if necessary. It also means that unexpected failures do not require the entire process to start again from scratch, which is an invaluable feature in large-scale deployments, where every minute of downtime can be costly.

In addition to persistence, well-designed state management includes mechanisms for concurrency control, versioning, and rollback. Agents operating in a concurrent environment can accidentally create conflicts or inconsistent states unless safeguards exist. Techniques such as transaction-

based updates or optimistic concurrency checks help ensure that multiple agents can update shared state safely. If a conflict or error occurs, robust versioning and rollback mechanisms can revert state to a known good version, preventing minor issues from spiraling into system-wide failures.

Finally, a consistent approach to state management simplifies monitoring, logging, and auditing. When all agent interactions with state are routed through the same interface (for example, an API), it becomes much easier to track changes, generate meaningful logs, and audit who or what made a particular update. By building state management into the core agent design, organizations can mitigate the limitations of monolithic, in-memory workflows and create truly scalable, resilient agent ecosystems.

Treating state and conversation management as core components of an agent's design allows developers to avoid piecemeal solutions that grow unwieldy as more capabilities are added. Instead of writing custom code for every edge case or conversation path, agents can rely on standardized ways of storing their state and managing multi-turn dialogues. By removing the need to retrofit conversation/state-tracking logic onto each new agent or workflow, organizations can drastically reduce development overhead and ensure that all agents behave in a predictable,

well-governed manner from day one.

Enterprise-Grade Agent Capabilities

Enterprise-grade agent capabilities let enterprises build agents fast, run agent ecosystems at-scale, and manage large agent ecosystems efficiently and effectively. They offer execution, development, and operational scale.

First, how can we build agents fast? We start by standardizing the way agents are developed—using templates, coding conventions, and guidelines that ensure consistency across the entire organization. Much like code frameworks and style guides in traditional software development, these templates provide a blueprint that new agents can follow, flattening the learning curve for teams. They also help mitigate the messy patchwork of scripts and modules that can emerge when each agent is built in an ad-hoc manner.

By adopting well-documented standards and best practices up front, enterprises establish a coherent baseline from which dozens, hundreds, or even thousands of agents can be deployed without reinventing the wheel each time. Similarly, development templates typically define how an agent should communicate, handle errors, log information, and authenticate users or other agents. So, when every agent follows the same blueprint, creating agents and

integrating agents into an agent ecosystem becomes far easier, faster, and less costly.

In addition to development consistency, enterprise-grade capabilities focus heavily on run-time concerns like security, discoverability, and observability. A robust security model - covering everything from encryption in transit to role-based access - ensures that each agent only performs authorized actions and handles data responsibly. Discoverability mechanisms, such as service registries or agent directories, let new agents or services find each other dynamically, preventing the brittleness of hard-coded endpoints. Observability tools—ranging from log aggregation to distributed agent conversation tracing—equip operations teams with the real-time visibility needed to diagnose problems and measure performance across the network of agents.

Reliability is another foundational pillar of enterprise-grade agent design. The goal is to make each agent resilient to failures (for example, network outages, hardware crashes, or transient errors in data processing). Techniques like automatic retries, circuit breaking, and load balancing become part of the default toolkit rather than optional afterthoughts.

At an operational and governance level, explainability and traceability become paramount. Many industries, especially

regulated industries such as healthcare, finance, and government, among others, must comply with rules and policies mandating a clear audit trail for all actions performed by automated systems. Therefore, agents that can record who requested an action, when it was completed, and why a particular decision was made align well with regulatory and compliance requirements. Mechanisms for capturing and storing this metadata should be integral to the agent's architecture, ensuring that accountability and compliance checks are possible even when the system spans hundreds of interconnected agents.

Finally, the concept of certification or accreditation builds upon all these capabilities—development standards, security, discoverability, observability, reliability, and traceability—to provide assurance that an agent meets the rigorous requirements of an enterprise environment. Whether it's achieving internal quality benchmarks or satisfying external audits, agents must demonstrate they are fit for large-scale, mission-critical operations. When built into an agent's design from the start, these certification processes become far less cumbersome.

So, while some enterprise-grade capabilities directly impact run-time scaling issues and let you run large agent ecosystems, others just as importantly let you build agents at-scale, while others let you manage large agent

ecosystems effectively. The result is an ecosystem of agents that are built fast, can handle high-volume tasks, *and*maintain the governance, visibility, and assurance demanded by modern enterprises looking to deploy large numbers of intelligent agents.

Agents as the Quantum of Reuse

Agents are the new granular reusable enterprise component, offering development scale.

When agents are treated as the core unit, or *quantum*, of reuse, organizations can develop them once and redeploy them multiple times in different scenarios without reinventing the wheel. Note that this is not an “agent thing” but rather is really applying software best practices from decades past.

However, reusing agents changes the unit of reuse in a profound way. In the past, the unit of reuse was a function, perhaps available in a library. With the evolution of standard communication approaches (HTTP/REST, gRPC, and so on), APIs encapsulated higher-level business capabilities and thereby provided a much more valuable unit of reuse.

With agents we not only can encapsulate even higher levels of abstraction, and interact in a much simpler manner, but we can also introduce “smart” capabilities (via the LLM

embedded in, or available to, agents) that can handle much more complex business logic. In other words, agents become the natural next step in the evolution of reuse.

This level of reusability helps solve the development scalability challenge, which is one of the most pressing obstacles in building large agent ecosystems. If teams are constantly forced to start from scratch, the ramp-up time for each new agent becomes quite substantial, and parallel projects risk duplication of effort. By contrast, reusing agents accelerates project timelines; it shifts the focus from basic functionality building toward higher-value capabilities like customization, domain-specific intelligence, and user-facing improvements. Moreover, it curbs technical debt by encouraging consistent development practices rather than letting one-off solutions proliferate.

Importantly, the larger the agent ecosystem, the stronger the incentive and possibility for reuse. As the registry or marketplace fills with specialized agents, teams are increasingly likely to find that exactly what they need has already been built—be it a language translation agent, a fraud detection agent, or a logistics planning agent. Each agent becomes a LEGO block that fits into a vast system of interlocking services. This virtuous cycle accelerates innovation: by lowering the barrier to adding new features, organizations can quickly prototype and deploy

combinations of existing agents to address emergent business requirements or adapt to market changes.

In terms of business impact, this reuse paradigm can result in substantial cost savings and shorter delivery times. By tapping into existing agents, development teams can focus on strategic goals—like refining user experience or enhancing model accuracy—rather than re-developing generic functionalities. In highly competitive industries, speed-to-market can make the difference between leading or lagging. Hence, adopting an “agents as the quantum of reuse” mindset ensures that each new project not only builds on a well-established foundation, but also contributes to a broader ecosystem that drives long-term growth and a much higher and faster return on investment.

Towards Agent Scalability

There are valid concerns with parts of this approach - scale is hard, and the tooling and skills may introduce complexities. But the benefits are massive. It brings faster development cycles, more cost-effective operations, and a new level of flexibility in how enterprises automate and coordinate complex tasks. As agents become more accessible, businesses can quickly adapt their automation strategies, freeing up human talent to focus on higher-level innovation and strategy—while large ecosystems of agents handle the heavy lifting.

Agent Discovery

To make agent discovery work, first, a metadata repository is needed that catalogs agent information (see Figure 7-14). Like a data catalog, the agent registry is a searchable book-of-record for agent (and agent tool) information within the broader agent ecosystem. Agents interrogate the registry to find out information about other agents. And people search an agent marketplace – a user interface on top of the registry – to find agents that they wish to engage.

Second, you need an agent ecosystem fabric – an agentic mesh – that provides the platform not only to let agents interact and collaborate, but to let them register themselves and, ultimately, find each other.

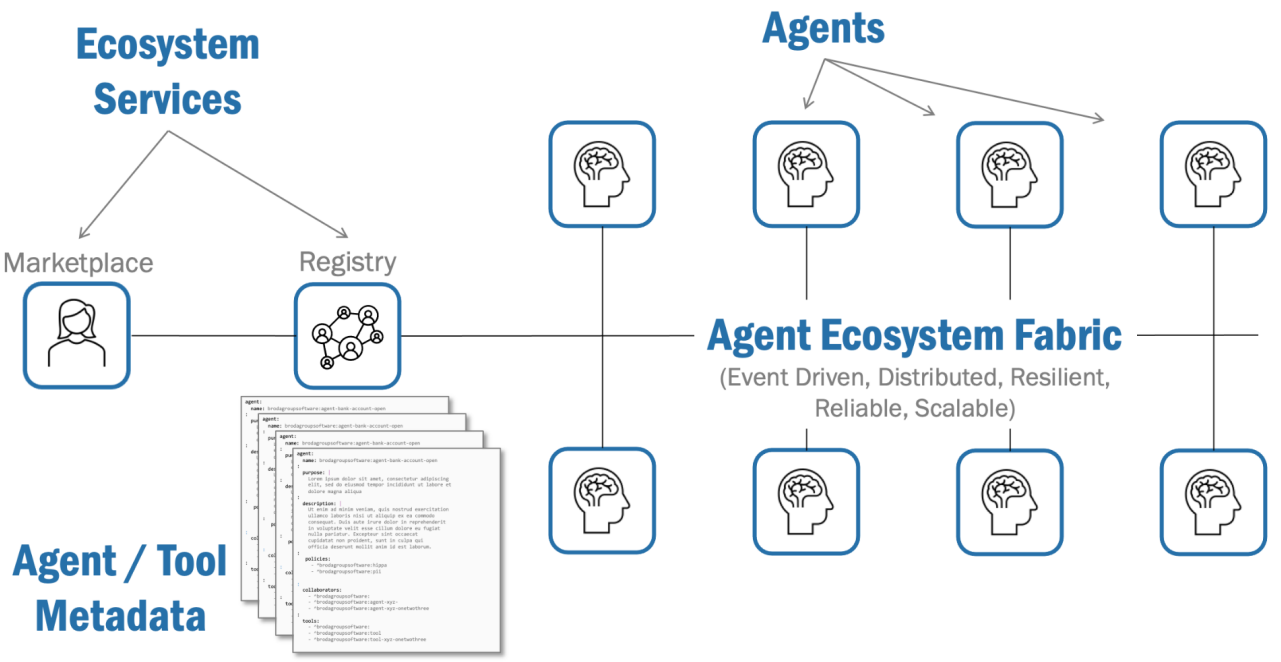


Figure 7-14. Agent discovery components

Finally, you need agents to be aware of ecosystem fabric and registry services, and then use these services to find other agents.

Beyond a Search Problem

It is tempting to frame agent discovery as a search problem. The difference with a search problem is that it identifies the top 10-100 results from potentially thousands of potential agent collaborators. But instead of the top 10-100 agents, what is needed is the ability to find the right *single* agent, for the right task at the right time.

So, we do not need search per se, but rather *relevant discovery* – a way to precisely filter the signal from the noise in the agent ecosystem (see Figure 7-15).

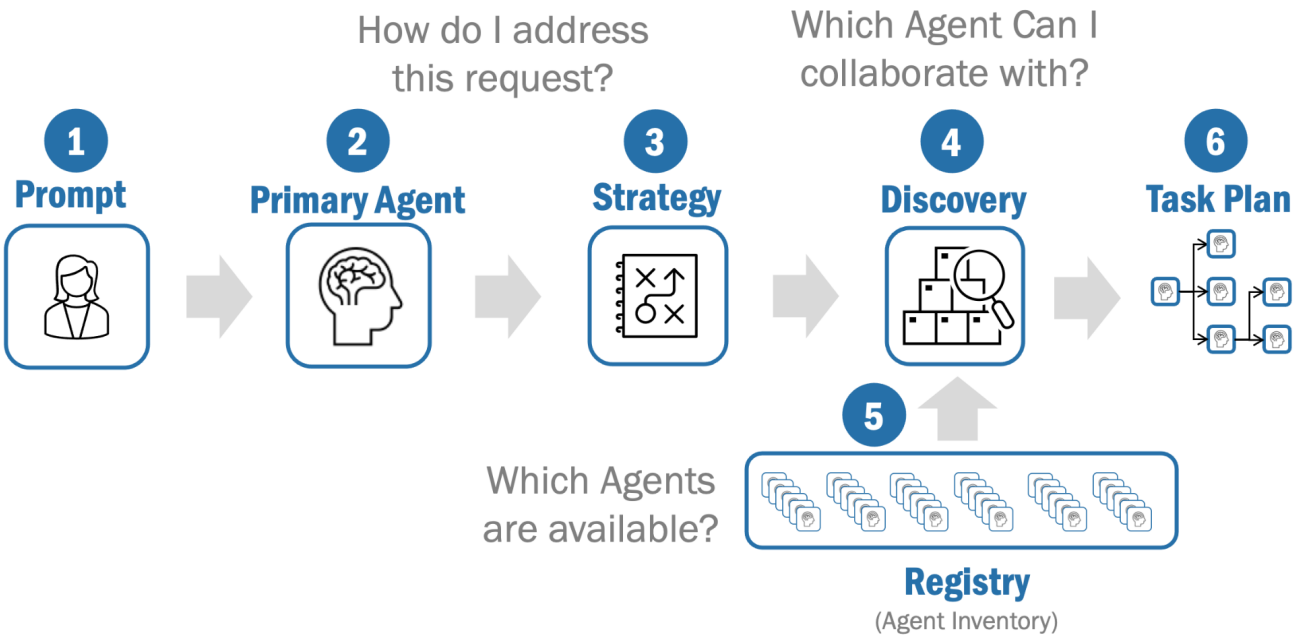


Figure 7-15. Agent discovery: finding the signal in the noise

This introduces a subtle design question: what is the most *relevant* agent? An agent doesn't just need to find a collaborating agent that can technically perform a task, rather, it needs one that aligns with the specific goals and constraints, and which adheres to the policies required by the task that needs to be fulfilled. In other words, the right agent is not necessarily the most capable, but the most contextually appropriate.

Finding the Right Agent

The practical question is how to filter the vast ecosystem of agents and find the exactly right single relevant agent collaborator that meets a specific need. We recommend two filtering approaches (see Figure 7-16), which we call

discovery scoping rules:

- *Visibility scope:* Coarse-grained filtering explicitly limits the set of agents that could be considered for collaboration.

- *Characteristics scope:* Fine-grained filtering targets a list of agents that have the exact attributes required to fulfill a given request.

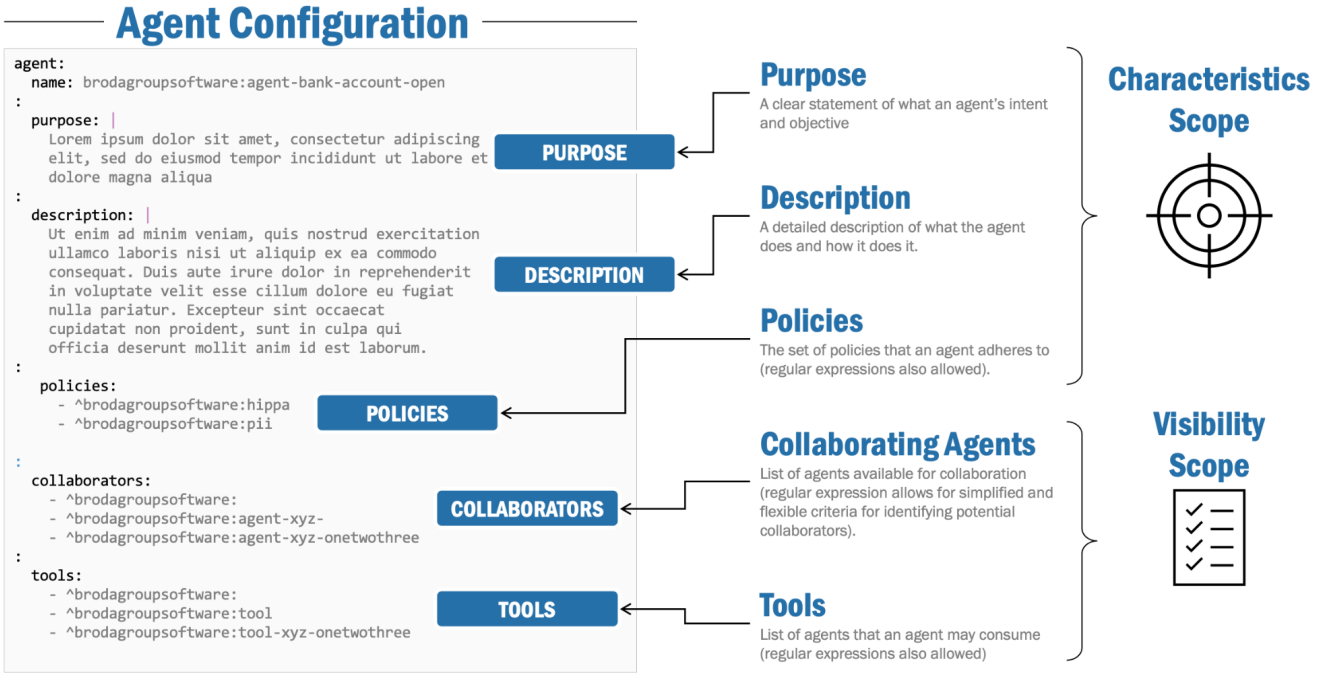


Figure 7-16. Agent discovery: visibility and characteristics scoping rules

In effect, these discovery scoping rules first provide guidance to an agent on how to identify collaborators that meet the needs of a specific task, and then provide a list of agent attributes that is required to find the right single agent. A set of attributes would minimally include things like:

- *Purpose:* The primary purpose of the agent
- *Description:* A more detailed statement about what an agent does, why it does it, and the problem that it solves
- *Policies:* A list of rules (corporate, regulatory, or other) that an agent adheres to

Visibility and characteristic scope can be implemented in two main ways:

- *Strict naming:* The list of collaborators is limited to a very specific and limited set of agents; this may work well in a situation where definitive selection and use are required (for example, in regulated industries).
- *Flexible naming:* The list of collaborators is specified using a regular expression; this lets an agent creator specify collaborators based upon criteria such as, for example, agent namespaces ("all agents in this namespace are valid collaborators").

It is our fundamental belief that as the number of agents grows, agent discovery – the ability to find the exact right agent for the right task at the right time – will become one of the defining capabilities for an agent ecosystem. Hopefully this section gives you an appreciation for an approach to address this fundamental need.

Agent Observability

Enterprises need continuous visibility into agent performance, resource usage, and error states—requirements that mirror those of any other enterprise system. This visibility is broadly captured by the term *observability*. In modern distributed systems, observability refers to the ability to measure the internal state of a system based on the data it produces: logs, metrics, and traces. It allows operators and developers to monitor system health, detect anomalies, and diagnose issues. In the absence of observability, systems become black boxes that are difficult to troubleshoot, scale, or trust.

The same challenges apply to agents, perhaps more acutely. Agents are not simple request-response applications; they often maintain long-running conversations, invoke tools, coordinate across multiple participants, and evolve their internal state. If something goes wrong—an unexpected tool failure, a misinterpreted instruction, or an unresponsive peer—identifying the source of the issue requires visibility into the agent's decision process, conversation state, and operational metrics. This is particularly important in regulated or safety-critical environments where audit trails and post-incident reviews are mandatory.

Observability and Traceability

Traceability builds on observability by enabling linkage between discrete events within a broader multi-agent (or multi-tool) conversation. In agent ecosystems, this means associating each action—a tool invocation, an agent message, a database write—with a trace or task identifier. This identifier allows system administrators and developers to follow the life of a single request as it moves through a chain of agents and tools. Without traceability, debugging distributed failures or understanding cascading behavior becomes prohibitively difficult.

Microservice-based agents, or MicroAgents, provide a viable foundation for embedding observability and traceability. Enterprise systems have spent years refining techniques to observe microservices using service meshes, distributed tracing platforms and metric aggregation tools (for example, Prometheus, Grafana). MicroAgents benefit from these practices: each agent service can be instrumented with standard libraries, expose health endpoints, and propagate trace context using HTTP headers or message metadata. These are established, well-documented techniques with broad support across programming languages and platforms.

An observability-first agent framework should ensure that all agents are instrumented to capture operational metrics by

default. These metrics might include CPU and memory usage, request latency, error rates, and tool invocation success/failure counts. Beyond simple metrics, agents should also expose state transitions, conversation depth, and pending request queues. Such data enables meaningful monitoring and capacity planning. Crucially, this instrumentation should be built into the framework itself rather than left as an afterthought or left to individual developers.

Agent Operability

Uptime, scalability, and resiliency are considered baseline expectations for enterprise systems. Meeting these demands requires rigorous attention to *operability*, the set of practices that ensures that systems can be reliably monitored, maintained, and supported. Operability encompasses availability monitoring, failure detection, system alerts, incident response protocols, and structured deployment workflows. Without it, even the most sophisticated systems can become liabilities, prone to unpredictable failures and difficult to support.

Agents introduce new complexities into the domain of operability. Unlike traditional services that respond to discrete requests, agents often engage in long-running,

stateful interactions. These interactions may involve multiple tools, span across systems, and evolve based on learned behaviors. As a result, it becomes essential to ensure that agents emit health signals, respond to liveness probes, and generate alerts when anomalies or failures occur. Equally important is integration with existing enterprise operations tools to support timely diagnosis and intervention when problems arise.

Effective operability requires logging and auditing to be designed with care, especially given the sensitive nature of the data agents may process. Agents that interact with humans or financial systems might encounter personally identifiable information (PII), health records, or payment data. Logging practices must comply with privacy and regulatory standards such as GDPR or PCI-DSS. This means that logs should redact or anonymize sensitive content, use access-controlled storage, and avoid recording excessive details that could later be exfiltrated or misused. Secure audit trails are necessary for both internal oversight and external compliance.

Microservice-based agents, or MicroAgents, offer a practical path to high-operability agent systems. They align naturally with the DevOps (or DevSecOps, which is DevOps including security) model, which integrates development, security, and operations into a continuous lifecycle. Decades of enterprise

experience managing microservices can be directly applied: container health checks, service monitoring, automated deployments via CI/CD pipelines, and rollout strategies such as blue/green or canary deployments. MicroAgents benefit from existing tooling ecosystems and standard protocols for observability, alerting, and rollback, making operational management tractable and reliable.

To formalize these practices, we recommend a specialized discipline: AgentOps. AgentOps builds on DevSecOps and LLMOps but addresses the unique lifecycle needs of agents. It emphasizes a strong developer experience, ensuring that agents are modular, composable, and testable. It supports controlled migration of agents from development to production, robust versioning strategies, and dynamic routing between agent versions. Through AgentOps, enterprises can ensure that their agent systems remain operable under load, secure by design, and agile in the face of evolving requirements.

Summary

We started this chapter highlighting NVIDIA CEO Jensen Huang's and Microsoft CEO Satya Nadella's visions for a thriving ecosystem of potentially millions of agents. We believe in this vision, and hence we also believe it is in our

interests to prepare for this outcome. And, so, we suggest that enterprise-grade agents are not just foundational but mandatory.

We are hopeful that you see how MicroAgents - agents built upon a microservices foundation - provide an architecture that is not only reliable and scalable, but also secure, discoverable, observable, and operable. In other words, MicroAgents are enterprise-grade, and ready to support the grandest of agent visions.

A tthis point we have spent several chapters describing first, agents, and then enterprise-grade agents. The next chapter will describe how our enterprise-grade agents run in an enterprise-grade agent ecosystem, which we call Agentic Mesh.