Enterprise Agentic Governance: A Five-Pillar Framework

How The Agile Monkeys approaches agent integration in medium-to-large organizations

The Agile Monkeys

March 2026

The Landscape

AI agents are entering the enterprise whether organizations are ready or not. The question is no longer whether to adopt them, but how to avoid the failures that are already happening at scale.

The data is sobering. In Cleanlab's survey of 1,837 practitioners, only 95 — about 5.2% — reported having AI agents live in production. Among regulated enterprises in that group, 70% rebuild their agent stack every three months or faster. Gartner predicts that over 40% of agentic AI projects will be canceled by the end of 2027 due to escalating costs, unclear business value, or inadequate risk controls.

Meanwhile, an AIUC-1 Consortium industry briefing reported by Help Net Security (2026) found that 80% of organizations reported risky agent behaviors — including unauthorized system access and improper data exposure — while only 21% of executives reported complete visibility into agent permissions, tool usage, or data access patterns. Separately, Silverfort reports that non-human identities outnumber human identities 50:1 in the average enterprise — a relevant signal for agent governance, even though that figure covers all machine identities (service accounts, API keys, cloud roles), not AI agents specifically.

The BYOA (Bring Your Own Agent) trend is adding urgency. Microsoft uses the term directly and has positioned Agent 365 as a control plane for governing agents built both inside and outside its stack. The core risk: shadow IT accesses data; shadow AI takes actions. An unsanctioned agent with access to corporate email doesn't just read messages — it can send them, forward them, and trigger downstream workflows.

The failure mode isn't agents themselves. It's ungoverned agents.

Why Agent Projects Fail

Three patterns emerge:

Integration, not models, is often the bottleneck. Vendor analyses such as Composio (2025) argue that brittle connectors, weak retrieval architectures, and non-event-driven designs break many agent pilots before model quality becomes the dominant issue. This is a widely observed practitioner pattern, though independent research quantifying it remains limited.

Multi-agent doesn't always help. Kim et al. (Google Research, Google DeepMind, and MIT, 2025) demonstrated empirically that multi-agent architectures degrade sequential reasoning by 39-70%, with communication

overhead scaling super-linearly (exponent 1.724). On tool-heavy tasks, multi-agent incurs a 2-6x efficiency penalty versus single-agent approaches. Not every problem benefits from more agents.

Governance remains immature. Only 19% of businesses are deploying agentic AI at scale (IBM, 2026). When organizations do deploy governance, they discover that existing frameworks lack agent-specific controls — tool-call validation, prompt-injection logging, and containment testing are not covered by traditional IAM, DLP, or compliance tooling.

The Maturity Model

Most organizations are still early:

Level	Description	Where most orgs are
0	No AI agents	Declining
1	Copilots and chat assistants	Most are here
2	Single-purpose task agents	Early adopters
3	Coordinated agent workflows	Pilots
4	Governed multi-agent systems	Almost nobody

Our framework targets Level 4, but each pillar adds value independently. You don't need to build the full architecture to benefit from any single piece.

Design Principles

Three principles guide every architectural decision in this framework. Critically, none of them require new governance infrastructure — they extend tools organizations already use to manage people and services (IAM, network isolation, event streaming, relationship-based permissions) to cover agents.

Human sovereignty over agent behavior. Every employee controls what their agents can see and do. Autonomous agents operate within boundaries designed and auditable by humans. The goal is not human involvement in every action — that doesn't scale — but human authority over the rules governing agent behavior. CSA's Agentic Trust Framework embodies this principle: "Trust must be earned through demonstrated behavior and continuously verified through monitoring." Agents progress through increasing levels of autonomy, with clear criteria and controls at each stage.

Least privilege by design. Agents can only access the data they need for their specific task, and nothing more. This is enforced architecturally — through credential delegation, policy engines, and network isolation — not through prompting or behavioral guidelines. As UC Berkeley's governance framework emphasizes, agents should be treated as untrusted entities given current evaluation limitations.

Transparency at every level. Every data access and every action is logged, auditable, and visible. For personal agents, the employee sees everything their agents do. For autonomous agents, the organization sees everything. Opacity is the enemy of governance.

The Five Pillars

Pillar 1: Personal Agent Layer

Every employee gets a personal set of agents managed through a control-plane dashboard that shows which agents they have, every data access, and every action taken. Employees define agent profiles, manage tool access, and set boundaries. The dashboard is a transparency tool for the employee, not just an IT governance tool.

The critical architectural constraint: agents inherit the employee's own access level — never more. The technical mechanism is OAuth 2.0 token exchange (RFC 8693) with scope restriction, where the employee's credentials serve as a ceiling that agent permissions cannot exceed. MIT's formal framework for authenticated delegation provides the academic foundation (MIT Media Lab, 2025); Oso's delegated access model provides the implementation pattern. Microsoft is building exactly this with Agent 365 — per-agent Entra identity, employee self-service agent creation, and a centralized control plane (GA May 2026).

In practice, most personal agent tooling connects to third-party services through MCP (Model Context Protocol) servers. MCP is the emerging standard for agent-to-tool integration, and most MCP servers use OAuth as their authentication mechanism — when an agent connects to Jira, GitHub, or Linear through MCP, the underlying OAuth delegation is what enforces the employee's permission ceiling. MCP also enables per-tool permission control: organizations can activate or deactivate specific operations (e.g., allow reading issues but not closing them), and require human confirmation before executing potentially destructive actions like deletions or bulk updates. Tools like Claude Desktop already implement this personal tool management when connecting MCP servers.

For autonomous harnesses — tools like OpenClaw or Hermes that run agents as unattended background processes — this per-tool governance becomes especially important. A personal agent control panel where employees manage all their agents' profiles, tool access, and permissions in a single place is the natural interface. This is the dashboard described at the top of this section: not just visibility into what agents are doing, but active control over what they're allowed to do.

A critical gap: OAuth 2.0 delegation tokens do not automatically restrict scope. Without additional policy enforcement — OPA, Just-in-Time access controls — agents can inadvertently escalate privileges beyond the user's intent (CyberArk, 2025). The identity management problem at scale (thousands of agent identities per enterprise) requires agent-native IAM, not traditional IAM extended.

For model execution, cloud-based models are perfectly viable for personal agents as long as company policies allow it — and most do. Local open-source models make sense in two scenarios: to reduce costs on high-frequency, low-complexity tasks (no per-token cost), or when privacy requirements or organizational policy prohibit sending data to external providers. Most production setups use cloud models as the default and route locally only when cost or policy demands it.

Scenario: An engineer connects their personal agent to GitHub, Jira, and Slack through MCP servers. Through the control panel, they grant the agent read access to all three but restrict write access: the agent can create GitHub PRs and comment on Jira tickets, but closing issues or deleting anything requires human confirmation. The agent inherits the engineer's own permissions in each service — it cannot access repositories or projects the engineer doesn't have access to. When the engineer leaves for lunch, the agent continues triaging notifications and drafting PR reviews in the background, but queues any destructive actions for approval.

We explore personal agent architectures, credential delegation models, and on-device/cloud tradeoffs in detail in "Personal Agent Architecture & Credential Delegation."

Pillar 2: Autonomous Agent Services

Autonomous agents — those that run as cloud services without direct human operation — must be treated with the same infrastructure rigor as any production microservice. Dedicated VPCs, API gateways, short-lived credentials, mutual TLS, comprehensive audit logging. If your organization runs microservices securely, much of agent governance is already familiar.

What is genuinely new comes from three challenges with no microservices equivalent:

Semantic data transformation. Agents summarize, paraphrase, and translate data — creating information leaks invisible to pattern-based DLP. When an agent distills a confidential report into a summary and sends it to an unauthorized recipient, regex-based data loss prevention won't catch it. Semantic DLP — systems that analyze meaning, not patterns — is required. Microsoft Purview now extends DLP policies to agents, representing the most enterprise-ready approach available today.

Structured output enforcement. While raw LLM output is nondeterministic, modern agent SDKs provide multiple layers to enforce deterministic behavior where it matters. Agents can be constrained to produce validated JSON outputs conforming to strict schemas, and deterministic pre- and post-execution hooks can validate business logic before any action is taken — retrying automatically when validation fails. The pattern is separation of concerns: the LLM generates an intent (e.g., "process a $500 refund for customer #1234"), structured output enforcement guarantees the intent conforms to the expected schema, and a deterministic policy validator confirms the action is within policy before execution. This layered validation — schema enforcement, business logic hooks, policy gates — is what makes SOC 2 compliance achievable for agent systems.

Scoped tool access for sub-agents. When an agent delegates tasks to sub-agents, each sub-agent should be provisioned with only the specific tools it needs — not a subset of the parent's full permissions dynamically delegated at runtime. In practice, production agent systems predefine which tools each sub-agent can access at design time: a payment processing sub-agent gets access to the payment API and nothing else; a fraud-checking sub-agent gets read access to transaction history but no write capabilities. This is the same principle as microservice architecture — each service has its own credentials scoped to its function. The governance challenge is ensuring this scoping is consistent and auditable, which is where NIST's AI Agent Standards Initiative is working to establish identity and authentication standards for autonomous agents.

A critical identity principle: agents should be first-class principals in the organizational identity system, not second-class entities managed through a separate registry. Cloud infrastructure has already validated this — Google Cloud IAM, AWS IRSA, and the CNCF's SPIFFE standard all treat non-human workloads as principals in the same policy system as human users. The principle is agents-as-principals, not agents-as-users: same policy engine, same audit trails, same governance tools — without implying legal equivalence to human users. Agents don't have privacy rights or personal accountability; they do have identities, permissions, and

auditable behavior.

How to implement access control — whether through RBAC, PBAC, task-scoped JIT permissions, or some combination — is an engineering decision that depends on the use case. What matters is that agents, like any other service, are treated as principals in the organization's existing permission enforcement infrastructure (Obsidian Security, 2025). The same tools teams already use to manage microservice permissions apply to agents.

Scenario: A customer service agent processes a refund request. It accesses the customer record via task-scoped JIT permission, generates a refund intent, which passes through the deterministic policy validator — refund amount within limits, customer identity verified, no fraud flags. The agent cannot access other customer records or escalate its own permissions.

We explore agent governance architecture, agents-as-principals identity models, compliance mapping (HIPAA, PCI, SOC 2, GDPR, ISO 42001), and agent testing frameworks in detail in "Agent Governance & Compliance for the Enterprise."

Pillar 3: Enterprise Knowledge Graph

Flat knowledge bases — single vector databases containing all organizational documents — break at enterprise scale. LLMs struggle with multi-hop reasoning and enterprise-specific language without structured scaffolding (Glean, 2025). Flat retrieval quality degrades as corpus size grows; structured approaches achieve logarithmic search complexity (LATTICE, 2025). And critically, different roles, agents, and organizational levels need fundamentally different views of the same knowledge.

We propose an enterprise knowledge graph — a network of purpose-typed knowledge nodes connected by organizational, topical, and access edges. Each node is a knowledge container: a project status summary, a risk assessment, an architecture decision record, a team retrospective. Edges encode relationships: "belongs-to" (organizational hierarchy), "summarizes" (abstraction levels), "relates-to" (cross-cutting topics), "consumes" (agent access patterns).

The organizational hierarchy — personal, team, department, organization — emerges naturally from "belongs-to" and "summarizes" edges, mirroring the real org structure. An engineer sees their project knowledge in full detail and their department's themes in summary. A VP sees department-level detail and organizational themes. But this hierarchy is one view over a richer structure, not the architecture itself. Compliance

policies connect to every department through "applies-to" edges. A technology decision in engineering relates to product and marketing through "affects" edges. A security vulnerability cuts across every team. None of these fit a strict hierarchy — in a graph, they're first-class connections.

The key governance insight: every node has its own access configuration. Permissions flow through graph edges via Relationship-Based Access Control (ReBAC) — a manager sees their team's knowledge because they manage the team, not because someone assigned them a "manager" role. Within each node, Attribute-Based Access Control (ABAC) adds purpose-based segmentation: a department-level node might contain risk assessments, hiring decisions, and architecture records, each with different access requirements. The enforcement point matters: permissions should be applied as early as possible in the retrieval pipeline — ideally as pre-retrieval filters so the search engine evaluates fewer candidates, improving both security and performance. When pre-filtering isn't feasible (e.g., permissions depend on content that only exists post-retrieval), post-retrieval filtering before the LLM sees anything is the fallback (Permit.io, 2025). This isn't new governance infrastructure. It's the same relationship model organizations already use to manage people, applied to knowledge.

Provenance is non-negotiable. When knowledge nodes contain derived artifacts — summaries, aggregations, extracted themes — they must maintain structured references back to the original source content. LLMs make mistakes, and if a summary cannot be traced back to the documents it was derived from, errors become impossible to detect and correct. Every derived node in the graph should carry provenance metadata: what sources it was built from, when, and by which process. This makes the knowledge base self-correcting — when a source document is updated or found to be inaccurate, all derived artifacts that depend on it can be identified and regenerated.

GraphRAG demonstrates the quantitative case: 72-83% comprehensiveness win rate over flat RAG for global sensemaking queries (Microsoft Research, 2024). The production knowledge systems we examined — Glean's multi-level graphs, GraphRAG's community detection, Salesforce's semantic layer — all use graphs internally.

The honest tradeoff: graph-based architectures cost 2-3x more than flat RAG due to relationship modeling, multi-level summarization, and permission evaluation at query time (NStarX, 2025). For organizations under 1,000 employees with uniform access levels, flat RAG with good chunking may be entirely adequate. Start flat, monitor where it fails — queries that return irrelevant results, access control requests that can't be expressed, cross-cutting knowledge that needs to live in multiple

places — and build graph structure where the failures point.

Scenario: A VP asks their agent "what are our biggest technical risks this quarter?" The agent traverses the knowledge graph from the VP's department node to risk-assessment nodes across teams, following "summarizes" edges to the right abstraction level and filtering by the VP's ReBAC permissions. The VP sees aggregated risk themes without accessing individual project details they're not authorized for. Drilling into a specific risk follows graph edges down to the source nodes.

We explore knowledge graph architectures, five implementation patterns, purpose-segmented access control, and graph-based retrieval in detail in "Hierarchical Knowledge Architectures for Enterprise Agents."

Pillar 4: Purpose-Built Data Pipelines

Agents have limited context windows. Flooding them with raw, unprocessed data causes four documented failure modes: context poisoning (hallucinations entering persistent context), context distraction (excessive irrelevant information), context confusion (superfluous details obscuring what matters), and context clash (contradictory information from different sources) (Breunig, via LangChain 2025). RAG failures are, as an AWS Builders Community analysis puts it, "a data engineering problem disguised as AI" (AWS Builders, 2025) — caused by stale data, schema drift, and absent data contracts, not by model limitations.

The theoretical frame comes from Andrej Karpathy (2025): if LLMs are CPUs, context windows are RAM. Context engineering — filling that RAM with exactly the right information for each step — is the discipline that makes agents effective. Purpose-built data pipelines are context engineering infrastructure.

Our proposed architectural pattern is to treat incoming data as an event source, combined with CQRS (Command Query Responsibility Segregation). Raw data sources — Slack messages, email threads, ticket updates, meeting notes — are captured as immutable event streams with full metadata, including structured provenance identifiers that trace each event back to its original source (a specific Slack message, a particular Jira field change, a calendar event). Purpose-built read models are materialized from these events for specific agent consumption patterns: topic extraction models that cluster messages across channels, project status models that aggregate deadlines and blockers, decision log models that identify decisions and their downstream actions, and risk models that detect urgency signals. Each read model carries correlation identifiers linking it to the events that produced it, enabling full traceability from any

derived artifact back to its source facts.

The key distinction is between facts and knowledge. Raw events are facts — immutable, timestamped records of what happened. Read models are knowledge — structured, purpose-built views that can be rebuilt when context changes. The power of treating incoming data as events is that you can attach different processors to the same event stream to generate different purpose-specific artifacts or trigger different actions. The same stream of Slack messages can feed a topic extractor, a decision logger, and an urgency detector simultaneously — each producing a different knowledge artifact optimized for a different agent consumption pattern. When a project's scope shifts, historical events can be reprocessed through updated logic to generate new projections. The purpose of each pipeline — the reason you're processing the data — determines what knowledge it extracts and how that knowledge is structured as nodes in the enterprise knowledge graph (Pillar 3). Knowledge builders don't just process data; they build the graph.

In practice, different data sources require different processing expertise. A Slack knowledge builder understands threading, reactions, and conversational patterns. A Jira knowledge builder understands state transitions, field relationships, and linking semantics. Source-specialist knowledge builders — dedicated processors per data source — produce higher-quality read models than generic pipelines because they encode domain-specific knowledge about what matters in each source. Their outputs can compose: team-level builders consume source-level artifacts to produce team summaries, department-level builders consume team artifacts, and so on up the hierarchy — with mandatory quality gates and data lineage at every stage to prevent cascading errors.

Scenario: Instead of dumping all Slack messages into a vector database, a Slack knowledge builder captures messages as an immutable event stream and materializes purpose-specific read models: a topic extractor clusters conversations by subject across channels, a project status view aggregates deadlines and blockers, and a decision log identifies key decisions with their context. A team-level builder consumes these artifacts alongside Jira and Calendar builders' outputs to produce a unified project status view. An agent querying "what are the open blockers for Project Alpha?" retrieves structured, current, purpose-built data — not a similarity search across thousands of raw messages.

We explore event sourcing patterns, CQRS for agent data, source-specialist knowledge builders, composable knowledge flows, and purpose-specific tool design in detail in "Context Engineering: Purpose-Built Data Pipelines for Agents."

Pillar 5: Agent Orchestration & Human Oversight

Designing how agents coordinate, delegate, and interact with humans is a core engineering and product design challenge — one that requires deliberate context engineering, not generic patterns applied as an afterthought.

Communication protocols. The agent communication protocol landscape is consolidating around four complementary layers: MCP (Model Context Protocol, for agent-to-tool integration), A2A (Agent-to-Agent Protocol, for peer coordination), ACP (Agent Communication Protocol, for edge/local), and ANP (Agent Network Protocol, for internet-scale discovery). These are complementary, not competing — MCP handles the vertical axis (tools), while A2A and ACP handle the horizontal axis (agent-to-agent). This landscape is evolving rapidly; specific protocols may consolidate by the time you read this.

Orchestration patterns. Anthropic's agent framework (2025) defines several well-tested coordination patterns that inform governance decisions:

- Orchestrator-worker: A lead agent decomposes goals and delegates to specialist subagents, each running in an isolated context with only the tools it needs. The orchestrator manages coordination; workers focus on execution.
- Sequential pipelines with deterministic gates: Steps execute in a predefined order, with programmatic validation between each step — schema checks, business logic assertions, test suites. Each gate is a governance checkpoint.
- Parallelization: Independent subtasks are distributed across multiple subagents running simultaneously. This is where multi-agent genuinely helps — not for serial reasoning, but for embarrassingly parallel work.
- Routing: Inputs are dynamically directed to specialized models or workflows based on content classification. This is a product design decision: what gets handled automatically, what gets escalated.
- Evaluator-optimizer: One agent generates output while another evaluates and provides feedback in a loop, with iterative refinement until quality criteria are met.

The key governance question is when to use a single orchestrator agent with subagents versus treating agents as separate services communicating through protocols like A2A. A single orchestrator is appropriate when all agents share a common goal and need tight coordination — an agent processing a customer request that delegates to specialized workers. Separate agent services make sense when agents belong to different teams, have independent lifecycles, or need to communicate across organizational

boundaries. The choice has direct implications for identity management (Pillar 2), data access (Pillar 3), and auditability.

The event-driven gap. Current agent protocols are built on HTTP and JSON-RPC — request/response patterns designed for synchronous, point-to-point interactions. But enterprise systems run on event-driven architectures where services communicate through event streams (Kafka, Pulsar), not API calls. This mismatch is being actively addressed: Confluent has demonstrated MCP and A2A messages transported over Kafka topics (Waehner, 2025), and has introduced streaming agents on Apache Flink that combine data processing with AI reasoning natively. A2A's community is drafting an "Inbox Pattern" for broker-based transport with topic-based inboxes and correlation IDs. ACP (IBM/Linux Foundation) was designed async-first, though it remains REST-based. Confluent has also documented four event-driven multi-agent patterns — orchestrator-worker, hierarchical agent, blackboard, and market-based — adapted as distributed streaming systems.

What's still missing is the full lifecycle: agent protocols describe how agents exchange messages, but not how agents operate as event consumers. In an event-driven enterprise, agents should work like specialized microservices — reading batches of events from relevant topics, building purpose-specific read models (connecting to Pillar 4's knowledge builders), acting on those models, and writing results back to the event stream. This "agent-as-event-processor" pattern is the natural convergence of event-driven architecture and agentic AI, and no current protocol fully specifies it. Organizations building event-driven agent systems today are designing this layer themselves.

Human oversight as product design. How agents interact with humans is not a one-size-fits-all governance layer — it's a product design decision that varies by use case. An agent answering questions in a chat interface has different oversight needs than an agent that outputs task definitions for humans to pick up in Jira. Some interactions are synchronous (a human reviewing an agent's proposed action before execution); others are asynchronous (an agent flagging a risk for human review within the next business day).

The practical patterns for human-in-the-loop are:

- Deterministic validation for actions with clear right/wrong criteria (schema validation, business rule checks, test suites). These don't need human review — the validation is the governance.

- Confidence-based escalation for actions where the agent can assess its own uncertainty. Routine actions proceed automatically; edge cases are routed to humans. Anthropic's research on measuring agent autonomy (2025) provides a framework for calibrating these thresholds.

- Mandatory human checkpoints for high-stakes, irreversible actions — not as a blanket policy, but for specifically identified categories (financial transactions above a threshold, external communications, access changes).

The engineering challenge is designing these interaction patterns into the agent system from the start, not bolting them on after deployment. Each orchestration pattern above implies different human oversight integration points.

Conclusion

The agent governance problem is not hypothetical. It is happening now, at organizations that deployed agents without architectural foundations, and the costs — in security incidents, compliance violations, and canceled projects — are real.

But the solution is not to slow down adoption. It's to build the right foundations. This framework addresses the full stack: from personal agent control planes that give employees sovereignty over their tools and data, through autonomous services treated as first-class principals in the organization's identity system, to knowledge graphs that let agents navigate organizational context with provenance and access control at every node, purpose-built data pipelines that treat incoming data as event sources and produce structured knowledge with full traceability, and orchestration patterns designed with human oversight as a product concern from the start.

Each pillar reinforces the others. Identity and permissions (Pillars 1-2) govern who can access what. Knowledge architecture (Pillar 3) structures what agents can reason about. Data pipelines (Pillar 4) build and maintain that knowledge. Orchestration (Pillar 5) coordinates how agents work together and with humans. The organizations that build these foundations now will move faster than those that don't — not because they're taking more risk, but because they're taking less.

This article is the first in The Agile Monkeys' Enterprise Agentic Governance series. Deep dives cover: agent governance and compliance mapping, hierarchical knowledge architectures, purpose-built data pipelines, and model portability. Companion articles cover embedding privacy ("Embeddings Are Not Private") and a secure protocol for cross-team knowledge discovery ("Connecting the Dots").

References

- Cleanlab. "AI Agents in Production 2025." Survey, n=1,837. 2025. cleanlab.ai
- Gartner. "Over 40% of Agentic AI Projects Will Be Canceled by 2027." 2025.
- AIUC-1 Consortium / Help Net Security. "Enterprise AI Agent Security 2026." 2026. helpnetsecurity.com
- Composio. "Why Most AI Agents Fail." 2025. composio.dev
- Silverfort. "Non-Human Identities Outnumber Humans 50:1." 2025.
- Kim, Y. et al. "Towards a Science of Scaling Agent Systems." Google Research, Google DeepMind, and MIT, 2025. arXiv:2512.08296
- IBM. "The Year Companies Stop Building AI Agents and Start Running Them." 2026. ibm.com
- MIT Media Lab. "Authenticated Delegation and Authorized AI Agents." 2025.
- Microsoft. "Agent 365: The Control Plane for AI Agents." 2025. microsoft.com
- CyberArk. "Zero Trust for AI Agents: Delegation, Identity and Access Control." 2025. developer.cyberark.com
- Obsidian Security. "Security for AI Agents." 2025. obsidiansecurity.com
- NIST. "AI Agent Standards Initiative." February 2026. nist.gov
- Cloud Security Alliance. "Agentic Trust Framework." 2026. cloudsecurityalliance.org
- Edge, D. et al. "From Local to Global: A Graph RAG Approach." Microsoft Research, 2024. arXiv:2404.16130
- Gupta, N. et al. "LATTICE: LLM-guided Hierarchical Retrieval." 2025. arXiv:2510.13217
- Glean. "Knowledge Graphs as Agentic Engines." 2025. glean.com
- NStarX. "The Next Frontier of RAG." 2025. nstarxinc.com
- Permit.io. "Building AI Applications with FGA and RAG." 2025. permit.io
- Karpathy, A. "Context Engineering." 2025.
- AWS Builders Community. "RAG Is a Data Engineering Problem Disguised as AI." 2025. dev.to
- Anthropic. "Building Effective Agents." 2025. anthropic.com
- Anthropic. "Measuring AI Agent Autonomy." 2025. anthropic.com
- Waehner, K. "Agentic AI with A2A and MCP using Apache Kafka as Event Broker." 2025. kai-waehner.de
- Confluent. "Event-Driven Multi-Agent Systems." 2025. confluent.io

- IBM. "Agent Communication Protocol (ACP)." Linux Foundation, 2025. ibm.com