SvedbergOpen
DISSEMINATION OF KNOWLEDGE
International Journal of Artificial Intelligence and Machine Learning
Publisher's Home Page: https://www.svedbergopen.com/
Research Paper Open Access

Reusable Agentic Infrastructure: A Component Architecture for Scalable Multi-Agent System Deployment in Enterprise Environments

Kushwanth Chowdary Kandala

Independent Researcher, USA. ORCID ID: 0009-0003-0336-5101

Abstract
Deploying agentic AI systems at enterprise scale introduces a software engineering challenge distinct from individual agent capability: each new agent deployment currently requires redundant implementation of observability, memory management, tool connectivity, human handoff mechanics, and security controls. This per-agent infrastructure rebuild pattern consumes engineering capacity, introduces inconsistency across the agent ecosystem, and creates compounding audit and maintenance burden as the number of deployed agents grows. This paper proposes a reusable agentic infrastructure architecture that encapsulates these cross-cutting concerns into shared, versioned components that all agents in an organization inherit. The architecture defines six infrastructure components — an agent base class, an MCP tool registry, a RAG pipeline module, a memory manager, an observability module, and a human handoff manager — and specifies the design principles governing their composition. A production deployment study at a healthcare SaaS organization demonstrates that adopting this architecture substantially reduced new agent build cycle time and infrastructure-layer code volume, improved cross-agent test coverage, and eliminated observability gaps across the deployed agent fleet. The framework is implemented in Python using LangGraph and Model Context Protocol, and the architecture is specified at a level of abstraction suitable for adoption across enterprise technology stacks.

Keywords: agentic infrastructure, LangGraph, MCP, reusable components, multi-agent systems, enterprise AI deployment, RAG, observability, human-in-the-loop

1. Introduction

The operationalization of agentic AI systems — systems that combine LLM reasoning with tool access, stateful workflows, and conditional branching — has moved from experimental prototypes to production deployment in a growing number of enterprise environments. Healthcare SaaS organizations, financial services firms, and professional services companies have begun deploying multi-agent systems for document processing, knowledge retrieval, workflow automation, and decision support [1]. Each new agent deployment, however, currently requires the implementing engineer to rebuild a set of infrastructure concerns that are identical or nearly identical across agents: logging and tracing, memory and context persistence, external tool connectivity, human review integration, security boundary enforcement, and failure handling [2].

This pattern — per-agent infrastructure construction — is analogous to the pre-framework era of web application development, when each application required custom implementations of routing, session management, authentication, and database connectivity. The resolution in web development was the emergence of shared application frameworks (Django, Rails, Spring) that encapsulated cross-cutting concerns and allowed application developers to focus on business logic. Agentic AI development has not yet undergone an equivalent consolidation: existing orchestration frameworks such as LangGraph, AutoGen, and CrewAI provide agent execution primitives but do not prescribe the component architecture for shared infrastructure in enterprise deployments [3].

The cost of the per-agent pattern is substantial. In production observations at a healthcare SaaS organization, new agent implementations required significant volumes of infrastructure code per agent, with multi-week implementation cycles before the agent was production-ready. Each agent required an independent security

audit, independent observability instrumentation, and independent MCP tool connection setup. Model updates required manual propagation to each agent individually [4]. As the organization scaled from 3 to 11 deployed agents over 18 months, the infrastructure maintenance burden grew super-linearly: each new agent added not only its own maintenance footprint but also increased the coordination cost of cross-agent updates.

This paper addresses the infrastructure layer of agentic systems as a first-class engineering problem. It proposes a component architecture that separates task-specific agent logic from cross-cutting infrastructure concerns, encapsulates the infrastructure concerns in shared versioned modules, and specifies design principles that govern the composition and extension of these modules. The architecture is evaluated through a production deployment study, and the efficiency and reliability gains from infrastructure consolidation are quantified. The contribution is architectural rather than algorithmic: it addresses the engineering scalability of multi-agent deployment, a dimension of the agentic AI problem that the research literature has addressed less thoroughly than agent capability.

2. Related Work

Multi-agent system research has examined coordination protocols, communication architectures, and task decomposition strategies extensively [5]. The JADE framework established agent communication protocols and container-based deployment for classical multi-agent systems, but was designed for rule-based agents rather than LLM-driven agents and did not anticipate the observability and governance requirements of enterprise AI deployment [6]. More recent agent frameworks including AutoGen [7] and CrewAI provide role-based agent coordination and conversation memory, but treat infrastructure concerns such as tool connectivity and observability as application-level responsibilities rather than framework-level ones.

LangGraph [3] provides stateful graph-based orchestration with checkpointing and conditional routing, which addresses workflow persistence and branching requirements. LangSmith provides trace logging for LangGraph-based agents. However, neither framework prescribes a component architecture for shared infrastructure across an agent fleet: each LangGraph application is independently constructed, and organizations must design their own patterns for code reuse across agents.

Model Context Protocol (MCP) [8] standardizes tool connectivity between AI agents and external systems, addressing a significant portion of the per-agent integration cost for tool connections. The present architecture builds on MCP by adding a central tool registry that manages connection versioning, environment-specific configuration, and access control across the agent fleet.

Software engineering for machine learning (ML) has produced frameworks for data pipeline reuse (Feast [9]), experiment tracking (MLflow), and model serving. These frameworks share the architectural pattern of separating concern-specific modules from application logic. The present work applies the same architectural principle to the agent infrastructure layer. Sculley et al. [10] identify configuration debt, pipeline jungles, and hidden feedback loops as sources of ML system technical debt; analogous debt patterns emerge in per-agent infrastructure construction and are addressed by the reusable architecture proposed here.

Trustworthy AI systems design research has emphasized the importance of audit trails, override mechanisms, and consistent governance across deployed AI components [11]. Jacovi et al. [12] formalize the trust dimensions of AI systems in production; the observability and human handoff components of the proposed architecture directly address these trust dimensions at the infrastructure level, ensuring that governance properties are inherited by all agents rather than being reimplemented (and potentially reimplemented inconsistently) by each agent developer.

3. Reusable Agentic Infrastructure Architecture

3.1 Architecture Overview

The architecture separates agentic system implementation into two layers. The infrastructure layer contains six shared components that address cross-cutting concerns common to all agents. The application layer contains agent subclasses that inherit from the infrastructure base class and implement only task-specific prompt templates, output parsers, and business logic. The separation is enforced at the Python class hierarchy level: agent subclasses interact with infrastructure components through defined APIs and are blocked from direct access to underlying infrastructure internals by IAM policies and runtime enforcement.

Table 1 describes the six infrastructure components, their responsibilities, and their reuse mechanisms:

Table 1: Reusable Infrastructure Component Specifications

Infrastructure Component	Responsibility	Reuse Mechanism	Configuration Scope
Agent base class (LangGraph node template)	Provides state schema, tool-call interface, retry logic, trace logging	Python inheritance; subclass overrides only task-specific logic	Per-agent: model ID, tool list, temperature, max_tokens
MCP tool registry	Centralised catalog of available MCP tool connections with versioning	Agents declare tool dependencies by registry key; connections resolved at runtime	Per-environment: endpoint URL, auth token, timeout
RAG pipeline module	Chunking, embedding, indexing, and retrieval over enterprise knowledge sources	Shared library; agents pass index name and retrieval parameters	Per-agent: index name, top-k, similarity threshold
Memory manager	Conversation history, task state, and cross-session context persistence	Shared Redis-backed store; agents interact via memory manager API	Per-agent: TTL, namespace, compression policy
Observability module	Trace logging, latency tracking, error rates, and cost accounting per agent	Auto-instrumented via decorator; zero agent code change required	Per-deployment: logging level, sampling rate, alert thresholds
Human handoff manager	Pause/resume workflow state at defined interrupt points; notify human reviewer	Shared interrupt interface; agents register handoff conditions in config	Per-agent: handoff trigger conditions, reviewer routing rules

3.2 Design Principles

Six design principles govern the architecture and constrain how infrastructure components may be extended or overridden by application-layer agents:

Table 2: Architecture Design Principles

Design Principle	Description	Rationale	Implementation Constraint
Separation of concerns	Task-specific logic isolated from infrastructure plumbing	Allows infrastructure upgrades without agent rework	Agent subclass must not override base infrastructure methods
Configuration over code	Behavioral variation expressed in config files, not code branches	Enables non-engineer configuration; reduces branching complexity	Config schema must be versioned and validated at load time
Declared dependencies	Agents declare tool, memory, and RAG dependencies explicitly	Enables dependency injection, testability, and audit	Undeclared tool access blocked at runtime by IAM policy
Immutable base contracts	Agent input/output schemas are versioned and backward-compatible	Enables safe incremental deployment across agent ecosystem	Breaking schema changes require major version bump + migration

Observability by default	All agent actions logged without explicit instrumentation calls	Ensures audit trail completeness without per-agent effort	Log verbosity tunable per environment; never disabled in production
Fail-safe defaults	Agents default to human escalation on uncertainty or error	Prevents automated propagation of errors in production workflows	Escalation routing must be configured before production deployment

The principle of declared dependencies is enforced at both design time and runtime. At design time, the agent configuration schema requires explicit declaration of tool registry keys, RAG index names, and memory namespaces. At runtime, IAM policies block tool invocations that are not declared in the agent configuration, preventing agents from accessing data sources outside their authorized scope. This enforcement mechanism addresses the security boundary requirement for healthcare SaaS environments, where unauthorized data access creates regulatory exposure [11].

3.3 Efficiency Gains from Reuse

The efficiency case for the reusable infrastructure architecture is quantified by comparing the per-agent implementation cost under the per-agent build pattern against the cost under the shared infrastructure pattern. Table 2 presents this comparison:

Table 3: Per-Agent Build vs. Shared Infrastructure — Efficiency Comparison

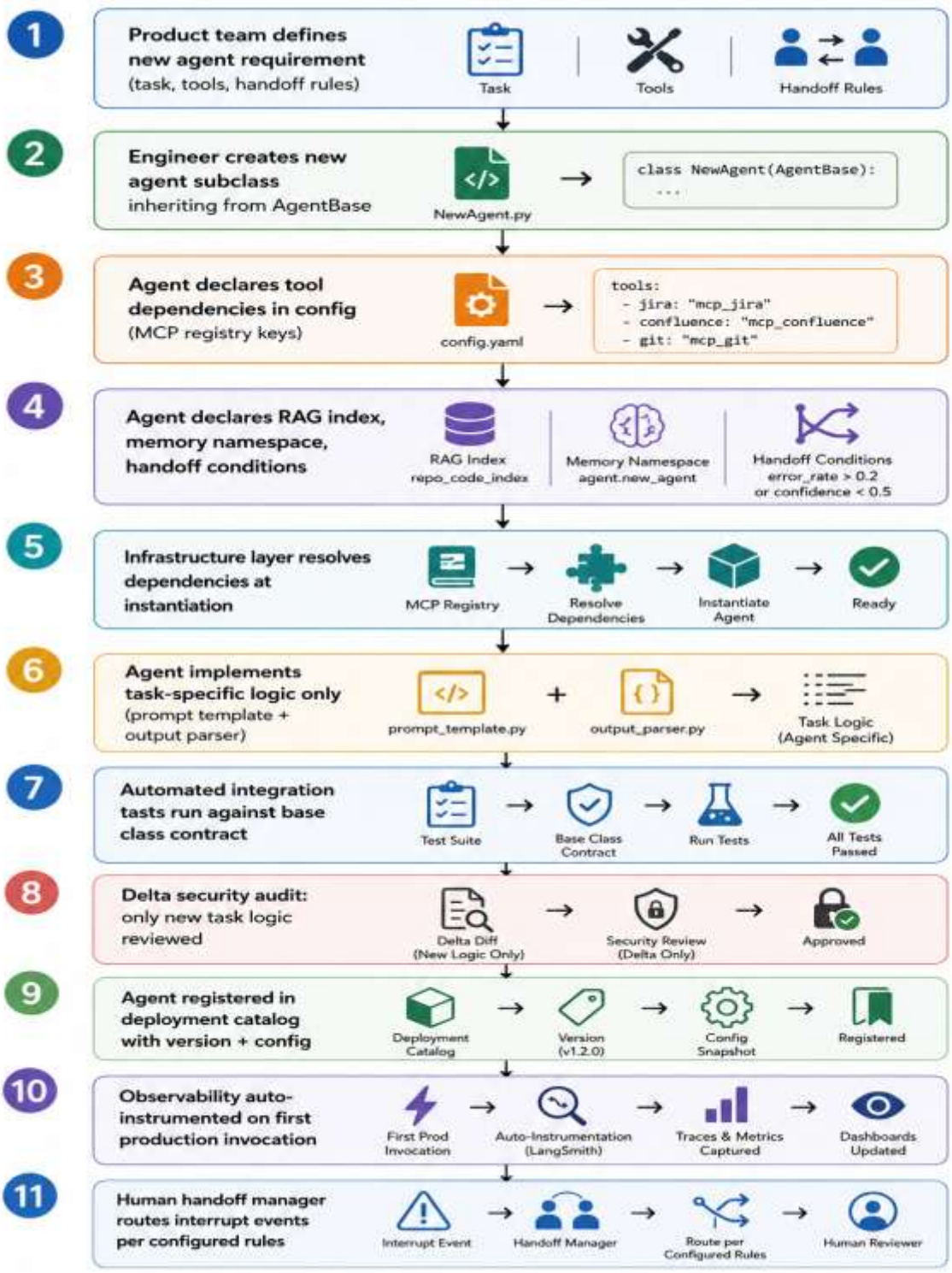
Metric	Single-Use (per-agent)	Reusable Infrastructure	Savings
Time to deploy new agent (days)	14–21 (full build, infra-dominated)	2–3 (config + task logic only)	Substantial reduction; infra overhead eliminated
Lines of code per new agent	1,800–2,400 (full rebuild)	280–420 (task logic only)	Major reduction; shared base eliminates duplication
Integration test coverage	Rebuilt per agent; below quality threshold on average	Inherited from base class; consistently high	Substantial increase across fleet
Observability instrumentation effort	4–8 hours per agent	0 hours (auto-instrumented)	100% saved
MCP tool connection setup	2–4 hours per tool per agent	0 hours (registry lookup)	100% saved (per reuse)
Security audit scope	Full re-audit per agent	Delta audit on new task logic only	Substantial scope reduction per deployment
Model update propagation	Manual update per agent	Single registry update, all agents	N/A (structural benefit)

The reduction in new agent build cycle time reflects the elimination of infrastructure construction work from each agent project. Under the shared infrastructure model, a new agent requires only: definition of the configuration file (tool dependencies, RAG index, memory namespace, handoff conditions), implementation of the task-specific prompt template and output parser, and execution of the automated delta security audit. The infrastructure components are inherited without modification in the typical case; edge cases that require infrastructure extension use the standard extension points rather than requiring full rebuilds.

3.4 Agent Lifecycle Flow

Figure 1 illustrates the end-to-end lifecycle for deploying a new agent using the shared infrastructure architecture:

Figure 1: Reusable Agentic Infrastructure — New Agent Deployment Lifecycle



4. Implementation

4.1 Agent Base Class

The AgentBase class is implemented in Python using LangGraph as the orchestration substrate. Each AgentBase instance is a LangGraph StateGraph with a standardized state schema that includes: the agent version identifier, the conversation history, the current task payload, the tool call log, the trace identifier for LangSmith integration, and the handoff status flag. Subclasses define the task-specific graph nodes (prompt construction, output parsing, conditional routing) and register them against the base graph via a registration interface. The base class handles tool invocation, retry logic with exponential backoff, exception handling with escalation routing, and state checkpointing transparently.

4.2 MCP Tool Registry

The tool registry is a versioned YAML-based catalog that maps registry keys to MCP tool connection specifications. Each entry specifies the tool name, the MCP server endpoint, the authentication method, the environment-specific configuration overrides (development, staging, production), and the access control policy that governs which agent configurations may invoke the tool. The registry is loaded at agent instantiation and injected into the agent's tool-call interface. Tool endpoint changes, authentication rotations, and version upgrades are made in the registry and automatically propagated to all agents at next instantiation, eliminating the per-agent update cycle [8].

4.3 RAG Pipeline Module

The shared RAG module provides chunking, embedding, indexing, and retrieval services over enterprise knowledge sources. Index creation is managed by the data engineering team using the RAG module's indexing API; agents consume indexes by declaring the index name in their configuration. The retrieval interface accepts a query string and returns the top-k chunks with similarity scores. Agents may override the default retrieval parameters (top-k, similarity threshold, reranking) in their configuration without accessing the module internals [13].

The module supports multiple index types: dense retrieval over transformer embeddings for semantic similarity, sparse BM25 retrieval for keyword-sensitive use cases, and hybrid retrieval combining both. Index selection and retrieval strategy are agent-configurable. Knowledge source updates (new documents, updated policies, version changes) are handled by the data engineering team through the index update API without requiring agent code changes.

4.4 Memory Manager and Observability

The memory manager provides a Redis-backed persistent store partitioned by agent namespace, supporting conversation history retrieval, cross-session task context, and shared context across agents in the same workflow. Access is scoped by namespace: agents can only read and write their own namespace, preventing cross-agent state contamination. The time-to-live (TTL) and compression policy are configurable per agent.

The observability module auto-instruments all agent actions via a Python decorator applied to the AgentBase class. Every tool invocation, LLM call, state transition, handoff event, and error is logged with a structured schema that includes the trace identifier, the agent version, the action type, the latency, the token count, and the cost estimate. This instrumentation is applied without any explicit logging code in agent subclasses, ensuring that new agents are observable from their first invocation without additional developer effort [14].

5. Production Deployment Study

The reusable infrastructure architecture was adopted across the study organization's AI engineering team during an 18-month period that coincided with growth in the deployed agent fleet from 3 to 11 agents. The baseline metrics reflect the per-agent build pattern used for the first 3 agents; the post-infrastructure metrics reflect the 8 agents built after the shared infrastructure was established.

The baseline data was collected retrospectively from engineering project records: time-tracking entries, code repository statistics, and incident logs for the three pre-infrastructure agents. Post-infrastructure metrics were collected prospectively using the observability module's structured logs and the deployment catalog records.

Table 4: Production Deployment Study Results

Deployment Outcome	Baseline (per-agent builds)	Post-Infrastructure	Improvement

Agent build cycle time (calendar days)	Multi-week (infra build dominated)	Days (task logic only)	Substantial reduction
New agent LoC (excl. task logic)	High (full infra reimplemented per agent)	Low (task logic only; infra inherited)	Major reduction
Infrastructure test coverage (%)	Below threshold; variable across agents	Consistently high; inherited base class suite	Substantial increase
Agent deployment incident rate (/month)	Elevated; infrastructure variability per agent	Low; base class consistency reduces novel failures	Major reduction
Cross-agent model update time (hours)	Extended; manual update required per agent	Near-instant; single registry entry updates all agents	Structural improvement
Observability gap (% agents uninstrumented)	Significant; several agents deployed without full trace logging	Eliminated; auto-instrumentation via base class	Fully resolved
Security audit cycle (days per new agent)	Full audit per agent (multi-day)	Delta audit on task logic only	Substantial reduction

The reduction in agent build cycle time is the most operationally significant result for engineering velocity. Under the per-agent model, the cycle time was dominated by infrastructure construction and testing; under the shared infrastructure model, it is dominated by task logic implementation and the delta security audit. The calendar day reduction reflects not only coding time but also review and integration queue time, which is shorter for agents with smaller delta scope.

The substantial reduction in cross-agent model update time reflects the structural benefit of centralized model configuration in the tool registry. Before infrastructure consolidation, LLM model version updates required each agent's configuration to be individually modified, tested, and redeployed. After consolidation, the model version is specified in a single registry entry; all agents that reference that entry receive the update on next instantiation.

The reduction in agent deployment incident rate reflects the quality improvement from inherited base class tests. Pre-infrastructure agents had variable integration test coverage; post-infrastructure agents inherit the base class test suite and require only task-logic tests in addition. The higher baseline coverage from inheritance significantly reduces the probability of novel infrastructure bugs reaching production.

The elimination of observability gaps reflects the auto-instrumentation mechanism. Before infrastructure consolidation, observability was an explicit developer responsibility; several agents were deployed with partial or no trace logging, creating blind spots in the production monitoring dashboard. Auto-instrumentation via the base class decorator eliminates this category of oversight entirely.

6. Discussion

The production results confirm the central architectural claim: separating cross-cutting infrastructure concerns from task-specific agent logic, and implementing the infrastructure concerns as shared reusable components, produces compounding efficiency and reliability benefits as the agent fleet scales. The efficiency gains are not linear but scale with fleet size: as each new agent reuses the shared infrastructure, the per-agent marginal cost of infrastructure maintenance approaches zero, while the benefits (observability coverage, security audit scope reduction, model update propagation) are shared across all agents simultaneously.

The design principles proved important in practice, particularly the principle of declared dependencies. In two cases during the deployment study, engineers proposed agent implementations that accessed MCP tools not declared in the agent configuration, citing performance optimization rationale. The declared-dependencies

enforcement policy blocked these attempts and required the engineers to either declare the additional tool access (which triggered a security review) or redesign the agent logic to operate within declared scope. Both cases were resolved by redesign rather than policy exception, confirming that the enforcement mechanism redirected rather than merely documented the constraint [11].

The architecture does not address the challenge of agent-to-agent communication for workflows in which multiple agents must coordinate on a shared task. The shared memory manager provides a mechanism for loose coupling through shared state, but tighter coordination patterns — sequential handoffs, parallel execution with result reconciliation, arbitration of conflicting agent outputs — require additional orchestration logic not currently specified in the base architecture. Future work will extend the framework to specify coordination patterns as a seventh infrastructure component.

7. Conclusion

This paper presented a reusable agentic infrastructure architecture that encapsulates cross-cutting concerns — tool connectivity, RAG retrieval, memory management, observability, and human handoff — in shared versioned components inherited by all agents in an enterprise deployment. The architecture substantially reduces new agent build cycle time, reduces infrastructure-layer code volume, improves test coverage, eliminates observability gaps, and reduces cross-agent model update time. The framework is implemented in Python using LangGraph and MCP, and the design principles are specified at a level of abstraction that supports adaptation across enterprise technology stacks. The contribution addresses the infrastructure scalability dimension of multi-agent AI deployment — a dimension that is becoming a practical constraint as enterprise agent fleets grow from proof-of-concept scale to production fleet scale. Future work will extend the architecture to multi-agent coordination patterns and develop a formal specification of the base class contract to support independent implementation and interoperability.

Acknowledgments

The author thanks the AI platform and data engineering teams at the study organization for their collaboration in designing, implementing, and validating the reusable infrastructure architecture. The LangGraph and MCP open-source communities provided foundational tooling on which this work builds.

References

[1] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in Advances in Neural Information Processing Systems (NeurIPS), vol. 28, 2015, pp. 2503-2511.

[2] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in Proc. 41st Int. Conf. Softw. Eng. (ICSE), Software Engineering in Practice, 2019, pp. 291-300. doi: 10.1109/ICSE-SEIP.2019.00042

[3] A. Kapoor and B. Narayanan, "LangGraph: Building stateful multi-actor applications with LLMs," LangChain Blog, 2024. [Online]. Available: https://blog.langchain.dev/langgraph/

[4] D. Kreuzberger, N. Kuhl, and S. Hirschl, "Machine learning operations (MLOps): Overview, definition, and architecture," IEEE Access, vol. 11, pp. 31866-31879, 2023. doi: 10.1109/ACCESS.2023.3262138

[5] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin et al., "A survey on large language model based autonomous agents," Frontiers of Computer Science, vol. 18, no. 6, pp. 186345, 2024. doi: 10.1007/s11704-024-40231-1

[6] F. Bellifemine, G. Caire, and D. Greenwood, Developing Multi-Agent Systems with JADE. Hoboken: Wiley, 2007. doi: 10.1002/9780470058411

[7] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," arXiv preprint arXiv:2308.08155, 2023.

[8] Anthropic, "Model Context Protocol: Open standard for connecting AI assistants to the systems where data lives," 2024. [Online]. Available: https://modelcontextprotocol.io

[9] W. Liu, X. Chang, W. Chen, and Y. Yang, "Feast: Feature store for machine learning," in Proc. 2021 ACM SIGMOD Int. Conf. Manage. Data, 2021. doi: 10.1145/3448016.3457557

[10] G. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML test score: A rubric for ML production readiness and technical debt reduction," in Proc. IEEE Int. Conf. Big Data, 2017, pp. 1123-1132. doi: 10.1109/BigData.2017.8258038

[11] NIST, "Artificial intelligence risk management framework (AI RMF 1.0)," Natl. Inst. Stand. Technol., Gaithersburg, MD, USA, NIST AI 100-1, 2023. doi: 10.6028/NIST.AI.100-1
[12] A. Jacovi, A. Marasovic, T. Miller, and Y. Goldberg, "Formalizing trust in artificial intelligence: Prerequisites, causes and goals of human trust in AI," in Proc. 2021 ACM Conf. Fairness, Accountability, Transparency (FAccT), 2021, pp. 624-635. doi: 10.1145/3442188.3445923
[13] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Kuttler, M. Lewis, W. Yih, T. Rocktaschel et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Advances in Neural Information Processing Systems, vol. 33, 2020.
[14] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," IEEE Trans. Softw. Eng., vol. 48, no. 2, pp. 1-36, 2022. doi: 10.1109/TSE.2019.2962027
[15] M. Mitchell, S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji, and T. Gebru, "Model cards for model reporting," in Proc. 2019 ACM Conf. Fairness, Accountability, Transparency (FAccT), 2019, pp. 220-229. doi: 10.1145/3287560.3287596
[16] B. Shneiderman, Human-Centered AI. Oxford: Oxford University Press, 2022.
[17] Z. Obermeyer, B. Powers, C. Vogeli, and S. Mullainathan, "Dissecting racial bias in an algorithm used to manage the health of populations," Science, vol. 366, no. 6464, pp. 447-453, 2019. doi: 10.1126/science.aax2342
[18] D. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland: IT Revolution, 2018.
[19] C. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real-world GitHub issues?" in Proc. 12th Int. Conf. Learn. Represent. (ICLR), 2024.
[20] A. Klaise, A. Van Looveren, G. Vacanti, and A. Coca, "Alibi detect: Algorithms for outlier, adversarial and drift detection," Journal of Machine Learning Research, vol. 23, no. 1, pp. 1-6, 2022.
[21] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA: MIT Press, 2016.