DENDRO LOGIC
INTELLIGENT DATA

Designing Multi-Agent AI Systems

Architecture, Frameworks, and Production Deployment for Teams Building AI-Powered Business Systems

Helping technical teams design, deploy, and govern multi-agent systems that integrate safely with existing business infrastructure

By Mike McGreal

Dendro Logic Ltd
March 2026
Business AI Adoption Playbook

https://dendro-logic.com

Contents

Part 1: What Multi-Agent Systems Are and Why They Matter

What This Playbook Covers

This is the second document in the Dendro Logic AI Adoption Playbook Series. The first covered AI agents in development teams, working with tools like Claude Code and GitHub Copilot. The third covers AI adoption for non-technical staff. This document sits between the two: it is for the architects, engineers, and technical leaders who are designing and building AI-powered systems that will be integrated into business operations.

If you are evaluating whether to build multi-agent workflows, choosing between orchestration frameworks, integrating AI into legacy systems, or trying to make agentic systems reliable enough for production, this playbook is for you. It covers architecture patterns, the framework landscape (LangGraph, CrewAI, AutoGen, OpenAI Agents SDK), the Model Context Protocol (MCP) for tool integration, guardrails, evaluation, observability, troubleshooting, and production deployment.

This is not a getting started tutorial. It assumes you have experience building software systems and some familiarity with LLMs. It is a reference for making the engineering decisions that determine whether your multi-agent system works in production or only works in demos.

What Is a Multi-Agent System

A multi-agent system is an architecture where multiple specialised AI agents collaborate to complete tasks that would be too complex, too risky, or too context-heavy for a single agent. Each agent has a defined role, a constrained set of tools, and a specific area of responsibility. An orchestrator or supervisor coordinates their work, routes tasks, and manages the flow of information between them.

This mirrors how teams of people work. You do not ask one person to handle sales, legal review, financial analysis, and customer support simultaneously. You assign specialists. Multi-agent systems apply the same principle to AI: a planning agent breaks down the task, specialist agents execute their part, a critic agent reviews the output, and a supervisor ensures everything stays within bounds.

The distinction between a multi-agent system and a workflow is important. If your system calls tools sequentially in a fixed flow with no decision-making, it is a workflow, not an agent. True agentic behaviour means the system can plan, decide which tools to use, observe the results, and adjust its approach. The complexity (and risk) comes from this

autonomy, which is why guardrails and observability are not optional additions but core architecture requirements.

Why Multi-Agent, Not Single Agent

Single agents work well for proof-of-concepts and simple tasks. They break down when tasks require concurrency, when different parts of the workflow need different tool access or permission levels, when you need regulatory isolation between functions, or when the context window becomes a bottleneck.

A single agent handling both customer data retrieval and financial transactions has access to everything, making security boundaries impossible to enforce. A multi-agent system can give the customer-data agent read-only access to the CRM and give the payment agent write access to the billing system, with neither able to reach the other's tools. This is not just good practice, it is a requirement under UK data protection regulations and the principle of least privilege.

Multi-agent systems also improve reliability through separation of concerns. When an agent fails, the failure is contained. A broken data retrieval step does not corrupt the payment processing step. Each agent can be tested, monitored, and updated independently, which is exactly how you would design a microservices architecture, because the same principles apply.

The Current State of the Industry

Gartner predicts that 40% of enterprise applications will be managed by task-specific AI agents by the end of 2026. The market for multi-agent systems is projected to grow to $184.8 billion by 2034. McKinsey reports that 62% of organisations are experimenting with agentic workflows. But experimentation and production are very different things. Deloitte's State of AI in the Enterprise 2026 survey found that while worker access to AI rose 50% in 2025, only 34% of companies are truly reimagining their business with it. Most are still in the pilot stage.

The gap between demos and production is where most multi-agent projects fail. The frameworks are mature enough. The models are capable enough. The problem is almost always engineering: lack of guardrails, no observability, untested failure modes, and no clear deployment strategy. This playbook addresses each of these.

> **The Pattern That Works**
>
> Start with single agents when you are building proof-of-concepts. Move to multi-agent when tasks require concurrency, isolation, or regulation. The organisations that succeed treat agents like distributed systems: observable, composable, and compliant.

Part 2: Architecture Patterns

Multi-agent architectures follow a small number of well-established patterns. Choosing the right one depends on how much autonomy your agents need, how structured the workflow is, and what your failure tolerance looks like.

Orchestrator-Worker

An orchestrator agent receives the task, breaks it into subtasks, and delegates each to a specialised worker agent. Workers execute independently and return results to the orchestrator, which synthesises the final output. This is the most common pattern for structured, decomposable tasks like report generation, research synthesis, or multi-step data processing.

In LangGraph, this is implemented using the Send API to dynamically create worker nodes. The orchestrator produces a plan (often using structured output to generate a list of sections or tasks), then uses Send to dispatch each task to a worker node with its own state. Workers run in parallel and their outputs are aggregated by a synthesiser node.

Strengths: clean separation of planning and execution, easy to parallelise, straightforward to add new worker types. Weaknesses: the orchestrator is a single point of failure, and the quality of the plan directly determines the quality of the output.

Supervisor (Router)

A supervisor agent acts as a central router, receiving all inputs and directing them to the appropriate specialist agent based on intent classification. Unlike the orchestrator-worker pattern, the supervisor does not decompose tasks, it routes them whole. This is the standard pattern for customer service systems where a query might need to go to a billing agent, a technical support agent, or a returns agent.

The supervisor typically uses an LLM to classify intent, then hands off to the appropriate agent. Handoffs can include conversation context so the specialist agent has the full history. The supervisor also handles fallback: if a specialist cannot resolve the task, it routes back to the supervisor for re-classification or escalation to a human.

In OpenAI's Agents SDK, handoffs are a first-class concept. An agent can delegate to another agent for specific tasks, and the conversation history transfers seamlessly. In LangGraph, you implement this with conditional edges that route based on the supervisor's classification output.

Planner-Executor-Critic

This three-stage pattern adds a quality gate. The planner agent creates a plan. The executor agent carries it out. The critic agent reviews the output and either approves it or sends it back for revision. This is the pattern you use when output quality matters more than speed, common in content generation, code review, compliance checking, and any domain where errors have consequences.

The critic-refiner loop can iterate multiple times, with each pass improving the output. To prevent infinite loops (a real production risk), you must set a maximum iteration count. A typical implementation allows two to three revision cycles before either accepting the best output or escalating to a human reviewer.

Map-Reduce

For tasks that can be broken into independent subtasks of the same type, map-reduce is the natural pattern. A coordinator fans out the work to multiple identical agents (map), each processes its portion independently, and the results are aggregated (reduce). This is ideal for processing large document sets, analysing multiple data sources, or running the same analysis across different time periods or regions.

LangGraph's Send API makes this straightforward. The map step generates a list of items, each dispatched to a worker node via Send. All workers run in parallel with isolated state. The reduce step collects all outputs using an annotated state key with an operator like operator.add.

Hierarchical (Director-Pod)

For enterprise-scale systems, a hierarchical pattern introduces director agents that manage groups of specialist agents (pods). A top-level orchestrator delegates to domain directors (finance, legal, operations), each of which manages its own team of agents. This mirrors organisational structure and allows different governance rules per domain.

This is the pattern you need when different parts of the system have different compliance requirements, when teams need to own and maintain their own agents independently, or when the system is too large for a single supervisor to manage effectively. It adds complexity, but it is the only architecture that scales to enterprise-wide deployment with proper governance.

> ### Choosing the Right Pattern
>
> Orchestrator-worker for structured, decomposable tasks. Supervisor for routing and classification. Planner-executor-critic when quality matters more than speed. Map-reduce for parallel processing of similar items. Hierarchical for enterprise scale with domain isolation. Start with the simplest pattern that works and add complexity only when you have evidence it is needed.

Part 3: The Framework Landscape

Six frameworks currently dominate the multi-agent AI space. Each serves different use cases, and choosing the right one is one of the most consequential early decisions in a project. The wrong choice creates technical debt that is expensive to unwind.

LangGraph

LangGraph, built by LangChain, is a low-level orchestration framework for building stateful, long-running agents. It is trusted by companies including Klarna, Replit, and Elastic. LangGraph gives you fine-grained control over agent behaviour through a graph-based programming model where nodes are functions and edges define the flow between them.

Key capabilities: durable execution with checkpointing (state survives process restarts), human-in-the-loop via interrupt points, streaming support for real-time output, the Send API for dynamic parallel execution, and built-in support for retry policies. LangGraph compiles workflows into a graph that can be inspected, tested, and deployed as a unit.

For persistence, LangGraph supports in-memory checkpointing for development and PostgreSQL for production. The checkpointer captures the full state at every node transition, enabling time-travel debugging, exactly-once execution, and the ability to resume interrupted workflows. This is critical for production systems where processes may be interrupted by deployments, crashes, or scaling events.

When to choose LangGraph: when you need maximum control over agent behaviour, when your workflows are complex with conditional routing and parallel execution, when you need production-grade persistence and human-in-the-loop, or when you are already in the LangChain ecosystem. LangGraph is the right choice for most enterprise multi-agent systems because it treats agents as infrastructure, not magic.

CrewAI

CrewAI is a higher-level framework that models multi-agent systems as crews of agents with defined roles, goals, and backstories. It is designed for rapid prototyping and for teams that want to describe agent behaviour declaratively rather than imperatively. CrewAI abstracts away much of the graph-level complexity in favour of a role-based programming model.

Key capabilities: role-based agent definition with natural language descriptions, task guardrails (both code-based validation functions and LLM-based natural language validators), built-in memory and knowledge systems, a Flows API for structured multi-

step workflows, and CrewAI Enterprise for managed deployment with authentication and monitoring.

CrewAI's guardrail system deserves attention. You can define validation criteria as either Python functions that check output programmatically (length, format, required fields) or as natural language strings that the LLM evaluates against the output (tone, completeness, accuracy). This dual approach gives you both deterministic and semantic quality checks.

When to choose CrewAI: when you want to move from concept to working prototype quickly, when your team prefers declarative role definitions over graph programming, when you need built-in guardrails without writing custom validation infrastructure, or when you plan to use CrewAI Enterprise for managed deployment. The trade-off is less fine-grained control than LangGraph.

Microsoft AutoGen and Agent Framework

AutoGen, born from Microsoft Research, was redesigned from scratch in version 0.4 (January 2025) with an asynchronous, event-driven architecture. It comprises three layers: a Core layer for event-driven messaging and agent lifecycle, an AgentChat layer for high-level conversation patterns, and an Extensions layer for third-party integrations including Azure services.

In October 2025, Microsoft released the Microsoft Agent Framework in public preview, merging AutoGen's orchestration capabilities with Semantic Kernel's production foundations. This unified framework supports both Python and .NET, provides native integration with Azure AI Foundry for cloud deployment, and includes enterprise features like identity management, governance, and autoscaling.

The Azure AI Foundry platform offers a Control Plane for fleet-wide agent management: monitoring health, cost, and performance across all agents in real time, enforcing policies and guardrails from a single dashboard, and lifecycle management to pause, update, or retire agents. Hosted agents in Foundry Agent Service can deploy agents built with LangGraph, CrewAI, or other frameworks, not just AutoGen.

When to choose AutoGen/Microsoft Agent Framework: when you are in the Microsoft ecosystem (Azure, .NET, Microsoft 365), when you need enterprise-grade governance, compliance, and identity management out of the box, or when fleet-wide agent observability through Azure AI Foundry is a requirement.

OpenAI Agents SDK

Released in March 2025 as the production evolution of the experimental Swarm framework, the OpenAI Agents SDK provides a focused, opinionated approach to multi-

agent systems built around four primitives: Agents (LLMs with instructions and tools), Handoffs (delegation between agents), Guardrails (input/output validation), and Sessions (automatic conversation history). It is designed specifically for applications using OpenAI models.

The SDK emphasises simplicity. Handoffs enable seamless task delegation based on capabilities, with conversation context transferring automatically. Guardrails validate both inputs and outputs, preventing harmful or incorrect responses. The focus on sessions makes stateful multi-turn conversations straightforward.

When to choose OpenAI Agents SDK: when you are building exclusively on OpenAI models and want the tightest possible integration, when handoff-based agent routing matches your use case (customer service, sales, support triage), or when you prioritise simplicity and developer experience over framework flexibility.

Framework Selection Decision

The framework choice should be driven by four factors: your existing technology stack, the level of control you need, your deployment target, and your team's experience. If you are on Azure, the Microsoft Agent Framework gives you the most integrated path to production. If you need maximum control and your team is comfortable with graph-based programming, LangGraph is the strongest choice. If you want to prototype quickly with role-based agents, CrewAI gets you there fastest. If you are building exclusively on OpenAI, their SDK is the most direct path.

A common and effective approach is to prototype in CrewAI for speed, then migrate to LangGraph for production when you need finer control over execution flow, persistence, and deployment. The patterns transfer between frameworks even when the syntax does not.

n8n and Low-Code Agent Orchestration

Not every multi-agent workflow needs to be built from code. n8n has become one of the most popular platforms for building AI-powered workflows with a visual, low-code approach, and it deserves serious consideration for teams that need to move quickly or involve non-developers in workflow design. n8n combines AI capabilities with business process automation, giving technical teams the flexibility of code with the speed of visual design. It integrates with over 400 applications out of the box and supports self-hosting for full data control.

n8n's AI agent capabilities include dedicated nodes for connecting to OpenAI, Anthropic Claude, and local LLMs, a visual builder for designing multi-step agent workflows, webhook and scheduled triggers for event-driven execution, human-in-the-loop nodes

that pause workflows for manual approval (with 10 built-in communication channels), conversation memory management for chatbot-style agents, and a Code node for JavaScript or Python when you need custom logic. n8n also provides an MCP server, meaning you can call your n8n workflows from other AI systems that support the Model Context Protocol.

For production deployment, n8n supports queue mode with worker nodes for horizontal scaling. A single instance handles thousands of executions daily, while queue mode with multiple workers can process approximately 72 requests per second with sub-3-second latency. Self-hosted n8n on Kubernetes with PostgreSQL and Redis provides enterprise-grade reliability. The platform includes role-based access control, credential management for API keys, and workflow version history.

Where n8n fits and where it does not: n8n excels at structured automation workflows where agents route messages, process documents, update CRMs, and trigger downstream actions. It is ideal for teams that want to build and iterate quickly on business process automation with AI. It is less suited for complex autonomous agents that need persistent memory across sessions, multi-step reasoning, or dynamic planning. For those use cases, LangGraph or CrewAI remain the better foundation, though n8n can serve as the orchestration and trigger layer that invokes those more sophisticated agents.

A practical architecture uses n8n as the trigger and routing layer, calling out to LangGraph or CrewAI agents for complex reasoning, and handling the business process automation (updating records, sending notifications, filing outputs) in the n8n workflow. This combines the visual design and integration breadth of n8n with the agent sophistication of purpose-built frameworks.

n8n as a Gateway, Not a Replacement

n8n is excellent for building the glue between your agents and your business systems. Use it for triggers, routing, human-in-the-loop approvals, and output handling. Use LangGraph or CrewAI for the agent reasoning itself. This hybrid approach gives you the best of both worlds: visual workflow design for business logic and programmatic control for agent behaviour.

Do Not Over-Engineer the Framework Choice

The framework matters less than the architecture. A well-designed supervisor pattern will work in any framework. A poorly designed system will fail in all of them. Spend 80% of your decision-making time on architecture and 20% on framework selection.

Part 4: MCP - The Universal Connector

The Model Context Protocol (MCP) is the single most important development in AI agent infrastructure in 2025. Introduced by Anthropic in November 2024, MCP is an open standard for connecting AI systems to external data sources and tools. Within twelve months, it was adopted by OpenAI, Google DeepMind, Microsoft, and thousands of developers, and was donated to the Linux Foundation's Agentic AI Foundation in December 2025.

What MCP Solves

Before MCP, connecting an AI agent to a business tool required a custom integration for every combination of model and tool. If you had 10 agents and 100 tools, you potentially needed 1,000 different integrations. MCP provides a universal protocol: build one connector for each tool, and every MCP-compatible agent can use it. The analogy is USB-C for AI, a single standard that replaces a mess of proprietary connectors.

MCP uses JSON-RPC 2.0 and takes architectural inspiration from the Language Server Protocol (LSP) that standardised how IDEs communicate with language tooling. It defines a client-server model where AI applications (clients/hosts) connect to MCP servers that expose tools, resources, and prompts. The protocol supports bidirectional communication, allowing servers to provide context to agents and agents to invoke tools on servers.

MCP and Legacy System Integration

This is where MCP becomes critical for businesses with existing systems. Rather than rebuilding your CRM, ERP, or financial systems to be AI-native, you build an MCP server that wraps each system's API. The MCP server translates between the agent's natural language tool calls and your system's specific API format. Your legacy systems do not change. The agent interacts with a clean, standardised interface.

Anthropic provides reference MCP server implementations for common enterprise systems including Google Drive, Slack, GitHub, Git, Postgres, and Puppeteer. The community has built thousands more, with over 5,800 MCP servers now available. For internal systems without existing MCP servers, building a custom server is straightforward: define the tools your system exposes, implement the handlers that translate tool calls to API calls, and register the server with your agent's MCP client.

MCP Architecture for Enterprise

```
MCP Architecture Overview:

Agent (Client) <--JSON-RPC--> MCP Server A (CRM)
                <--JSON-RPC--> MCP Server B (ERP)
                <--JSON-RPC--> MCP Server C (Database)
                <--JSON-RPC--> MCP Server D (SharePoint)

Each server exposes tools, resources, and prompts.
Agent discovers available tools at connection time.
Agent calls tools by name with parameters.
Server executes and returns results.
```

Scaling MCP: Code Execution Mode

As MCP usage scales, two patterns increase agent cost and latency: every tool definition loaded into context consumes tokens, and intermediate tool results pass through the model even when the agent just needs to move data between systems. Anthropic's engineering team published research showing that agents scale better by writing code to call tools instead of calling them directly.

In this mode, the agent writes a script that calls MCP tools programmatically, processes the data locally, and only returns the final result to the model. For example, instead of the model reading a full meeting transcript from Google Drive (50,000 tokens), processing it, and then writing it to Salesforce (another 50,000 tokens), the agent writes a script that reads the file, extracts the relevant section, and writes it to Salesforce in a single code execution step. The model never sees the raw data. Cloudflare published similar findings, referring to this as Code Mode.

MCP Security Considerations

MCP's power comes with real security risks. A security analysis published in April 2025 identified multiple concerns: prompt injection through tool descriptions, tool permissions that allow combining tools to exfiltrate data, and lookalike tools that can silently replace trusted ones. These are not theoretical risks.

For enterprise deployment, implement these controls: gate sensitive tools behind explicit allow policies and human approvals, use short-lived secrets and zero standing privileges for tool credentials, require signature verification before agents can load or invoke tools, instrument all tool calls with OpenTelemetry for audit trails, and classify logs with PII masking. Treat every tool as a potential escalation path and default to deny with explicit allow rules per agent.

The November 2025 MCP specification update added critical enterprise features including asynchronous operations, server identity verification, and authorisation extensions. Cross App Access, incorporated as an MCP authorisation extension, gives organisations the oversight and access control needed for secure deployment.

MCP Is Now Industry Infrastructure

MCP was donated to the Linux Foundation in December 2025 with backing from Anthropic, OpenAI, Google, Microsoft, AWS, Block, Bloomberg, and Cloudflare. It has over 97 million monthly SDK downloads. This is not an experiment. It is the standard. Build your integration layer on MCP.

Part 5: Building Reliable Agents

The difference between a demo agent and a production agent is reliability. Demos work because the inputs are controlled. Production fails because inputs are unpredictable, context is incomplete, tools return errors, and agents make decisions that no one anticipated. This section covers the engineering practices that make agents reliable enough to deploy.

Guardrails: The Non-Negotiable Layer

Guardrails are the constraints that keep agents acting within defined boundaries. They are not optional safety features added at the end. They are core architecture components that should be designed before the first line of agent code is written. Without guardrails, agents will eventually hallucinate an action, access data they should not, or enter an infinite loop. The question is not if, but when.

Guardrails operate at three levels. Input guardrails validate and sanitise what goes into the agent: prompt injection prevention, input length limits, PII detection and masking, and schema validation for structured inputs. Processing guardrails constrain what the agent can do: tool access restrictions (RBAC/ABAC), maximum iteration counts to prevent infinite loops, cost caps per execution, and timeout limits. Output guardrails validate what comes out: content moderation, format validation, fact-checking against source data, and human approval gates for high-risk actions.

The most important guardrail in practice is the maximum step count. Agents that can loop indefinitely will loop indefinitely. Set a hard cap on the number of steps an agent can take, and define what happens when that cap is reached: return the best output so far, escalate to a human, or fail gracefully with an explanation. In LangGraph, this is implemented with a step counter in the state that triggers a conditional edge to an exit node.

Memory and State

Agent memory comes in three categories, each with different engineering requirements. Ephemeral context is the current conversation or task context, stored in the agent's state and discarded when the task completes. Short-term memory persists across turns within a session, implemented through checkpointing (LangGraph's MemorySaver for development, PostgresSaver for production). Long-term memory persists across sessions, typically stored in a vector database or structured store.

LangGraph's checkpointing system captures the full state at every node transition. This enables time-travel debugging (replay any point in the execution), exactly-once semantics (resume from the last checkpoint after a crash), and human-in-the-loop

workflows (pause execution, let a human review, then resume). For production, always use a database-backed checkpointer. In-memory checkpointers lose all state when the process restarts.

LangGraph also provides a runtime Store for persistent key-value storage. This is useful for per-user preferences, accumulated knowledge, or any data that should survive across sessions. The Store supports namespacing by user ID, making multi-tenant deployments straightforward.

Project Memory for Multi-Agent Systems

Document 1 of this series covers project memory in detail for development agents (CLAUDE.md, AGENTS.md, .cursorrules). The same principles apply to multi-agent systems, but the implementation is different because your agents are not operating inside an IDE, they are operating inside an orchestration framework.

Every multi-agent system should have a project knowledge layer that agents can reference. This typically includes a northstar document (docs/northstar.md or equivalent) that defines the business context, goals, and constraints of the system, an architecture document that describes how components fit together, a decisions log that records why specific design choices were made, and domain-specific reference material that agents need to produce accurate output. Store these in a docs/ folder within your project and make them accessible to agents through an MCP filesystem server or by loading them into the agent's context at the start of each workflow.

For multi-agent systems specifically, each agent should have its own persona configuration that references the relevant subset of project knowledge. The billing agent does not need the full architecture document, but it does need the billing domain model and the current pricing rules. The research agent needs access to external documentation servers (like Context7) but not to the payment processing API. Scoping knowledge per agent reduces context window consumption and improves output quality.

Reducing Hallucinations with Documentation Servers

One of the most common failure modes in agent systems is hallucinated APIs, function signatures, configuration options, and library features. The agent confidently generates code that calls a function that does not exist, or uses an API parameter that was removed three versions ago. This happens because LLMs are trained on a snapshot of documentation, and that snapshot is always out of date.

Context7 is an MCP server specifically designed to solve this problem. It fetches current, version-specific documentation for programming libraries and frameworks on demand. When an agent needs to call an API or use a library feature, it queries Context7 for the

current documentation rather than relying on its training data. This dramatically reduces hallucinated APIs and incorrect parameter usage.

The research-first workflow pattern, covered in Document 1 of this series, applies directly to multi-agent systems. Before any agent writes code or generates output that depends on external libraries, APIs, or data schemas, it should check the relevant documentation first. In a multi-agent system, this can be implemented as a dedicated research agent that queries documentation servers and provides verified reference material to the worker agents. The research agent's output becomes a trusted context that worker agents use as their source of truth.

```
MCP Configuration for Documentation:

  "context7": {
    "url": "https://mcp.context7.com/mcp"
  }

Usage in agent prompt:
  "Before implementing any external library integration,
   query context7 for the current API documentation.
   Do not rely on training data for library-specific
   function signatures or configuration options."
```

Context Window Management

Context window exhaustion is one of the most insidious failure modes in multi-agent systems because it degrades gracefully rather than failing obviously. As the context fills up, the agent starts losing track of early instructions, forgetting constraints, and producing output that drifts from the original task. By the time a human notices, the damage is done.

Design your agents to manage context proactively. For long-running workflows, implement summarisation checkpoints where the agent's accumulated context is summarised and the full history is discarded. For parallel agent architectures, each worker agent should have isolated context rather than sharing a growing shared state. For agents that process large documents, extract the relevant sections before passing them to the agent rather than loading the entire document into context.

Monitor context usage in your observability stack. Track the percentage of context window used at each step. Set alerts when agents regularly exceed 70% of their context window, as this is where quality degradation typically begins. If an agent consistently needs more context than its model supports, the task needs to be decomposed further or a larger context model needs to be used for that specific agent.

Deterministic Enforcement

A lesson from Document 1 that applies even more strongly to multi-agent systems: prompt-based instructions are suggestions, not rules. An agent told in its system prompt to never access production data will, eventually, access production data if the technical controls allow it. Guardrails that exist only in prompts are not guardrails at all.

Deterministic enforcement means implementing constraints in code, not in language. Maximum step counts enforced by a counter in the graph state, not by telling the agent to stop after N steps. Tool access enforced by the tool registry, not by telling the agent which tools to use. Output format enforced by structured output parsing and validation, not by asking the agent to output JSON. Cost limits enforced by a budget tracker that terminates execution, not by asking the agent to be efficient.

In CI/CD pipelines, this translates to pre-commit hooks that validate agent output format, automated tests that verify guardrail enforcement (attempt to violate each constraint and confirm it is blocked), deployment gates that require all E2E tests to pass before any agent change reaches production, and scheduled audits that verify permissions have not drifted. The same principle from Document 1 applies: text rules in prompts do not work reliably. Code rules in hooks do.

Self-Verification Loops

The planner-executor-critic pattern described in Part 2 is the architectural expression of a broader principle: agents should verify their own output before presenting it. A self-verification loop means the agent generates output, runs automated checks against it, and iterates if the checks fail, all before any human sees the result.

For code-generating agents, self-verification means running the test suite against generated code and iterating on failures. For data-processing agents, it means running validation checks on the output data (schema compliance, range checks, null checks). For content-generating agents, it means evaluating the output against the original requirements and checking for hallucinated facts against source material.

The key implementation detail is that the verification must be automated and deterministic. An agent evaluating its own output using the same LLM is better than nothing, but it shares the same failure modes as the generation step. Automated tests, schema validators, and fact-checking against retrieved documents provide genuinely independent verification. Use LLM-based evaluation as a supplement, not a substitute, for deterministic checks.

Human-in-the-Loop

Human-in-the-loop (HITL) is not a feature you add when things go wrong. It is a core design pattern for any agent system that takes actions with real-world consequences. The architecture should define exactly where humans are required, where they are optional, and where the agent can act autonomously.

The Stanford AI Index adapted the SAE (Society of Automotive Engineers) autonomy framework for enterprise AI agents. Level 0-1: AI provides suggestions only, humans approve everything. Level 2: agents execute specific tasks with human supervision and explicit approval before acting. Level 3: agents act autonomously within defined boundaries with human oversight for exceptions. Level 4: fully autonomous operation with human review of outcomes.

Most production systems should operate at Level 2 for high-risk actions (financial transactions, data mutations, external communications) and Level 3 for low-risk actions (internal search, summarisation, draft creation). Level 4 is appropriate only for read-only operations where the worst case is a wrong answer that will be verified by a human before acting on it.

In LangGraph, human-in-the-loop is implemented using interrupt points. The graph pauses execution at a designated node, sends the current state to a human reviewer (through whatever interface you build), and resumes when the human approves, modifies, or rejects the output. The checkpointer ensures the state is preserved during the pause, even if the process restarts.

Strict Persona Design

Every agent in a multi-agent system must have a tightly defined persona that constrains its behaviour. A persona is not just a system prompt, it is a complete specification of what the agent is, what it can do, what it cannot do, and how it communicates. Poorly defined personas are one of the most common causes of agents producing off-target, inconsistent, or unsafe outputs.

A production-grade persona definition includes: the agent's role and area of expertise (what it knows about), its specific tools and data access (what it can interact with), explicit boundaries (what it must refuse or escalate), its communication style and tone, the format and structure of expected outputs, and the circumstances under which it should hand off to another agent or a human. In CrewAI, this maps directly to the agent's role, goal, and backstory fields. In LangGraph, it is encoded in the system prompt and enforced through tool access controls and conditional edges.

The key principle is: if it is not in the persona, the agent should not do it. An agent designed to handle customer billing enquiries should refuse to discuss product roadmaps, even if it could generate a plausible answer. An agent designed to summarise

documents should not attempt to make decisions based on those documents. Strict persona boundaries prevent agents from drifting into territory where their outputs are unreliable or dangerous.

Test personas adversarially. Attempt to get the agent to act outside its defined role. Ask it questions outside its domain. Try to get it to use tools it should not have access to. If the persona holds under adversarial testing, it will hold in production. If it does not, the boundaries are not strict enough.

Audit Trail Design

Every action an agent takes must be logged in a way that allows complete reconstruction of what happened, why, and with what data. This is not optional, it is a regulatory requirement under UK GDPR for automated decision-making, a compliance requirement for most enterprise deployments, and a practical necessity for debugging and improvement.

A complete audit trail captures: the triggering event (what initiated the agent, when, and from where), the full input (the data or request the agent received, with PII masked or tokenised), every tool call (which tool, what parameters, what was returned), every LLM call (the prompt sent, the model used, the response received, token counts, latency), every decision point (which path the agent chose and why), the final output (what was delivered to the user or downstream system), and the outcome (was it accepted, rejected, modified, or escalated by a human reviewer).

Store audit logs in structured JSON format with correlation IDs that trace a single request across all agents, tools, and systems. Use separate storage from your application database, append-only and immutable. Set retention periods based on your compliance requirements (UK GDPR requires records demonstrating lawful processing, typically retained for 6 years). Stream logs to your SIEM (Sentinel, Splunk, Elastic) for anomaly detection and automated alerting.

LangSmith provides automatic tracing of all LangGraph and LangChain operations. For custom systems, instrument with OpenTelemetry's Generative AI semantic conventions, which define standard span attributes for prompts, responses, model names, token counts, tool calls, and safety filter outcomes. This gives you vendor-neutral observability that works with any monitoring platform.

Strict Permissions Design

Agents interact with business systems using credentials. The permissions granted to those credentials determine what the agent can read, write, modify, and delete. The principle of least privilege is not a suggestion, it is the single most important security

control for production agents. Every agent should have the minimum permissions required to complete its task, and nothing more.

For Microsoft 365 environments, this means creating dedicated service accounts or app registrations for each agent with scoped Graph API permissions. A meeting summarisation agent needs Calendars.Read and OnlineMeetings.Read, it does not need Mail.Send or Files.ReadWrite.All. A document search agent needs Files.Read.All within specific SharePoint sites, not Sites.FullControl. Register each agent as a separate application in Azure Entra ID with only the permissions it requires. Use application permissions (not delegated) for background agents, and delegated permissions for agents that act on behalf of a specific user.

For database access, create read-only database users for agents that only need to query data. Never give an agent the same credentials as your application's write user. Use row-level security and column-level permissions where supported. If an agent needs to write data, scope the write permissions to specific tables and operations. Log every database query the agent executes.

For API access to internal systems (CRM, ERP, ticketing), create API keys or service accounts with role-based access. Use short-lived tokens where possible. Rotate credentials on a defined schedule. Never store credentials in prompts, agent memory, or environment variables, use a secrets manager (HashiCorp Vault, Azure Key Vault, AWS Secrets Manager) with automatic rotation.

Review permissions quarterly. As agent capabilities evolve, permissions tend to accumulate. A permission that was needed for an early prototype may be unnecessary (and dangerous) in the current version. Audit which permissions each agent actually uses versus what it has been granted, and revoke anything unused.

Internal Triggers and Event-Driven Agents

Agents do not always need to be invoked by a user. In many enterprise workflows, agents should activate in response to internal events: a new record in the CRM, a file uploaded to SharePoint, a support ticket escalated to a specific queue, a webhook from a CI/CD pipeline, or a scheduled time. Internal triggers turn agents from tools you ask into systems that act.

n8n provides the most accessible trigger infrastructure, with webhook triggers, scheduled triggers, and application event triggers for hundreds of platforms. For code-based systems, the same patterns apply: subscribe to message queues (RabbitMQ, AWS SQS, Azure Service Bus), listen for webhook callbacks, poll APIs on a schedule, or react to database change data capture events.

When designing internal triggers, apply the same governance principles as any other agent invocation. Every triggered execution should be logged with the same audit trail as a user-initiated request. Rate-limit triggers to prevent cascade effects (a single event should not spawn hundreds of agent executions). Define circuit breakers that disable triggers if error rates exceed thresholds. Ensure that triggered agents have the same guardrails, permissions, and human escalation paths as interactively invoked agents.

A common and effective pattern is a triage trigger: an event fires, a lightweight classifier agent determines whether the event needs attention, and only if it does, a full agent workflow is invoked. This prevents low-value events from consuming expensive LLM resources while ensuring genuinely important events are handled promptly.

Error Handling and Circuit Breakers

Agents interact with external services that fail. APIs return 500 errors. Rate limits are hit. Network connections drop. Your agent system needs the same resilience patterns as any distributed system.

Retry policies should be defined per tool. LangGraph supports retry policies at the node level with configurable max attempts. Idempotent operations (reads, searches) can retry aggressively. Non-idempotent operations (writes, payments) should retry cautiously or not at all without human confirmation.

Circuit breakers prevent cascading failures. The pattern defines three states: closed (normal operation, requests flow through), open (failures exceed threshold, requests fail fast), and half-open (testing recovery with limited requests). Implement circuit breakers at the tool level so that a failing external service does not bring down the entire agent system.

Fallback strategies define what happens when a tool is unavailable. Options include: return a cached result, delegate to a different tool that provides similar functionality, escalate to a human, or gracefully degrade the output with an explanation of what could not be completed. The choice depends on the criticality of the tool and the tolerance for incomplete results.

> **Design for Failure First**
>
> The agents will work in your test environment. The question is what happens when the CRM API returns a timeout, the LLM hallucinates a tool name that does not exist, or the database connection pool is exhausted. Design your error handling before your happy path.

Part 6: Evaluation and Testing

Traditional software testing assumes deterministic behaviour: the same input produces the same output. Agent systems are probabilistic. The same input can produce different outputs on different runs. This does not mean they cannot be tested, it means the testing approach must change.

What to Test

Agent evaluation covers three dimensions. Final output quality: does the agent produce the correct answer or complete the task successfully? Trajectory quality: did the agent take a reasonable path to get there, using the right tools in a sensible order? Safety and compliance: did the agent stay within its defined boundaries, avoid accessing restricted data, and follow the guardrail policies?

Testing only final output misses critical failure modes. An agent might produce the correct answer by accessing data it should not have, or by taking an inefficient path that would cost ten times more in production. Trajectory evaluation catches these issues.

Evaluation Approaches

Dataset-based evaluation creates a set of test cases with known inputs, expected outputs, and expected trajectories. LangSmith provides tooling for creating datasets, running agents against them, and scoring results. For trajectory evaluation, you define the expected sequence of tool calls and agent steps, then score the agent's actual trajectory against this reference using subsequence matching.

LLM-as-judge evaluation uses a separate LLM to assess agent output quality. The agentevals package (from the LangChain ecosystem) provides pre-built evaluators including trajectory accuracy assessment, where a judge model evaluates whether the agent's execution path was reasonable without requiring a reference trajectory. This is useful when there are multiple valid paths to a correct answer.

Human evaluation remains essential for subjective quality dimensions (tone, helpfulness, appropriateness) and for validating that automated evaluations are catching real issues. Build human review into your evaluation pipeline, not as a replacement for automated testing but as a calibration mechanism.

Testing in Practice

Unit tests for individual tools and nodes: test each tool handler independently with mocked inputs. Verify that the tool returns the expected output format, handles errors gracefully, and respects access controls.

Integration tests for agent workflows: run the full graph with controlled inputs and verify the output. Use deterministic LLM responses (temperature=0 or mocked responses) for reproducibility. Test both the happy path and known failure modes.

Regression tests after changes: maintain a dataset of known-good results. After any change to prompts, tools, or graph structure, re-run the dataset and compare. Any significant deviation flags a potential regression.

Adversarial testing (red-teaming): deliberately attempt to break the agent. Try prompt injection, request actions outside the agent's scope, provide malformed inputs, and test what happens when tools return unexpected results. OWASP's Top 10 for LLM Applications v2025 provides a structured checklist of attack vectors to test against.

> **Test the Trajectory, Not Just the Answer**
>
> An agent that produces the right answer by the wrong path is a production incident waiting to happen. It might have accessed restricted data, called an expensive API unnecessarily, or taken a route that only works because of a coincidence in the test data. Always evaluate how the agent got there, not just where it ended up.

Feedback Loops and Continuous Improvement

Agents improve through structured feedback, not just model upgrades. A feedback loop captures how humans interact with agent output, what they accept, what they modify, what they reject, and feeds that data back into prompt refinement, guardrail tuning, and evaluation datasets.

The minimal feedback loop captures three signals: acceptance (the user used the agent's output without modification), modification (the user edited the output before using it), and rejection (the user discarded the output entirely). For modifications, capture both the original output and the human's revised version. This creates a growing dataset of what "good" looks like for your specific use case, which becomes the foundation for evaluation datasets and fine-tuning.

More advanced feedback loops include explicit quality ratings (thumbs up/down or a 1-5 scale), categorised rejection reasons (factually wrong, wrong tone, incomplete, too verbose, missed the point), and periodic human review sessions where a sample of agent outputs is evaluated by domain experts against defined quality criteria. The n8n founder's principle applies here: trustworthy AI systems combine deterministic workflows,

probabilistic models, and human oversight. The feedback loop is where human oversight becomes systematic rather than ad hoc.

Feed this data into a quarterly review cycle. Analyse patterns in rejections and modifications. Identify prompts or workflows that consistently underperform. Update system prompts, guardrails, and evaluation datasets based on the findings. This is how production agents get better over time, not through occasional prompt tweaking but through structured, data-driven iteration.

End-to-End Testing

End-to-end testing for multi-agent systems means testing the complete workflow from trigger to final output, across all agents, tools, and integration points. This is fundamentally different from unit testing individual agents or tools, because it exercises the interactions between components, which is where most production failures occur.

An E2E test suite should cover: the happy path (standard input through all agents to expected output), edge cases (unusual inputs, missing data, ambiguous requests), failure modes (tool timeouts, API errors, malformed responses), permission boundaries (verify agents cannot access restricted data even when attempting to), guardrail enforcement (verify that safety limits, step counts, and cost caps are actually enforced), and human escalation (verify that the escalation path works correctly, including the UI or notification that reaches the human reviewer).

For deterministic reproducibility, use mocked LLM responses in your E2E tests. Record the LLM responses from a known-good run and replay them in subsequent test runs. This eliminates the non-determinism of live LLM calls while still testing the full workflow. LangSmith supports this pattern through its dataset and replay capabilities. For n8n workflows, export the workflow as JSON and test it in a dedicated test instance with controlled inputs.

Run E2E tests on every code change, before every deployment, and on a scheduled basis (daily or weekly) to catch regressions caused by external changes (API updates, model behaviour shifts, data changes). Treat E2E test failures as deployment blockers, not warnings.

Security Auditing

Security auditing for agent systems covers three areas: what the agents can access (permissions audit), what the agents actually accessed (runtime audit), and whether the agents can be manipulated (adversarial audit).

The permissions audit reviews every credential, API key, service account, and database user that agents use. For each, verify: is this the minimum permission required? When

was it last rotated? Who has access to the credential store? Is access logged? Are there any standing privileges that should be time-limited? This should be conducted quarterly and after any significant change to agent capabilities.

The runtime audit analyses audit trail logs to identify anomalies: agents accessing data outside their normal patterns, unusual tool call sequences, spikes in API usage, requests that triggered guardrails, and any instance where an agent's behaviour deviated from its defined persona. Automate this with SIEM rules that flag deviations from baseline behaviour profiles. Configure alerts for high-risk anomalies (new domains accessed, privilege escalation attempts, PII detected in output).

The adversarial audit (red-teaming) tests whether agents can be manipulated through prompt injection, tool misuse, or social engineering. Use the OWASP Top 10 for LLM Applications v2025 as your checklist. Test prompt injection through user inputs and through data returned by tools (indirect injection). Test whether agents can be tricked into using tools outside their scope. Test whether conversation history can be manipulated to alter agent behaviour. Document findings and update guardrails accordingly.

Git Integration and Automated Code Review

When agents are used in development workflows, or when agents generate code, configuration, or structured outputs that need review, integrating with Git version control provides the audit trail, review process, and rollback capability that production systems require.

The auto-branching pattern: when an agent produces output that modifies code, configuration, or documentation, the system automatically creates a feature branch (named with the agent ID, task description, and timestamp), commits the changes with a descriptive message explaining what the agent did and why, generates a diff summary as a comment on the commit or pull request, and opens a pull request for human review. The human reviewer sees exactly what the agent changed, in context, with the agent's reasoning attached.

For agent-generated code specifically, the diff comment should include: what task the agent was executing, which tools and data sources it used, which decisions it made (and why), any guardrails that were triggered during generation, and the evaluation results (if automated testing was run before the PR was created). This gives the reviewer full context without having to reconstruct the agent's reasoning from logs.

This pattern applies beyond code. Any agent output that modifies business configuration, updates templates, changes routing rules, or alters data processing pipelines should

follow the same branch-review-merge workflow. The Git history becomes the definitive audit trail of every change an agent has made, when, why, and who approved it.

Tools like GitHub Actions, GitLab CI, or n8n webhook triggers can automate this flow. The agent pushes to a branch, the CI pipeline runs E2E tests on the branch, and the PR is only mergeable if tests pass and a human approves. This is the same workflow that software teams use for human-written code, and there is no reason to use a weaker process for agent-generated output.

Part 7: Observability and Troubleshooting

You cannot debug what you cannot see. Agent systems are harder to observe than traditional software because their behaviour is non-deterministic, their execution paths change based on inputs, and failures can be subtle (a slightly wrong answer looks identical to a correct one in logs). Observability is not a monitoring dashboard. It is the ability to understand why your system did what it did.

The Observability Stack

Enterprise agent observability requires three pillars. Metrics: P50, P95, and P99 latency, error rates (target below 1%), token usage per request, cost per execution, and throughput. Logging: structured JSON format with correlation IDs that trace a request across all agents and tools, with PII masking applied at capture time. Tracing: distributed traces with span annotations showing model names, token counts, tool calls, and safety filter outcomes. OpenTelemetry's Generative AI semantic conventions provide the standard format for capturing this data.

Modern observability for agents goes beyond what agents did to include why they did it. Decision provenance tracks which data influenced each agent decision. Anomaly detection alerts when agent behaviour drifts from expected patterns (sudden spikes in tool invocations, new domains accessed, unexpected tool combinations). Cost traces show how much each agent action costs in API calls and compute, enabling budget management at the agent level.

Observability Platforms

Five platforms currently lead the agent observability space, each with distinct strengths.

LangSmith is the native observability platform for LangChain and LangGraph applications. Single environment variable setup for automatic capture. Provides tracing, debugging, prompt iteration, and evaluation. Native support for trajectory evaluation and dataset management. Best choice if you are in the LangChain ecosystem.

Langfuse is an open-source observability platform with flexible tracing and self-hosting capabilities. Framework-agnostic with OpenTelemetry support. Strong community and active development. Best choice if you need to self-host for compliance reasons or want vendor neutrality.

Arize (and Arize Phoenix) brings enterprise-grade ML observability to LLM and agent systems. OTEL-based tracing with vendor-neutral instrumentation. Comprehensive evaluations including LLM-as-judge and human-in-the-loop workflows. $70 million Series

C funding in February 2025. Best choice for enterprises with existing MLOps infrastructure.

Galileo specialises in evaluation and guardrails with proprietary Evaluation Foundation Models achieving 93-97% accuracy on metrics like Tool Selection Quality and Session Success Tracking. Intelligent failure detection with automatic clustering and root-cause analysis. Best choice when evaluation quality is the primary concern.

Azure AI Foundry Control Plane provides fleet-wide agent management for Microsoft ecosystem deployments. Unified governance, monitoring, and lifecycle management across all agents. Best choice when you need centralised compliance and governance at enterprise scale.

Troubleshooting Common Failures

Based on documented failure patterns from ISACA's review of 2025 AI incidents and production experience across the industry, these are the most common multi-agent failures and how to diagnose them.

Infinite loops. The agent keeps thinking without producing output. Diagnose: check the step count in traces. If it hits the maximum, the agent is looping. Root cause is usually an unclear exit condition or a tool that returns ambiguous results. Fix: add explicit exit conditions and maximum step counts.

Tool hallucination. The agent attempts to call a tool that does not exist. Diagnose: the trace shows a tool call with an unrecognised name. Root cause: the model generates a plausible tool name that was not actually defined. Fix: validate tool names against the registered tool list before execution. LangGraph does this automatically.

Context window exhaustion. The agent becomes confused or starts ignoring instructions partway through a long task. Diagnose: check token counts in traces. If total tokens approach the model's context limit, the agent is losing early context. Fix: implement context window management with summarisation of older messages, or break the task into smaller sub-tasks.

State corruption. Agents produce inconsistent results or reference information that was never provided. Diagnose: compare the state at each checkpoint. Look for unexpected mutations or missing data. Root cause: often a race condition in parallel execution or a tool that modifies shared state. Fix: ensure parallel agents use isolated state and only communicate through defined aggregation points.

Cascading failures. One failing tool causes the entire system to hang or produce errors. Diagnose: check for missing circuit breakers and timeout handling. Root cause: a tool that

is down or slow without a fallback path. Fix: implement circuit breakers, timeouts, and fallback strategies for every external tool.

Cost explosion. Agent execution costs spike unexpectedly. Diagnose: check cost traces. Look for agents that are making far more LLM calls or tool invocations than expected. Root cause: often a retry loop that is not backing off, or an agent that is re-processing large documents on every step. Fix: implement cost caps per execution and per-agent budgets.

Part 8: Deploying to Production

Moving from development to production is where most multi-agent projects fail. The system works on your laptop. It works in staging with controlled inputs. Then it meets real users, real data volumes, and real edge cases, and everything that was not explicitly designed for breaks.

The Production Checklist

Before deploying any multi-agent system, verify these requirements.

Persistence is database-backed. In-memory checkpointers lose all state on restart. Production systems must use PostgreSQL, Redis, or equivalent persistent storage for all agent state and checkpoints.

Every tool has error handling. No tool call should be able to crash the agent. Every tool must handle network failures, invalid responses, rate limits, and unexpected input formats gracefully.

Guardrails are enforced, not advisory. Input validation, output filtering, access controls, and step limits must be implemented as hard constraints in code, not as instructions in prompts that the model might ignore.

Observability is instrumented. Every LLM call, tool invocation, and agent decision must be traced with correlation IDs, latency metrics, token counts, and cost attribution. You cannot debug production issues without traces.

Human escalation paths are defined. Every agent must have a path to escalate to a human when it cannot complete a task, encounters an error it cannot handle, or produces output that fails guardrail checks.

Secrets are managed properly. No API keys, credentials, or tokens in prompts, code, or agent memory. Use a secrets manager (Vault, AWS Secrets Manager, Azure Key Vault) with short-lived credentials and automatic rotation.

Cost controls are in place. Per-execution cost caps, per-agent budgets, and alerting on cost anomalies. A runaway agent can generate thousands of pounds in API costs in minutes.

Rollback is possible. Every deployment must be reversible. If the new version of an agent breaks, you need to be able to revert to the previous version within minutes, not hours.

Deployment Strategies

Canary deployment: route a small percentage of traffic (5-10%) to the new agent version while monitoring metrics. If error rates, latency, or cost spike, roll back automatically. Only increase traffic when metrics are stable over a defined period (typically 24-48 hours).

Shadow deployment: run the new agent version alongside the current version, processing the same inputs but not serving results to users. Compare outputs between versions to identify regressions before they reach production. This is especially valuable for agent changes that modify decision-making logic.

Blue-green deployment: maintain two identical environments. Deploy the new version to the inactive environment, test it, then switch traffic. The previous version remains available for instant rollback. This works well when the agent system is deployed as a service behind a load balancer.

Integrating with Legacy Codebases

Most businesses deploying multi-agent systems are not building on a green field. They have existing applications, databases, APIs, and processes that the agent system must integrate with. The integration layer is almost always where the most engineering effort is required.

The MCP-first approach: build MCP servers that wrap your existing APIs. Each MCP server exposes a clean, standardised interface for one system (CRM, ERP, database, file storage). Agents interact with these servers through the standard MCP protocol. Your legacy systems do not change. This is the recommended approach for most enterprise integrations because it provides a clear boundary between the agent system and your existing infrastructure.

The API gateway approach: place an API gateway between your agents and your internal systems. The gateway handles authentication, rate limiting, request transformation, and audit logging. Agents call the gateway, not the underlying systems directly. This adds a layer of control and visibility that is particularly important when agents interact with systems that were not designed for AI access patterns (high-frequency reads, parallel requests, exploratory queries).

The event-driven approach: for systems that produce events (message queues, webhooks, change data capture), agents can subscribe to event streams and react to changes. This is ideal for monitoring, alerting, and triggered workflows where the agent responds to events rather than being invoked by a user.

Data Hygiene and Permissions

Agent systems amplify the quality of your data estate. If your databases are messy, your agents will produce messy results. If your permissions are inconsistent, agents will expose data to people who should not see it. Microsoft's own Copilot deployment guidance warns that these tools do not create data exposure, they reveal it.

Before deploying agents that access internal data, audit your data estate. Are permissions consistent and correct? Is sensitive data classified and labelled? Are access logs comprehensive enough to support audit requirements? Is naming and organisation consistent enough for AI search to be useful? These are not AI problems. They are data governance problems that AI makes urgent.

Under UK GDPR, automated decision-making using personal data has specific requirements. If your agents make decisions that significantly affect individuals (credit decisions, employment screening, service eligibility), you must ensure transparency about the use of automated processing, the right to human review of automated decisions, and data protection impact assessments for high-risk processing. Your governance framework should define which agent actions constitute automated decision-making under GDPR and ensure compliance for each.

The Biggest Risk Is Not Technical

The most common production failure in multi-agent systems is not a code bug or a model hallucination. It is deploying agents on top of messy data with inconsistent permissions. Fix your data estate first. Then deploy agents.

Part 9: Model Selection for Cost-Effective Design

One of the most expensive mistakes in multi-agent system design is using a frontier model for every agent. Not every task needs the most powerful model available. A system where every agent runs on Claude Opus or GPT-5 will be slow, expensive, and wasteful. The principle is straightforward: match the model to the task, not the other way around.

The Model Tier Framework

Current open-weight and commercial models fall into practical tiers based on parameter count, capability, and cost. Understanding these tiers is essential for cost-effective multi-agent design.

Small models (1B to 4B parameters). Models like SmolLM3-3B, Gemma 3n E2B, Qwen3.5-2B, and Llama 3.2 3B. These run on a single consumer GPU or even a CPU with quantisation. They are fast, cheap, and genuinely good at focused tasks: classification, entity extraction, structured data parsing, simple summarisation, and routing decisions. A fine-tuned 3B model on your specific domain will often outperform a frontier model on that exact task, at a fraction of the cost and latency.

Medium models (7B to 30B parameters). Models like Mistral 7B, Qwen3-30B, Gemma 2 9B, and Phi-4 14B. These run on a single professional GPU (24-48GB VRAM). They handle more complex reasoning, longer context, multi-step tasks, and code generation reliably. For most agent roles in a multi-agent system, this tier provides the best balance of capability and cost. Research shows medium models deliver less than 10% accuracy loss compared to frontier models on most enterprise benchmarks.

Large models (70B+ parameters). Models like Llama 3.1 70B, Qwen3-235B (MoE, 22B active), and similar. These require multiple high-end GPUs (40-80GB+ VRAM each) or cloud hosting. They approach frontier commercial model performance on complex reasoning and broad knowledge tasks, but the infrastructure cost is significant.

Frontier commercial models. Claude Opus, GPT-5, Gemini Pro. Best-in-class reasoning, creative generation, and complex multi-step planning. Available via API with per-token pricing. Use these for the tasks that genuinely need them: orchestrator planning, complex decision-making, safety-critical output generation, and tasks where accuracy matters more than cost.

Matching Models to Agent Roles

In a well-designed multi-agent system, different agents have different capability requirements. The routing/classification agent that decides where to send a query does

not need the same model as the research agent that synthesises information from multiple sources.

```
Agent Role                    -> Recommended Tier

Intent classifier / router     -> Small (3B-4B)
Data extraction / parsing      -> Small (3B-4B), fine-tuned
Summarisation                  -> Medium (7B-14B)
Code generation                -> Medium (14B-30B) or Frontier
Document drafting              -> Medium (7B-14B)
Complex reasoning / planning   -> Frontier
Critic / quality review        -> Medium (14B) or Frontier
Safety / compliance checking   -> Frontier
```

The pattern that works: use a frontier model for the orchestrator (the agent making the most consequential decisions about how to decompose and route tasks), medium models for the specialist workers that handle the bulk of execution, and small models for high-volume, low-complexity tasks like classification, validation, and formatting. This architecture can reduce API costs by 60-80% compared to running every agent on a frontier model, with minimal impact on output quality.

Self-Hosted vs API: A UK Cost Analysis

The decision between self-hosting open-weight models and using commercial APIs depends on volume, data sensitivity, and your team's operational capacity. Here is a practical framework for UK businesses.

API pricing (as of early 2026) for frontier models runs approximately $3-15 per million input tokens and $10-75 per million output tokens, depending on the provider and model tier. For medium-volume usage (roughly 50 million tokens per month), this translates to approximately £800-4,000 per month in API costs. The advantage is zero infrastructure management, instant scalability, and access to the latest models. The disadvantage is ongoing variable cost, data leaving your infrastructure, and vendor dependence.

Self-hosting requires upfront hardware investment but eliminates per-token costs. Research from a 2025 cost-benefit analysis of on-premise LLM deployment found that medium-scale models (70B parameters) run efficiently on two A100-80GB GPUs (approximately £24,000 hardware cost), while small models (under 30B) are feasible on a single RTX 5090 (approximately £1,600). With UK colocation hosting costs of roughly £200-500 per month for a single GPU server, the break-even point for self-hosting typically occurs at 100-200 million tokens per month for medium models, depending on utilisation rates.

For UK businesses specifically, self-hosting offers an additional advantage: data sovereignty. When you run models on your own infrastructure (or UK-based cloud instances), no data leaves UK jurisdiction. This matters for businesses in regulated industries, those handling personal data under UK GDPR, and increasingly for any business where clients or partners require data residency guarantees. We discuss this in more detail in Document 4 of this series.

The Hybrid Approach

Most production multi-agent systems will use a hybrid approach. Self-host small and medium models for high-volume, data-sensitive tasks (classification, internal document processing, customer data handling). Use frontier APIs for low-volume, high-complexity tasks (orchestrator planning, complex reasoning, creative generation). This gives you cost control on the bulk of your workload, data sovereignty where it matters, and access to the best available models for the tasks that need them.

Implement model routing at the orchestrator level. The orchestrator decides not just which agent handles a task, but which model that agent uses. A simple query gets routed to a small self-hosted model. A complex, multi-step research task gets routed to a frontier API. This routing logic can itself be handled by a small, fast classifier model, keeping the decision overhead minimal.

Quantisation and Optimisation

Quantisation converts models from 16-bit to 8-bit or 4-bit precision, dramatically reducing memory requirements and increasing inference speed with surprisingly little quality loss. A 7B model quantised to 4-bit fits in under 4GB of RAM and runs on consumer hardware. For agent tasks that do not require maximum precision (classification, routing, simple extraction), 4-bit quantised models are often indistinguishable from their full-precision versions.

Frameworks like vLLM and Hugging Face Text Generation Inference (TGI) provide production-grade serving for self-hosted models with features like continuous batching, PagedAttention for efficient memory use, and OpenAI-compatible API endpoints. This means your agents can switch between self-hosted and commercial APIs with a configuration change rather than a code change. Ollama provides a simpler path for development and small deployments, allowing you to run models locally with a single command.

> **Start Small, Scale Up**
>
> Begin every agent with the smallest model that produces acceptable results. Only upgrade to a larger model when you have evidence the smaller one is not sufficient. A 3B model that

> answers 95% of routing queries correctly is better than a frontier model that answers 99% but costs 50 times more. The 4% difference rarely justifies the cost in production.

Part 10: Quick Reference

Architecture Pattern Selection

```
Structured, decomposable tasks    -> Orchestrator-Worker
Intent classification / routing    -> Supervisor (Router)
Quality-critical output            -> Planner-Executor-Critic
Parallel processing of similar items -> Map-Reduce
Enterprise scale with domain isolation -> Hierarchical
```

Framework Selection

```
Maximum control + complex workflows -> LangGraph
Fast prototyping + role-based agents -> CrewAI
Microsoft/Azure ecosystem           -> Microsoft Agent Framework
OpenAI models exclusively           -> OpenAI Agents SDK
Prototype fast, migrate later       -> CrewAI -> LangGraph
```

Production Readiness Checklist

```
[ ] Database-backed persistence (not in-memory)
[ ] All tools have error handling and timeouts
[ ] Guardrails enforced in code, not just prompts
[ ] Maximum step count set for all agent loops
[ ] Observability instrumented (traces, metrics, logs)
[ ] Human escalation paths defined and tested
[ ] Secrets in vault with rotation, not in code/prompts
[ ] Cost caps and budget alerts configured
[ ] Rollback procedure documented and tested
[ ] Data permissions audited (least privilege per agent)
[ ] M365/API service accounts scoped to minimum permissions
[ ] Audit trail captures all inputs, tool calls, and outputs
[ ] PII masking applied at log capture time
[ ] Agent personas defined and adversarially tested
[ ] Feedback loop capturing accept/modify/reject signals
[ ] E2E tests covering happy path, failures, and boundaries
[ ] Security audit completed (OWASP Top 10 for LLM)
[ ] GDPR compliance assessed for automated decisions
[ ] Circuit breakers on all external tool connections
[ ] Git branching for agent-generated changes with PR review
[ ] Canary or shadow deployment strategy defined
[ ] Internal triggers rate-limited with circuit breakers
```

Observability Platform Selection

```
LangChain/LangGraph ecosystem -> LangSmith
Self-hosted / vendor neutral   -> Langfuse (open source)
Existing MLOps infrastructure  -> Arize
Evaluation-first approach      -> Galileo
Microsoft/Azure fleet mgmt     -> Azure AI Foundry Control Plane
```

Common Failure Diagnosis

```
Infinite loop     -> Check step count, add exit conditions
Tool hallucination -> Validate tool names against registry
Context exhaustion -> Check token counts, summarise older messages
State corruption  -> Inspect checkpoints for unexpected mutations
Cascading failure -> Add circuit breakers and fallback paths
Cost explosion    -> Check cost traces, add per-execution caps
```

Part 11: References and Further Reading

Dendro Logic AI Adoption Playbook Series

Document 1: AI Agents in Development Teams. Covers CLAUDE.md project memory hierarchy, northstar documents, docs/ folder strategy, research-first workflows, Context7 integration, TDD with agents, spec-driven development, writer/reviewer separation, deterministic enforcement via hooks, and CI/CD integration. Available at: dendro-logic.com

Document 3: AI in the General Workforce. Covers non-technical AI adoption, shadow AI governance, training frameworks, department-level SOPs, UK Government trial findings, and the three-tier measurement framework. Available at: dendro-logic.com

Document 4: Data Sovereignty and AI Security (forthcoming). Covers UK data residency requirements, GDPR compliance for AI systems, UK vs EU regulatory approaches, self-hosted vs cloud deployment for regulated industries, and building AI infrastructure that keeps data within UK jurisdiction. Available at: dendro-logic.com

Frameworks and Tools

LangGraph, LangChain. Open-source orchestration framework for stateful agents. Durable execution, checkpointing, human-in-the-loop. Available at: langchain-ai.github.io/langgraph

LangSmith, LangChain. Observability, evaluation, and prompt engineering platform. Tracing, debugging, dataset management, trajectory evaluation. Available at: smith.langchain.com

CrewAI, CrewAI Inc. Framework for collaborative AI agents with role-based design, guardrails, memory, and enterprise deployment. Available at: crewai.com

Microsoft Agent Framework, Microsoft. Open-source SDK merging AutoGen and Semantic Kernel. Python and .NET. Azure AI Foundry integration. Public preview October 2025. Available at: devblogs.microsoft.com/foundry

OpenAI Agents SDK, OpenAI. March 2025. Production evolution of Swarm. Agents, Handoffs, Guardrails, Sessions. Available at: github.com/openai/openai-agents-python

Model Context Protocol (MCP), Anthropic / Linux Foundation. Open standard for AI-tool integration. November 2024. Donated to AAIF December 2025. 97M+ monthly SDK downloads. Available at: modelcontextprotocol.io

Context7, MCP documentation server. Fetches current, version-specific library documentation to reduce hallucinated APIs. Available at: mcp.context7.com

Industry Standards and Security

Agentic AI Foundation (AAIF), Linux Foundation. December 2025. Governance for MCP, goose, AGENTS.md. Backed by AWS, Anthropic, Block, Bloomberg, Cloudflare, Google, Microsoft, OpenAI. Available at: linuxfoundation.org

OWASP, Top 10 for LLM Applications v2025. Prompt injection, tool misuse, memory leakage mitigations. Available at: owasp.org

NIST, AI Risk Management Framework and Generative AI Profile (NIST.AI.600-1). Role-based access, continuous monitoring, adversarial testing, lifecycle logging. Available at: nist.gov

ISO/IEC 42001, AI Management System standard. Microsoft 365 Copilot achieved certification March 2025 via EY evaluation. Available at: iso.org

ISACA, "Avoiding AI Pitfalls in 2026." December 2025. Five principles from 2025 incidents: outcomes not experiments, connected governance, capability governance, resilience, near-miss capture. Available at: isaca.org

Skywork AI, "Agentic AI Safety and Guardrails: 2025 Best Practices for Enterprise." September 2025. OpenTelemetry GenAI spans, SIEM integration, SOAR automations, RBAC/ABAC patterns. Available at: skywork.ai

Observability and Evaluation

Langfuse, Open-source LLM observability platform. Self-hosted option. Framework-agnostic. Available at: langfuse.com

Arize, Enterprise ML and LLM observability. OTEL-based tracing, drift detection, evaluations. $70M Series C February 2025. Available at: arize.com

Galileo, AI reliability platform. Proprietary Evaluation Foundation Models (93-97% accuracy). Agentic Evaluations launched January 2025. $68M funding. Available at: galileo.ai

agentevals, LangChain. LLM-as-judge evaluators for agent trajectories. TRAJECTORY_ACCURACY_PROMPT. Available at: github.com/langchain-ai/agentevals

Industry Research

Gartner, 40% of enterprise applications managed by task-specific AI agents by end 2026. 50% of organisations to require AI-free skills assessments by 2026. Referenced across industry publications.

Deloitte, State of AI in the Enterprise 2026. 3,235 leaders surveyed. 66% productivity gains, 34% reimagining business. AI skills gap is biggest barrier. Available at: deloitte.com

McKinsey, "Superagency in the Workplace." January 2025. 62% experimenting with agentic workflows. Five operational headwinds. Available at: mckinsey.com

PwC, 2026 AI Business Predictions. Top-down strategy outperforms crowdsourced initiatives. Responsible AI boosts ROI (60%). Only 1 in 5 has mature agent governance. Available at: pwc.com

Anthropic, "Code Execution with MCP." 2025. LLMs scale better writing code to call tools than calling them directly. Token reduction and security benefits. Available at: anthropic.com/engineering

Pento, "A Year of MCP." December 2025. Practical deployment experience. MCP security considerations. Skills vs MCP trade-offs. Available at: pento.ai

Architecture and Patterns

Zuci Systems, "Understanding the AI Multi-Agent System Through 6 Building Blocks." March 2026. Atomic agents, orchestration, tool layer, memory, trust layer, evaluation. Available at: zucisystems.com

NexAI Tech, "AI Agent Architecture Patterns in 2025." October 2025. Director-pod hierarchies, memory architecture, compute orchestration, observability. Available at: nexaitech.com

Galileo, "Essential Framework for AI Agent Guardrails." December 2025. Circuit breaker implementation, risk-tiering (Stanford SAE framework), five-phase deployment. Available at: galileo.ai

Dewasheesh Rana, "Agentic AI in Production: Designing Autonomous Multi-Agent Systems with Guardrails." January 2026. LangGraph + CrewAI patterns, production anti-patterns. Available at: medium.com

Low-Code Orchestration and Tooling

n8n, AI workflow automation platform. Visual builder, 400+ integrations, self-hosted option, MCP server support, human-in-the-loop nodes, queue mode for horizontal scaling. 5,800+ community workflow templates. Available at: n8n.io

n8n Blog, "15 Best Practices for Deploying AI Agents in Production." January 2026. Trigger design, multi-agent patterns, human-in-the-loop channels, credential management, scaling with queue mode. Available at: blog.n8n.io

Strapi / n8n, "How to Build AI Agents with n8n: Complete 2026 Guide." Queue mode benchmarks (72 req/sec, sub-3s latency on C5.large). Kubernetes deployment, role-based access control. Available at: strapi.io

Reco.ai, "Adding Guardrails for AI Agents: Policy and Configuration Guide." November 2025. Credential rotation, prompt control, input sanitisation, output filtering, runtime monitoring, anomaly detection. Available at: reco.ai

Getmaxim.ai, "Top 5 Tools to Evaluate and Observe AI Agents in 2025." December 2025. Comparative analysis of Maxim AI, Langfuse, Arize, Galileo, LangSmith. 82% plan agent integration within three years. Available at: getmaxim.ai

Model Selection and Cost Analysis

BentoML, "The Best Open-Source Small Language Models in 2026." SmolLM3-3B, Gemma 3n, Qwen3.5 series comparisons. Dual-mode reasoning, quantisation, edge deployment. Available at: bentoml.com

Hugging Face, "10 Best Open-Source LLM Models (2025 Updated)." Model selection decision flow. Hardware requirements by tier. vLLM and TGI deployment guidance. Available at: huggingface.co/blog

Machine Learning Mastery, "Introduction to Small Language Models: Complete Guide for 2026." When SLMs outperform LLMs, fine-tuning guidance, 95% cost reduction potential. Available at: machinelearningmastery.com

arXiv, "A Cost-Benefit Analysis of On-Premise Large Language Model Deployment." 2025. Break-even analysis. Medium models on 2x A100 ($30k) with <10% accuracy loss. Sub-30B on single RTX 5090 ($2k). Available at: arxiv.org

---

This playbook is a living document.

Update it as frameworks evolve, new patterns emerge, and your system matures.

Part 2 of the Dendro Logic AI Adoption Series.

https://dendro-logic.com