SvedbergOpen
DISSEMINATIONOFKNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/



Research Paper Open Access

Architecting AI-Native Systems: A Practitioner's Decision Guide

Ankit Jain

Independent Researcher, India. Email: aam.ankit@gmail.com

Abstract
Understanding that system design has entered an AI-native era is necessary but insufficient. Engineering leaders and architects face a cascade of consequential decisions before a single line of agentic code can be written: which capability on the AI spectrum is actually needed, which foundation model and embedding model should ground the system, which implementation stack balances developer control against operational overhead, and how cloud and infrastructure strategy should be adapted for AI workloads. Poor decisions at any of these layers produce systems that are over-engineered, under-performing, or impossible to govern and maintain. This paper provides a structured, practitioner-oriented decision guide for each of these layers. We introduce the Minimum Necessary Capability principle for navigating the prompt-engineering-to-autonomous-agents spectrum, a model selection framework covering frontier LLMs, mid-tier LLMs, and small language models (SLMs) across commercial and open-weight providers, criteria for embedding model selection including domain specificity and dimension-cost trade-offs, a decision framework for implementation stack selection ranging from no-code platforms to developer-controlled frameworks such as LangChain and LangGraph, and a cloud strategy framework covering single-cloud, multi-cloud, and hybrid approaches for AI-native workloads. Memory architecture is treated as a cross-cutting design concern woven into model selection, implementation stack, and the decision playbook rather than as a standalone decision. A deep reference implementation on AWS maps these abstract decisions onto a concrete cloud service architecture, and a cross-platform comparison to Microsoft Azure and Google Cloud demonstrates provider portability of the four-layer framework. An integrated seven-step decision playbook, a worked greenfield example, a migration playbook, and an anti-pattern catalog translate these frameworks into immediately actionable guidance. Together with its companion paper [1], this work provides architects and engineering leaders with both the conceptual foundation and the practical decision tools needed to design, build, and govern AI-native systems that endure.

Keywords: AI-native architecture, model selection, embedding models, memory architecture, LangChain, LangGraph, RAG, prompt engineering, fine-tuning, cloud strategy, decision framework, software architecture, reference implementation, AWS Bedrock, Microsoft Foundry, Google Vertex AI, Databricks.

I. Introduction

We define an AI-native system as one in which large-language-model inference, vector retrieval, and agentic orchestration are first-class architectural primitives rather than features bolted onto a conventional service. The defining test is whether removing these primitives degrades the system to inutility (AI-native) or merely to a less convenient version of itself (AI-augmented). A companion paper [1] established that system design has entered an AI-native era defined by four fundamental shifts: from static workflows to dynamic orchestration, from batch and streaming pipelines to continuous context, from stateless services to contextual memory, and from monolithic intelligence to specialized agents. That paper proposed a four-layer reference architecture, design patterns, and a maturity model. It deliberately left aside the tactical decisions that arise once the paradigm is accepted: which model, which embedding, which framework, which cloud.

This paper fills that gap with four specific contributions. First, we present a structured decision framework for the five most consequential design domains in AI-native systems (Fig. 1)—capability spectrum, model selection, embedding selection, implementation stack, and cloud strategy—each grounded in observable selection criteria rather than rules of thumb. Second, we provide a deep reference implementation on AWS that instantiates the four-layer architecture of [1] with concrete service mappings, complemented by a substantive comparison across Microsoft Azure and Google Cloud Platform that exposes where the three major clouds genuinely converge and where they diverge in capability rather than merely in naming. Third, we give an explicit treatment of memory and state architecture (Section VII.E), mapping the four-tier memory hierarchy of [1] to specific services and tracing the read/write lifecycle across a single request—a connection that, in our review of the literature, has not been made concretely elsewhere. Fourth, we demonstrate the genericity claim by showing how the same four-layer framework specializes for two contrasting workload types (Section VII.F).

The governing principle that runs through all five domains is minimum necessary capability: start with the simplest approach that solves the problem, and move to more complex approaches only when there is concrete, measurable evidence that the simpler approach is insufficient. This principle resists the over-engineering impulse that characterizes many early AI-native projects, and it directly reduces cost, latency, operational complexity, and governance burden.

A new contribution in this version is a deep reference implementation that instantiates the four-layer architecture from the companion paper [1] onto a concrete AWS cloud stack, complemented by a cross-platform comparison to Microsoft Azure and Google Cloud. By mapping abstract architectural layers to specific services and tracing an end-to-end request flow, we give practitioners a concrete starting point rather than requiring them to assemble the service puzzle from scratch. The cross-platform comparison demonstrates that the decision frameworks are provider-portable and that all three major clouds have converged on the same architectural shape. This section draws on the latest platform capabilities announced through re:Invent 2025 [21], Microsoft Ignite 2025 [22], and Google Cloud Next 2026 [23], and acknowledges the rapid maturation of the Model Context Protocol (MCP) [11] as a first-class interoperability layer across all major cloud providers.

Scope and Method. This is a descriptive architecture paper grounded in practitioner experience designing and operating AI-native systems across multiple enterprises in financial services and retail. The decision frameworks presented here are derived from observed patterns across approximately a dozen production deployments and from synthesis of recent academic and industry literature. Where we cite latencies, costs, or capability claims, we draw on published vendor documentation, peer-reviewed measurement studies, and the companion paper [1]. We have not conducted controlled benchmark experiments specific to this paper; the architecture descriptions and design heuristics should be read as a structured codification of practice rather than as empirical results. Quantitative validation under controlled conditions is identified as future work in Section XIII. The author declares no current employment or financial relationship with AWS, Microsoft, Google, Anthropic, OpenAI, or Databricks; cloud and tooling examples are drawn from public documentation and direct experience but do not represent endorsement of any provider.

The remainder of this paper is organized as follows. Section II develops the AI capability spectrum in depth. Section III presents the LLM and SLM selection framework. Section IV addresses embedding model selection. Section V covers implementation stack selection. Section VI discusses cloud and infrastructure strategy. Section VII presents a deep reference implementation on AWS with a cross-platform comparison to Azure and GCP. Section VIII integrates these into a seven-step decision playbook with worked examples, including memory architecture as a cross-cutting concern assessed at multiple decision points. Section IX covers evaluation and iteration. Section X catalogs common anti-patterns. Section XI discusses governance, compliance, and risk. Section XII addresses keeping decisions current. Section XIII concludes.

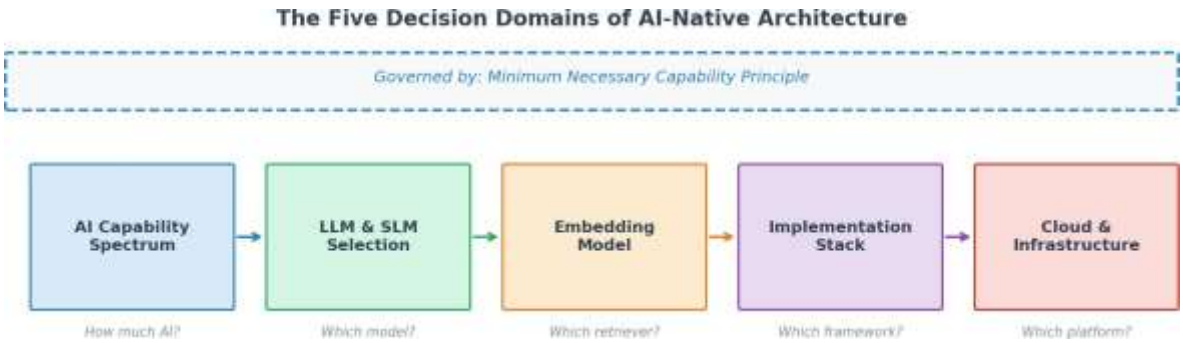


Fig. 1. The five decision domains of AI-native architecture. Each domain represents a consequential choice that shapes cost, capability, and operational complexity. The decision sequence proceeds left-to-right, governed by the minimum necessary capability principle.

THE AI CAPABILITY SPECTRUM IN DEPTH

The companion paper introduced the capability spectrum as a six-rung ladder: prompt engineering, RAG, fine-tuning, continued pre-training, supervised agentic AI, and autonomous agents [1]. This section develops each rung into a full decision framework with entry criteria, exit criteria, cost model, and characteristic failure modes.

A. Prompt Engineering

Prompt engineering is the lowest-cost, lowest-latency, and most underrated rung on the spectrum. A well-crafted system prompt with task decomposition, output format specification, few-shot examples, and negative examples can solve a surprisingly wide range of problems without any retrieval, fine-tuning, or agentic machinery [2]. Systematic prompt engineering should always be the first approach attempted.

Entry criteria: the task is well-bounded, the model already has sufficient domain knowledge, and latency and cost constraints are tight. Exit criteria (move to RAG): the model lacks current or proprietary knowledge, or source attribution is required. Failure modes: prompt brittleness (small phrasing changes cause large output changes), context drift in long conversations, and prompt injection vulnerability [3].

B. Retrieval-Augmented Generation

RAG grounds model outputs in external, current, or proprietary knowledge retrieved at query time [4], [5]. It is the most common productive step up from pure prompt engineering. The RAG design space has expanded rapidly: naive RAG (retrieve-then-generate in a single pass), advanced RAG (with query

rewriting, reranking, and multi-hop retrieval), and modular RAG (composable retrieval pipelines with pluggable components) represent increasingly sophisticated approaches [6], [7].

Five critical design decisions shape RAG quality. Chunking strategy determines how documents are split for embedding and retrieval. Fixed-size chunking is simple but splits content at arbitrary boundaries; semantic chunking preserves paragraph and section structure; hierarchical chunking creates parent-child relationships that allow both summary-level and detail-level retrieval. Retrieval method should combine dense vector search with sparse BM25 search in a hybrid approach; relying on either alone leaves blind spots. Reranking adds a cross-encoder model that re-scores retrieved chunks for relevance; it adds 100–300ms latency but typically improves top-5 retrieval precision by 10–25%. Multi-hop retrieval chains multiple retrieval calls for questions that require synthesizing information from different sources. Embedding model selection—the fifth and often most neglected decision—determines the semantic quality of retrieval itself; a model trained predominantly on web text will perform poorly on legal filings, clinical notes, or code. Section IV treats this decision in depth, but architects should recognize that the embedding model is integral to RAG design, not an afterthought. The training-corpus composition of the embedding model matters as much as its architecture: models trained on filtered web text exhibit different domain-coverage characteristics than those trained on academic or code corpora [25].

Entry criteria: the base model's knowledge is stale, incomplete, or insufficiently domain-specific; source attribution and verifiability matter. Exit criteria (move to fine-tuning): RAG consistently retrieves correct context but the model fails to use it properly due to style, format, or terminology issues. Failure modes: poor chunking that splits meaningful content across boundaries, embedding model mismatch (Section IV), retrieval of irrelevant context that misleads the generator, and excessive context stuffing that degrades generation quality and inflates token cost.

C. Model Fine-Tuning

Fine-tuning adapts a pre-trained model on domain-specific data to internalize tone, format, terminology, or task-specific behavior. It is not a substitute for knowledge injection—fine-tuning shapes behavior, not knowledge [8]. Modern approaches favor parameter-efficient methods: LoRA (Low-Rank Adaptation), QLoRA (quantized LoRA), and prefix tuning, which dramatically reduce compute requirements compared to full fine-tuning.

Entry criteria: prompt engineering and RAG produce consistent stylistic or structural errors that resist instruction correction; sufficient labeled examples exist (typically hundreds to low thousands). Exit criteria: no available base model covers the domain adequately. Failure modes: catastrophic forgetting, overfitting to training examples, treating fine-tuning as a knowledge-injection mechanism when RAG would have sufficed.

D. Continued Pre-Training and Agentic AI

Continued pre-training trains a base model further on a large domain corpus. It is rarely justified in enterprise settings due to data requirements, compute cost, and the need for ML research capability. Open-weight alternatives—using community-adapted models from Hugging Face—often provide a pragmatic substitute.

Supervised agentic AI adds multi-step reasoning, tool use, and synthesis capabilities under HITL or HOTL oversight [1], [14]. Autonomous agents extend this to goal-directed operation with minimal human intervention [15], [19]. Both require mature observability, evaluation harnesses, and governance infrastructure before deployment. The progression from supervised to autonomous should be gated by measurable validation criteria, not ambition. The ReAct pattern [16]—interleaving reasoning and acting in an iterative loop—has proven particularly effective for supervised agentic systems because its reasoning traces are interpretable and auditable. As systems advance beyond stateless prompt-response interactions, memory becomes a cross-cutting design concern. Rather than treating memory as a single decision, we address it at the points where it naturally arises: model selection (context window as short-term memory capacity, Section III), implementation stack (framework support for stateful workflows, Section V), and the integrated decision playbook (memory strategy as a cross-cutting assessment, Section VIII).

Fig. 2 presents a decision matrix mapping task characteristics to recommended spectrum rungs, and Fig. 3 compares cost and latency profiles across the spectrum.



Fig. 2. Capability spectrum decision matrix. Task characteristics (rows) mapped to recommended spectrum rungs (columns). Green indicates primary fit; yellow indicates possible use; red indicates the rung is not appropriate. Values are illustrative and based on practitioner observation.

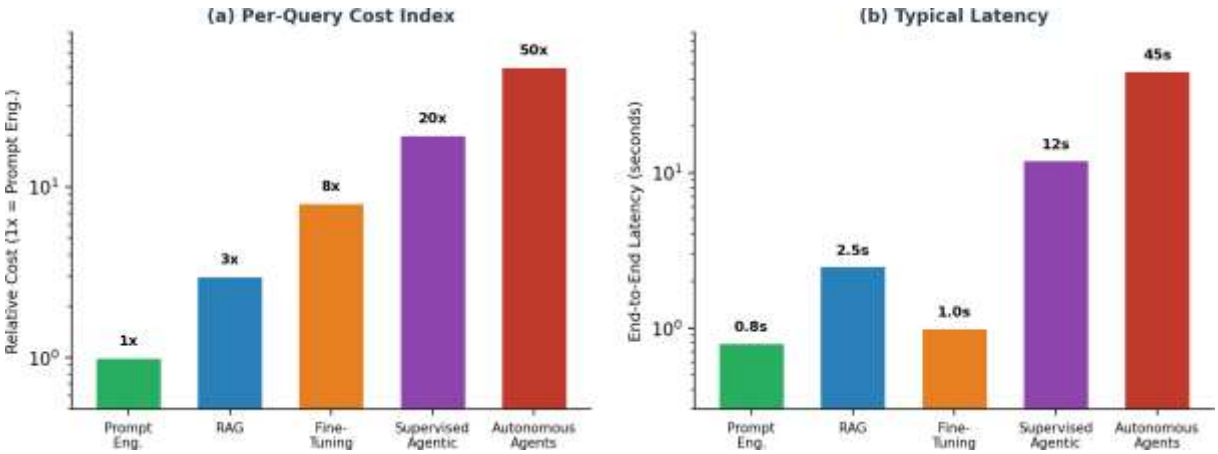


Fig. 3. Cost and latency profiles across the capability spectrum rungs (illustrative). (a) Per-query cost indexed to prompt engineering baseline;

(b) typical end-to-end latency. Note the log scale. Agentic and autonomous approaches are 10–50× more expensive and slower than prompt engineering; use only when the task genuinely requires them.

LLM AND SLM SELECTION FRAMEWORK

Which model should I use? This is the most frequently asked and least systematically answered question in AI-native system design. The answer is almost never the largest or newest model; it depends on task requirements, latency budget, cost ceiling, data privacy constraints, and operational maturity. We propose a three-tier selection framework (Fig. 4).

A. The Three-Tier Model Framework

Frontier LLMs (Claude Opus, GPT-4o [18], Gemini Ultra) deliver the highest reasoning capability but at the highest token cost (typically 10–30× the cost of mid-tier models) and latency (2–5 seconds per call). They should be reserved for complex multi-step reasoning, novel domain synthesis, and high-stakes outputs where quality directly determines business value. Most production systems should use frontier models for less than 10% of their total query volume.

Mid-tier LLMs (Claude Sonnet and Haiku, GPT-4o-mini, Gemini Flash) offer the best cost-capability balance for the majority of production workloads. They handle summarization, classification, structured extraction, and moderate reasoning tasks at 0.5–2 second latency with manageable token cost. This tier is the default starting point for new AI-native systems.

Small language models (SLMs) such as Llama 3.x 8B/70B, Phi-3, Mistral 7B, Gemma, and Qwen are open-weight models that can be self-hosted, eliminating per-token billing entirely. They excel at high-volume classification, simple extraction, edge deployment, and privacy-sensitive workloads where data must not

leave the organization. Latency can be sub-100ms with optimized serving infrastructure [9].



Fig. 4. The three-tier model selection framework. Frontier LLMs provide the highest capability at the highest cost; mid-tier LLMs offer the best cost-capability balance for most workloads; SLMs eliminate token billing entirely for high-volume or privacy-sensitive deployments.

B. Selection Criteria

Seven criteria should drive model tier selection (Fig. 6). (1) Task complexity and reasoning depth: does the task require multi-step reasoning and planning, or is it primarily pattern matching? Frontier models are substantially better at complex reasoning; the gap narrows for simpler tasks. (2) Latency budget: real-time user-facing applications may not tolerate 2–5 second frontier model latency; mid-tier and SLM models can respond in under a second; edge-deployed SLMs can achieve sub-100ms. (3) Cost ceiling: token cost (commercial API) versus compute cost (self-hosted) creates fundamentally different economic models, as analyzed in the companion paper [1]. (4) Data privacy and regulatory constraints: if data must not leave the organization—common in healthcare, financial services, and government—self-hosted SLMs or private-cloud deployments are required regardless of capability considerations. (5) Context window requirements: document length, conversation history, and retrieval payload size determine the minimum acceptable context window; frontier models generally offer the largest windows. The context window also serves as the system's short-term memory—it is the mechanism through which conversation history, prior reasoning steps, and retrieved context are made available to the model. When conversations exceed the context window, architects must decide between truncation, summarization, or sliding-window strategies, each with different fidelity-cost trade-offs. (6) Multilingual requirements: model performance varies significantly across languages; non-English applications should evaluate candidates on target-language benchmarks. (7) Vendor stability: model deprecation risk, API rate limits, pricing changes, and SLA terms differ across providers and affect long-term architectural decisions. The model selection decision tree in Fig. 5 guides architects through these criteria systematically.

C. Cross-Billing-Model Tiering

The most powerful cost optimization pattern is cross-billing-model tiering: routing low-complexity queries to a self-hosted SLM (paying compute cost, not token cost) while reserving commercial frontier API calls for queries that genuinely require advanced reasoning (paying token cost only where it adds value). Implementation requires a lightweight request classifier that assesses query complexity and routes accordingly. The cascade pattern (try the small model first, escalate to frontier only on low confidence) further reduces cost while maintaining quality for complex queries.

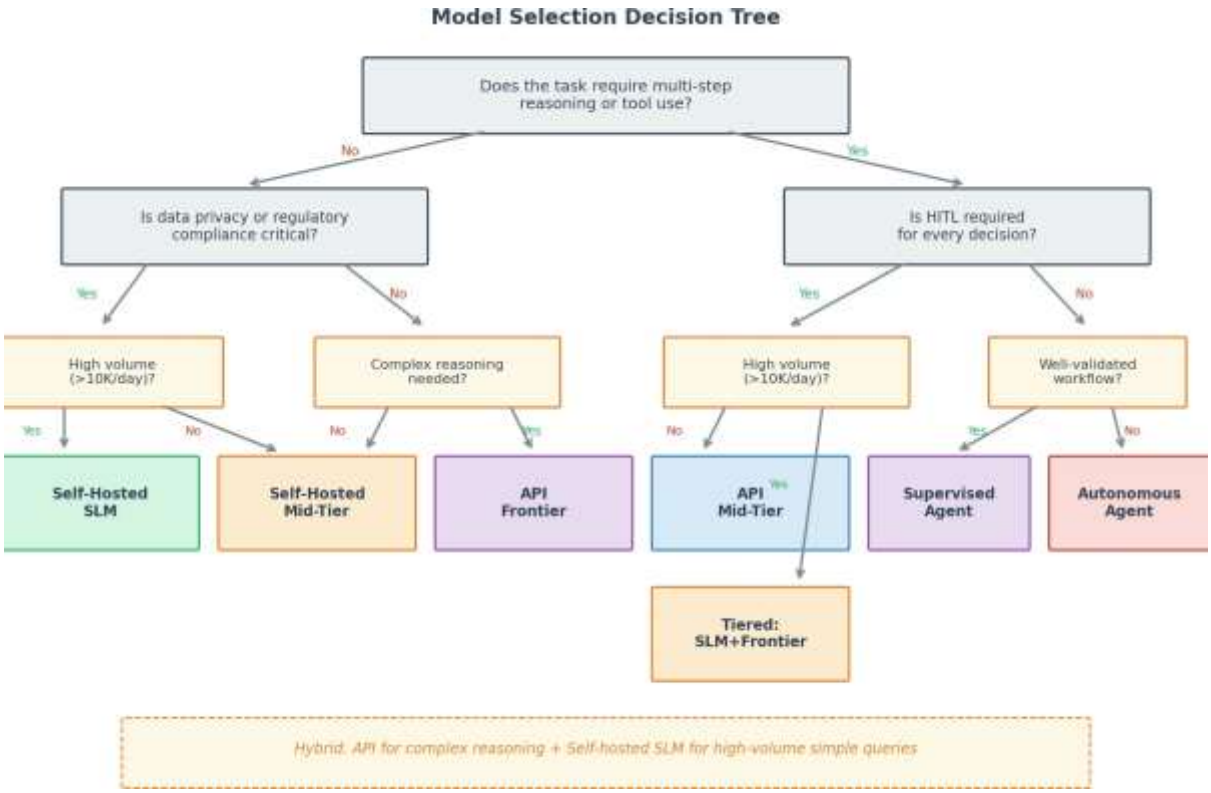


Fig. 5. Model selection decision tree. Starting from task requirements, the tree guides architects through privacy constraints, volume considerations, and reasoning complexity to arrive at the appropriate model tier and deployment pattern.

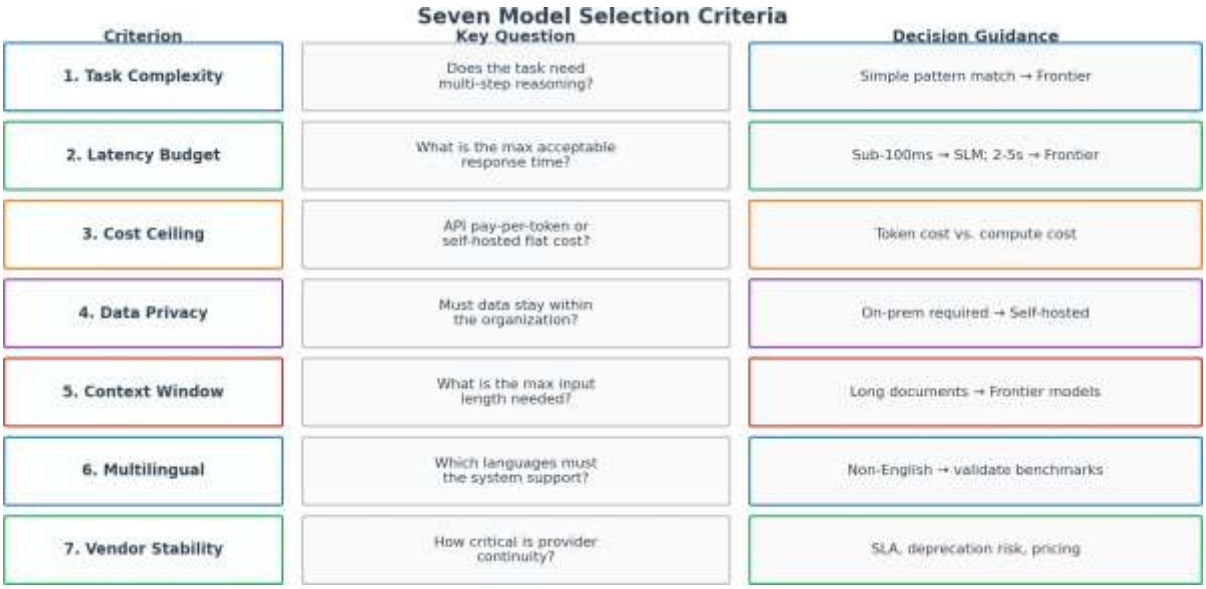


Fig. 6. Self-hosted versus commercial API decision framework. Each factor is evaluated independently; the hybrid pattern (API for complex reasoning, self-hosted SLM for high-volume simple queries) is the most common production architecture for enterprises with diverse workloads.

EMBEDDING MODEL SELECTION

Embedding models are the foundation of every RAG system and vector search capability. A poor embedding model choice silently degrades retrieval quality across the entire system; no amount of generator sophistication compensates for irrelevant retrieved chunks. Despite this, embedding models receive far less architectural attention than generative models.

A. Why Embedding Selection Matters

The retriever is the weakest link in most production RAG systems. When a user reports a hallucination, the root cause is often not the generator inventing facts but the retriever surfacing irrelevant context that misleads the generator [17]. The embedding model determines what semantically similar means for the retrieval pipeline; a model trained predominantly on web text will perform poorly on legal filings, clinical notes, or code.

B. Selection Criteria and Framework

Six criteria drive embedding model selection (Fig. 7): domain specificity (general-purpose versus domain-adapted), embedding dimension (higher captures more nuance but increases storage and compute cost), multilingual support, maximum sequence length (must match chunking strategy), hosting model (API versus self-hosted), and benchmark performance (MTEB scores are directional but insufficient for production selection) [10].

C. Leading Models and Evaluation

The embedding model landscape evolves rapidly. OpenAI's text-embedding-3 family provides strong general-purpose performance with hosted API convenience. Cohere Embed v3 offers the strongest multilingual performance among hosted APIs. The BGE and E5 families are the most widely used open-weight options for self-hosted deployment. Voyage AI provides specialized models for code retrieval. For specialized domains, domain-adapted models such as Legal-BERT and BiomedBERT can outperform general-purpose models significantly on domain-specific retrieval benchmarks.

Production embedding selection requires a domain-specific evaluation set—a curated collection of 100–500 query-document pairs drawn from real user queries. Generic benchmarks like MTEB provide useful directional comparisons but do not predict production quality for specialized domains [10]. The investment in building a domain-specific evaluation set pays for itself many times over by preventing the silent quality degradation that occurs when embedding models are selected based on generic benchmarks alone.

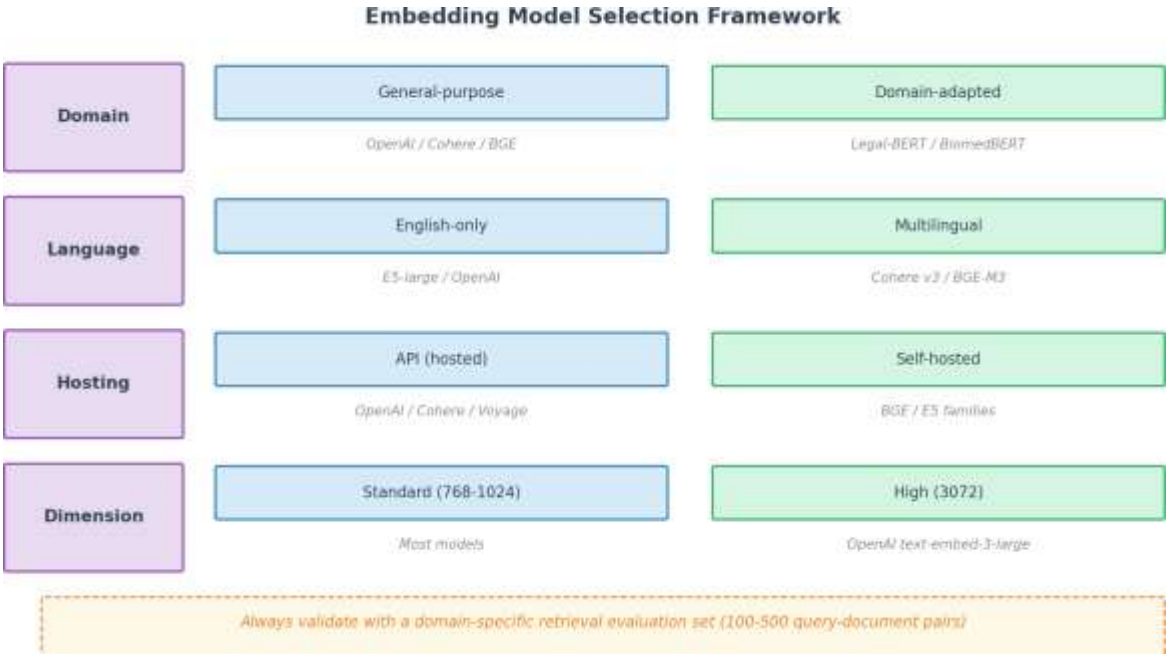


Fig. 7. Embedding model selection framework. For each decision axis (domain, language, hosting, dimension), the framework identifies the appropriate model family. Always validate with a domain-specific retrieval evaluation set; MTEB benchmarks alone are insufficient.

IMPLEMENTATION STACK SELECTION

The implementation stack determines how much control developers have over agent behavior, how quickly teams can prototype, and what operational visibility exists in production. The range runs from no-code platforms where business users configure workflows to direct SDK usage where every routing decision is explicit in code.

A. The Control-Velocity Spectrum

We propose a five-tier control-velocity spectrum (Fig. 8). At the top: no-code and low-code platforms (Microsoft Copilot Studio, Flowise, Dify, n8n) for business-user workflows and rapid internal tooling. In the middle: managed agent services (AWS Bedrock Agents, Microsoft Foundry, Google Vertex AI Agent Builder) for enterprises wanting managed infrastructure; orchestration frameworks (LangChain, LlamaIndex) for broad integrations and fast prototyping; and graph/flow frameworks (LangGraph, AutoGen, CrewAI, Haystack) for complex multi-agent workflows requiring explicit state machines. At the bottom: custom implementations using direct SDK calls for maximum control.

B. LangChain vs. LangGraph vs. Custom

LangChain provides the broadest ecosystem of integrations, document loaders, and sequential chain abstractions. It excels at composing linear pipelines—retrieve documents, transform them, pass to a model, parse the output—and offers a vast library of pre-built connectors to vector stores, APIs, and tools. Its strengths are rapid prototyping and connector breadth. Its weaknesses include abstraction leakage (debugging requires understanding internal implementation details), frequent breaking changes between

versions, and opacity in production debugging where the chain's internal state is difficult to inspect. LangGraph, built by the same team, takes a fundamentally different approach: it models agent workflows as directed graphs with explicit nodes, edges, and state management. This enables cycles (an agent can loop back to re-evaluate), conditional branching (different paths based on intermediate results), parallel execution, and human-in-the-loop checkpoints—capabilities that sequential chains cannot express naturally. LangGraph's graph-based design makes complex multi-agent coordination and stateful workflows first-class concepts rather than bolted-on extensions.

Custom implementations using direct SDK calls (Anthropic, OpenAI, Google APIs) offer maximum control, full observability, and no framework upgrade risk. They require more upfront engineering but produce systems with stable, minimal surfaces that are easier to maintain at scale. The framework ceiling problem deserves explicit attention: many teams reach a point where their use case outgrows the framework's abstractions, and migrating to custom code is expensive because the framework has been woven through the system. Two mitigations: isolate framework usage behind thin abstractions from day one, and adopt MCP [11] for tool and data-source portability so that tools are framework-agnostic.

Memory support is a frequently overlooked dimension of stack selection. LangGraph provides built-in checkpointing and state persistence—conversation history, intermediate reasoning steps, and tool call results are automatically serialized and recoverable, making it naturally suited for multi-turn agentic workflows that require long-term and episodic memory. LangChain offers memory modules (buffer, summary, entity, and vector-backed memory) but these are add-on components rather than architectural primitives, requiring explicit wiring. Managed agent services vary: Bedrock AgentCore provides a dedicated Memory service for episodic cross-session recall; Microsoft Foundry Agent Service offers built-in session management with configurable retention. Custom implementations give full control over memory architecture but require designing the persistence, retrieval, and eviction strategies from scratch. The right choice depends on whether the system needs stateless request-response (no framework memory needed), stateful multi-turn conversations (session memory—any framework suffices), or cross-session personalization and learning (long-term and episodic memory—favor LangGraph, managed services with built-in memory, or purpose-built custom stores).

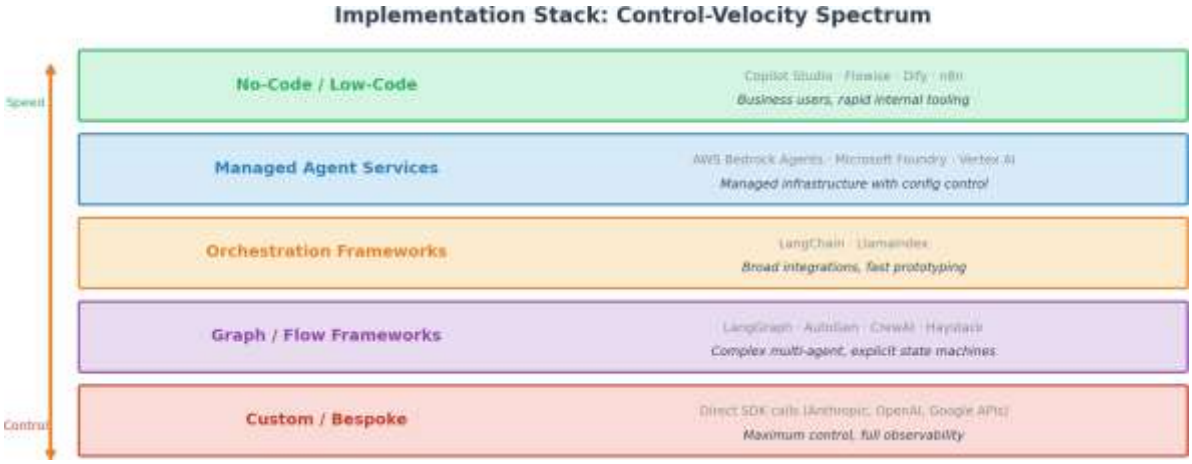


Fig. 8. The control-velocity spectrum for implementation stack selection. Higher tiers offer faster time-to-prototype but less fine-grained control; lower tiers offer maximum control but require more engineering investment.

Cloud And Infrastructure Strategy

Cloud strategy for AI-native systems introduces constraints that did not exist in prior cloud-native architectures: GPU availability and pricing volatility, model API dependency on specific providers, data gravity for retrieval workloads, and regulatory requirements that may demand data sovereignty.

A. AI-Native Infrastructure Requirements

Four infrastructure components are specific to AI-native workloads: GPU compute for self-hosted inference (on-demand, reserved, or spot instances); vector database hosting (managed services such as Pinecone, Weaviate Cloud, and Qdrant Cloud, or cloud-native options such as pgvector on RDS and Azure AI Search); a model gateway providing centralized routing, rate limiting, cost tracking, and provider fallback; and AI-specific observability infrastructure (LangFuse, LangSmith, Arize, or OpenTelemetry extensions).

B. Single-Cloud, Multi-Cloud, and Hybrid

Single-cloud is the right default for most organizations: it minimizes operational complexity, enables cost optimization through committed-use pricing, and provides unified managed services (Fig. 9). Each major provider now offers a comprehensive AI stack: AWS Bedrock with AgentCore and OpenSearch; Microsoft Foundry with AI Search and Foundry Agent Service; GCP Gemini Enterprise Agent Platform (formerly Vertex AI) with Agent Engine and AlloyDB.

Multi-cloud is justified when the best-fit model is on a different cloud, when data residency regulations

require geographic distribution, or when resilience demands provider independence. Hybrid cloud—sensitive data on-premise, inference on cloud—is common in regulated industries. The critical architectural decision, regardless of cloud strategy, is to design the model gateway as a cloud-agnostic abstraction from day one.

C. Cost Optimization Patterns

Four cost optimization patterns are specific to AI-native infrastructure. GPU spot instances can reduce self-hosted inference cost by 60–70% for batch workloads that tolerate interruption. Serverless inference eliminates idle-compute cost for low-volume workloads. Caching layers—semantic caching, response caching, and embedding caching—can reduce effective cost by 30–50% in production systems with repetitive query patterns. FinOps for AI extends cloud cost management to AI workloads: per-request token tagging, showback and chargeback by team or product, cost anomaly alerting that detects runaway agent loops, and token budget ceilings enforced at the gateway level. Without these controls, AI-native systems can generate cost surprises that undermine organizational trust in the investment.

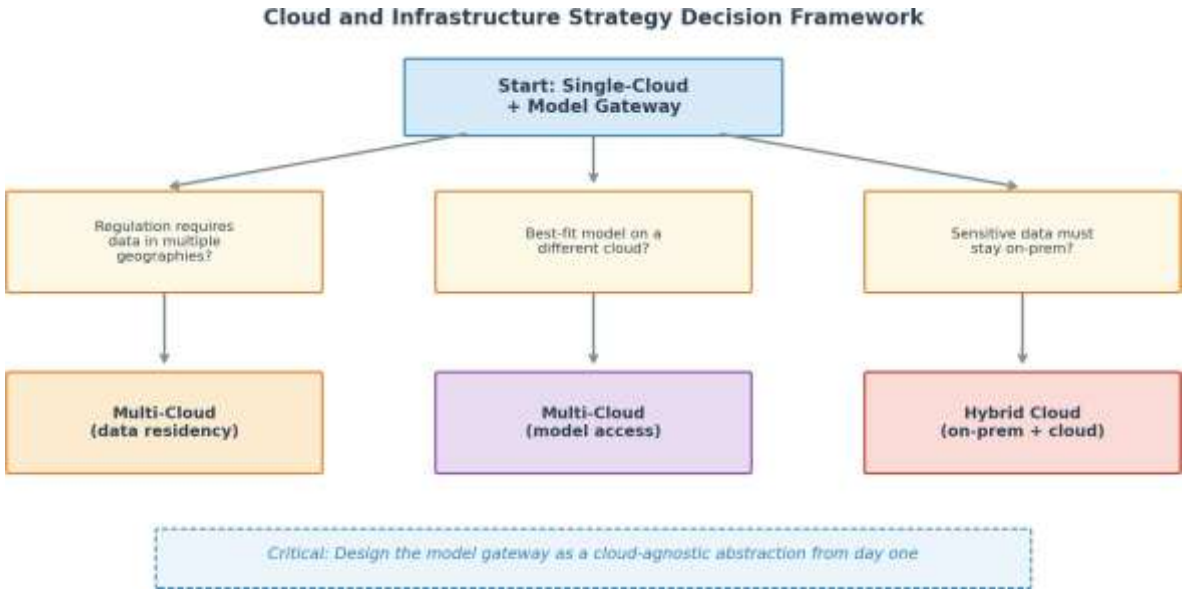


Fig. 9. Cloud and infrastructure strategy decision framework. The default recommendation is to start single-cloud with a model-gateway abstraction layer, expanding to multi-cloud or hybrid only when forced by regulation, model access, or data sovereignty requirements.

Reference Implementation: Instantiating The Four-Layer Architecture

The preceding sections present decision frameworks at a level of abstraction that is deliberately provider-neutral. This section bridges the gap between abstract architecture and concrete cloud services by mapping the four-layer reference architecture from the companion paper [1]—Data Layer, Capabilities Layer, Orchestration Layer, and Application Layer—onto a production-ready AWS stack in depth. We then demonstrate provider portability through a compact cross-platform comparison that maps the same architectural roles to Microsoft Azure and Google Cloud services. The reference implementation reflects the state of each platform as of early 2026, following the significant service announcements at AWS re:Invent 2025 [21], Microsoft Ignite 2025 [22], and Google Cloud Next 2026 [23].

A. Deep Dive: Enterprise AI-Native Architecture on AWS

Before walking through the architecture, we clarify a point of frequent confusion. AWS Bedrock AgentCore, announced at re:Invent 2025 [21], is a suite of seven distinct services rather than a single runtime: AgentCore Runtime (the orchestration engine), AgentCore Memory (long-term and episodic memory store), AgentCore Identity (workload identity for agent-to-tool authentication), AgentCore Gateway (the tool-invocation layer that handles MCP, OpenAPI, and Lambda action group connectivity, including OAuth 2.1 flows on behalf of the runtime), AgentCore Browser Tool (sandboxed web automation), AgentCore Code Interpreter (sandboxed code execution), and AgentCore Observability (trace and evaluation telemetry). In Fig. 10 the AgentCore Runtime appears explicitly in the Orchestration Layer; AgentCore Gateway is shown as a capability bullet within the Runtime because it is invoked by the Runtime, not by external callers; AgentCore Memory is shown in the Data Layer as the episodic store; AgentCore Observability is in the cross-cutting concerns bar; and AgentCore Identity is implicit in the IAM cross-cutting concern. The Browser Tool and Code Interpreter are not shown because they are workload-specific and not part of the generic architecture. This decomposition matters because each subservice has its own pricing, scaling, and SLA, and architects must reason about them independently when sizing a deployment. A second clarification, in light of the implementation-stack discussion in Section V: Fig. 10 instantiates the orchestration layer as Bedrock AgentCore Runtime, but an alternative implementation using LangGraph deployed on ECS or EKS, or a custom Python orchestrator, would replace this single component while leaving the rest of the architecture unchanged. In a LangGraph deployment, AgentCore

Memory is replaced by a LangGraph checkpointer backed by DynamoDB or Redis, AgentCore Gateway is replaced by LangChain tool abstractions plus custom MCP adapters, and AgentCore Observability is replaced by LangSmith or LangFuse. LangChain itself, as a library of prompt templates, output parsers, and tool wrappers, may be used inside either implementation as a code-level dependency rather than an architectural component, which is why it does not appear as a box in Fig. 10. The choice between AgentCore and LangGraph is the control-velocity tradeoff developed in Section V.A: managed-runtime convenience and platform lock-in versus self-hosted control and operational burden.

Fig. 10 presents a generic enterprise reference architecture for AI-native systems on AWS. It is not tied to a specific use case—the same four-layer structure applies equally to an enterprise knowledge assistant, a customer service automation platform, a clinical decision support system, or a software engineering copilot. The worked example in Section VIII.A instantiates this architecture for a specific scenario. Each component is described in detail below.

Agentic Applications Layer. The top layer (Fig. 10) organizes application surfaces in two rows. The upper row shows four agent-ownership categories—Client-Hosted Apps, Enterprise Agents (central-platform shared services such as compliance-checking or document-summarization), Ecosystem Partners (third-party integrations), and Divisional Agents (business-unit-specific). The architectural distinction between Enterprise and Divisional Agents is ownership and entitlements: Enterprise Agents carry open organization-wide entitlements while Divisional Agents carry restricted entitlements enforced by the Agent Discoverability Service's DRA layer. Both register in the same Tools Registry with different governance metadata.

The lower row contains three call-pattern surfaces. MCP Servers (awslabs/mcp and custom) are the programmatic entry point exposing platform capabilities through the Model Context Protocol [11], so external agents and partners can discover and invoke services without bespoke API integrations. Developer Copilots (GitHub Copilot, Claude Code, Q Developer, Codex, Cursor) and Business Copilots (M365 Copilot, Q Business, Genie, Agentforce, Gemini Enterprise) are shown as separate categories because enterprises routinely run multiple copilot products across user personas; they call the platform through MCP tool servers and conversational agent endpoints respectively. All application traffic—UI, MCP, or copilot—routes through API Gateway, which handles IAM authentication, rate limiting, and TLS termination before reaching the Grounding Agent.

Layer Roles and Inter-Layer Contracts. The Orchestration Layer's central component is the Bedrock AgentCore Grounding Agent Runtime, which performs query rewriting, task decomposition, dynamic routing, session memory, and result synthesis. It consults the Agent Discoverability Service (DynamoDB-backed registry) to find which tools and agents can handle each sub-task, then invokes them via AgentCore Gateway. The Capabilities Layer is divided into three sub-regions that the Runtime invokes independently: the Enterprise Agents and Tools Registry (Lambda Action Groups, MCP-compatible third-party integrations, Divisional Agent Tools, Data Scientist Agents); the Data Retrieval Agents (Bedrock Knowledge Bases, Unstructured Retrieval Agent, Text2SQL Agent—all MCP-compliant); and the Prompt and Agent Blueprint Library (reusable templates pulled at deploy time). Bedrock Models (Claude, Nova, Llama, Mistral) and SageMaker Custom Endpoints are alternative inference paths, not peers with a data dependency between them. The Data Layer holds the document corpus and structured stores—S3 with Glue Catalog, OpenSearch Serverless (vector index), RDS/DynamoDB/Neptune (relational, key-value, graph), and Divisional Data Sources—populated by AWS Glue, Step Functions, or Databricks ETL pipelines. The Cross-Cutting Concerns bar (Bedrock Guardrails, AgentCore Observability with 13 pre-built evaluators, CloudWatch/X-Ray, optional LangFuse or Arize Phoenix, IAM and CloudTrail, and the A2A Protocol) spans all four layers and is not confined to any single tier.

Design Rationale: Why These Specific Choices. Several component selections in Fig. 10 deserve explicit justification because the alternatives are not obviously inferior. DynamoDB rather than RDS PostgreSQL for the agent registry is justified by the access pattern: registry lookups are point reads on a stable schema dominated by one-to-many entitlement queries, where DynamoDB single-digit-millisecond p99 latency materially affects orchestration tail latency; relational query expressiveness is not exercised at this point in the request path. OpenSearch Serverless rather than Aurora pgvector for the dense vector index is justified by the scaling profile: vector ingest is bursty and dimensionality is high (typically 1024–3072), and OpenSearch Serverless provides automatic OCU scaling without a primary instance bottleneck. Aurora pgvector remains a defensible choice when transactional consistency between vectors and structured data is required, but for the read-dominated retrieval workload, the operational simplicity of OpenSearch Serverless wins. API Gateway rather than ALB alone is justified by the heterogeneous caller mix: copilots, MCP clients, and partner applications each need per-caller throttling and structured request/response transformation that ALB does not provide; ALB is appropriate when the caller surface is uniform and the dominant concern is L7 routing. Bedrock AgentCore Runtime rather than a self-hosted LangGraph deployment is justified for the same reason as managed Kubernetes is justified over self-hosted: the operational burden of running a stateful orchestration runtime with checkpointing, retries, and observability is substantial, and AgentCore subsumes that burden at the cost of platform lock-in for the runtime layer specifically. The portability principle is preserved by exposing AgentCore behind a

stable internal API that an alternative runtime can replace without disturbing the layers above or below.

Fig. 10. Enterprise AI-Native Reference Architecture on AWS. A generic four-layer framework applicable to enterprise AI-native workloads.

The Bedrock AgentCore Grounding Agent serves as the orchestration hub. The Agent Discoverability Service (AgentCore Gateway + DynamoDB) queries the Tools Registry to route tasks. Data Retrieval Agents and Bedrock Models are separate sub-systems within the Capabilities Layer. Design principles from the companion paper [1] are shown in the sidebar. Cross-cutting concerns including guardrails, observability, and governance span all layers.

B. End-to-End Request Flow

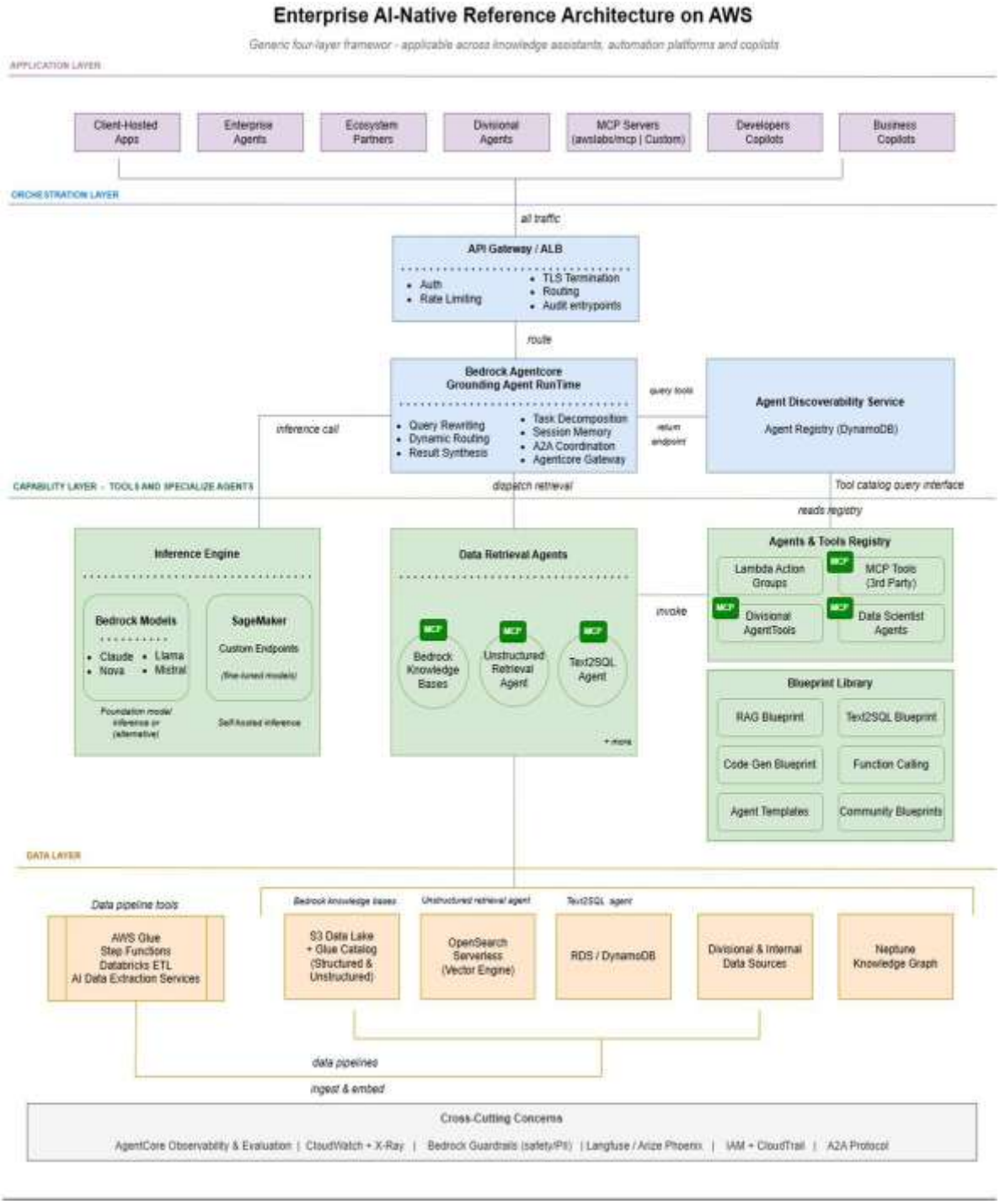


Fig. 11 traces a single request through the architecture in eight numbered steps. (1) User Query: a user submits a natural-language query through a client application or copilot surface. (2) API Gateway: authenticates the request via IAM and enforces rate limits and TLS termination. (3) Prompt Router: Bedrock's Prompt Router checks for a cached prefix match (reducing cost if found) and routes to the appropriate model tier. (4) AgentCore Runtime: receives the routed request in a session-isolated execution context, decomposes the query into sub-tasks, and dispatches retrieval calls. (5) Knowledge Base Retrieval: runs parallel vector and hybrid searches against OpenSearch Serverless, applies semantic reranking, and returns grounded context. (6) LLM Synthesis: the LLM—Claude Sonnet in this example—

synthesizes the response using the retrieved context. (7) Guardrails Validation: Bedrock Guardrails validates the output for PII, toxicity, and policy compliance. (8) Response and Traces: the response is returned to the user; full traces flow to AgentCore Observability, CloudWatch, and the cost dashboard. Every step is logged; no step is opaque. The numbering matches the boxes in Fig. 11 left-to-right.

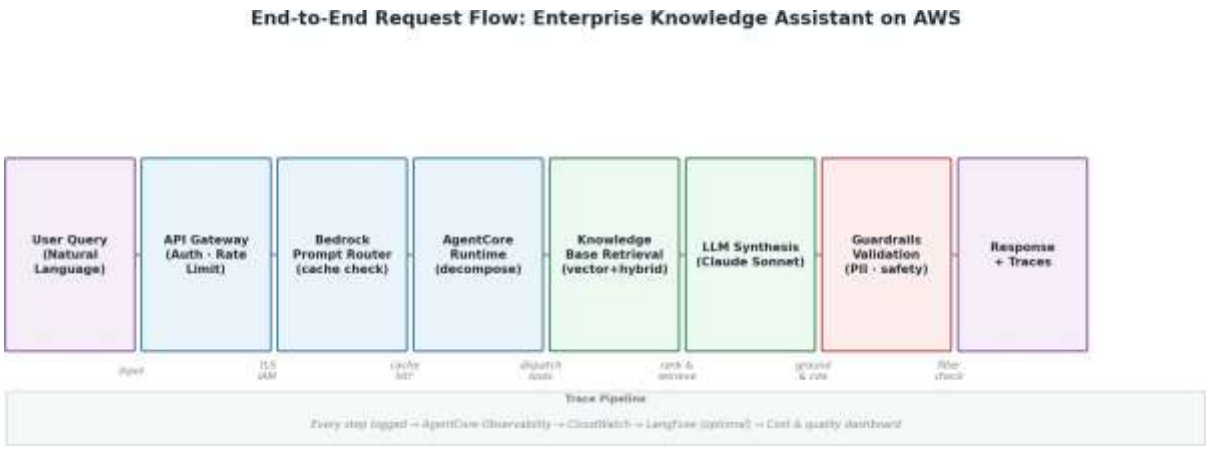


Fig. 11. End-to-end request flow for the enterprise knowledge assistant on AWS. Eight numbered steps from User Query (1) through Response and Traces (8) demonstrate how the four architectural layers interact in a production request cycle. Each step is traced and logged.

C. Cross-Platform Service Mapping: AWS, Azure, and GCP

To demonstrate that the four-layer architecture is provider-portable—not an AWS-specific pattern—Fig. 12 maps the same eight architectural roles to specific services on all three major cloud platforms. The mapping reveals a striking convergence: all three platforms have arrived at the same architectural shape. Each offers a model gateway for routing and cost control (Bedrock Prompt Routing on AWS, APIM GenAI Gateway on Azure, Apigee with Model Armor on GCP), a managed agent runtime (AgentCore, Foundry Agent Service, Agent Engine), a vector store with hybrid search (OpenSearch Serverless, Azure AI Search with Cosmos DB, Vertex AI Vector Search with AlloyDB ScaNN), and a copilot surface (Amazon Q, Microsoft 365 Copilot, Gemini Enterprise). MCP is supported natively across all three. Databricks operates at the data pipeline layer on all three clouds, providing a genuinely cloud-portable data engineering substrate.

Substantive Divergences. Convergence at the architectural-shape level coexists with meaningful capability differences that practitioners must reason about. Five differences are particularly consequential. (1) Memory consistency models differ: AgentCore Memory uses DynamoDB-backed eventually-consistent reads with explicit strong-read opt-in, while Foundry Agent Service persistent memory uses a Cosmos DB session-consistent default; this affects how rapidly cross-session updates become visible to the next agent invocation [21], [22]. (2) Vector store scaling profiles differ: OpenSearch Serverless scales OCUs in 6 GB increments with cold-start latency in the seconds for new indices; Vertex AI Vector Search ScaNN provides sub-50ms p99 even at billion-scale but requires explicit deployment sizing; Azure AI Search uses a different sharding model that favors mixed dense/sparse workloads [23]. These differences matter for systems with bursty embedding-refresh workloads. (3) Tool ecosystem maturity differs: as of mid-2026, AWS exposes the awslabs/mcp catalog of approximately 30 first-party MCP servers; Azure exposes Logic Apps with 1400+ connectors that can be projected as MCP tools; GCP's managed MCP servers are fewer in number but tightly integrated with Apigee for authentication. The right choice depends on whether the dominant integration cost is bespoke tools (AWS favorable) orbusiness-system connectors (Azure favorable). (4) Identity for agents differs: Microsoft Entra Agent ID is a first-class identity primitive [22]; AWS uses IAM with workload identities and AgentCore Identity; GCP uses Cloud IAM with Agent Identity bindings. The Entra approach is more granular today but ties governance more tightly to the Microsoft identity stack. (5) Public exposure pattern differs: GCP's Agent Engine does not expose a public REST endpoint and must be fronted by Cloud Run or Apigee, a constraint that does not exist on AWS or Azure [23]. These differences do not invalidate the four-layer abstraction—they specialize the component-level choices within it.

The practical implication is that organizations designing around the four-layer abstraction rather than around specific service names can switch providers at the component level without rearchitecting the entire system. The model gateway is the critical portability enabler—if it exposes a stable interface (such as the OpenAI-compatible API format), the layers above and below can be swapped with manageable effort.

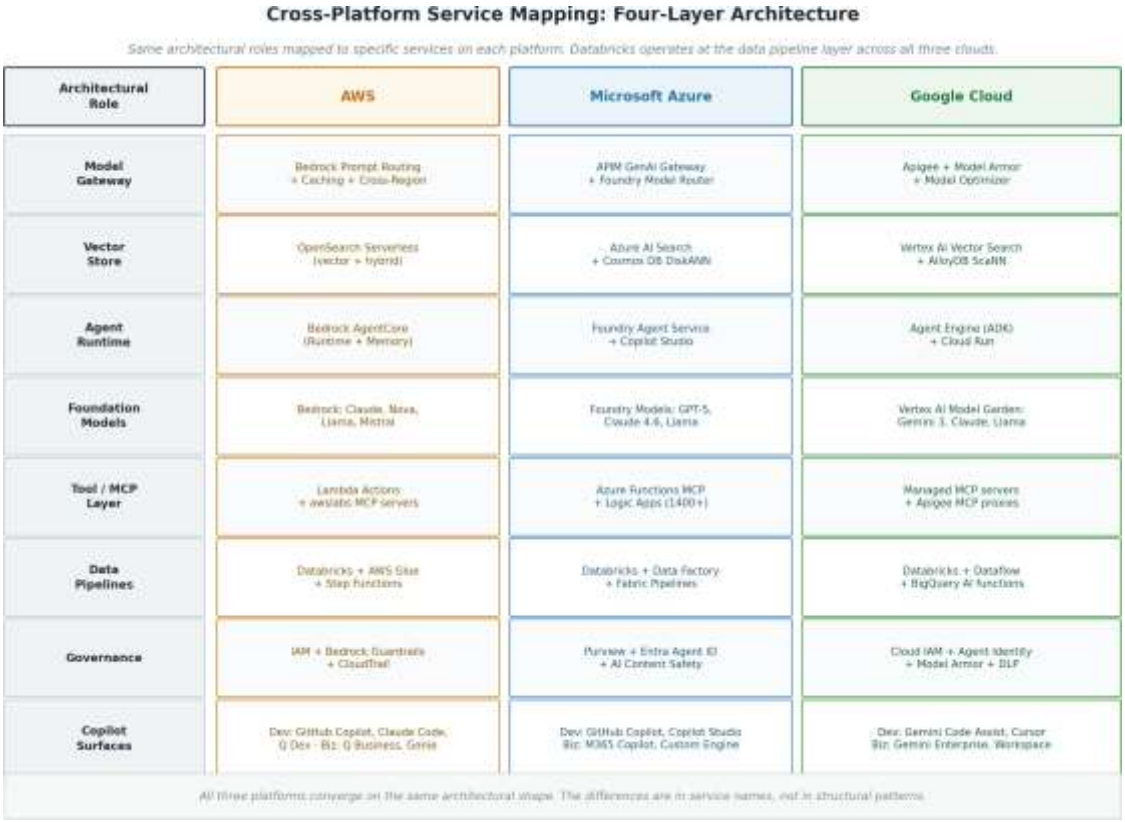


Fig. 12. Cross-platform service mapping. Eight architectural roles mapped to specific services on AWS, Microsoft Azure, and Google Cloud. Databricks operates at the data pipeline layer across all three. All platforms have converged on the same layered shape; the differences are in service names, not in architectural structure.

D. Design Principles for Provider Portability

Three design principles emerge from the cross-platform comparison. First, the model gateway should be treated as a horizontal control plane spanning both the Capabilities and Orchestration layers, not as a single point service. It is where authentication, token quotas, routing, payload logging, and guardrails converge. Second, MCP should be treated as a first-class connector layer between the Capabilities and Application layers, with the registry or catalog explicitly maintained. This is where 2025–2026 architectures most differ from the 2023–2024 LangChain-tool era and where governance, OAuth 2.1, and the forthcoming MCP Registry will land. Third, every component selection should be documented with a provider-portable alternative—not because migration is imminent, but because the exercise of identifying the alternative exposes hidden provider dependencies before they become structural.

Fig. 16 summarizes the eight design principles drawn from the companion paper [1] that govern the reference architecture: Enterprise Mindset, Standardized Interfaces, Interoperability via MCP and A2A, Modularized Design, Unified Access Patterns, Adaptability to Evolving Patterns, Entitlements and Data Resource Authorization, and Observability. The principles are reproduced in Fig. 10 as a sidebar and discussed in detail in [1]; they apply consistently regardless of cloud provider.

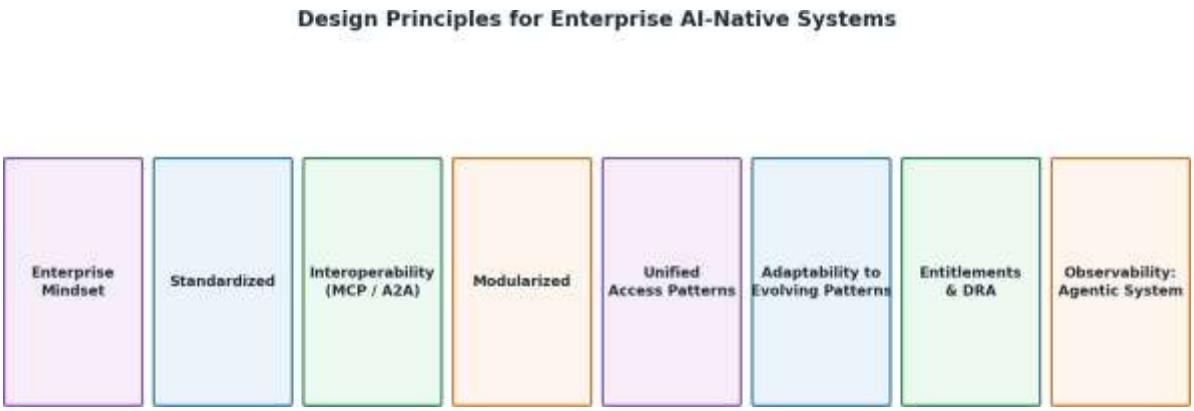


Fig. 16. Design principles for enterprise AI-native systems. These eight principles govern the reference architecture in Fig. 10 and apply regardless of cloud provider. They are drawn from the enterprise agent framework developed in the companion paper [1].

E. Memory and State Architecture in the Reference Implementation

The companion paper [1] identified the shift from stateless services to contextual memory (Shift 3) as one of the four defining transitions of the AI-native era, establishing a four-tier memory hierarchy: Working Memory (in-context, current prompt), Short-term Memory (session-scoped state), Episodic Memory (cross-session interaction history), and Semantic or Long-term Memory (domain knowledge). It also characterized the context window as a scarce architectural resource analogous to CPU cache or network bandwidth, subject to a 'lost in the middle' degradation effect at long lengths [1], [24]. Fig. 17 makes the concrete instantiation of this abstract hierarchy explicit, mapping each tier to specific components in the AWS reference architecture (Fig. 10) and tracing the full state read and write lifecycle across a single request. This subsection aligns the memory discussion woven through Sections II.D, III.B, V.B, and VIII into a unified architectural account.

Working memory is instantiated inside the Bedrock AgentCore Grounding Agent Runtime. It is the assembled prompt

— the structured combination of the current user query, conversation history from the current session, decomposed sub-task instructions, and retrieved context — that the Runtime constructs before each model inference call. The context window is the hard boundary on working memory capacity: anything that exceeds the window must be summarized, truncated, or offloaded. This is where the context window engineering strategies from Paper 1 [1] apply in practice: sliding window (retain the most recent N turns), summarization (compress prior turns into a condensed representation), chaining (route sub-tasks sequentially so each call has a smaller working memory footprint), and RAG (retrieve only the relevant subset of knowledge rather than pre-loading the entire corpus).

Short-term memory is the session state maintained by AgentCore Runtime across the multiple tool calls and model invocations that constitute a single user session. When the Grounding Agent decomposes a complex request into sequential sub-tasks, the intermediate results from earlier sub-tasks must be held while later sub-tasks execute. AgentCore Runtime manages this as session-isolated execution state: recoverable (checkpointed to DynamoDB on each step so a failed agent can resume without re-running prior steps), scoped to a single session, and automatically expired at session end. This is the mechanism that enables multi-step agentic workflows to appear coherent to the user even when they span dozens of internal model calls.

Episodic memory is managed by the AgentCore Memory Service, backed by DynamoDB. The architectural pattern—agents that distill experiences across sessions and reflect on them to update behavior—has been formalized in the academic literature [26]; AgentCore Memory operationalizes this pattern as a managed service. It stores cross-session information: user preferences established in prior sessions, recurring query patterns, past task outcomes (what approach worked, what failed), and any user corrections or feedback from prior interactions. At the start of each new session, the Grounding Agent reads from the episodic store to hydrate its working memory context before query rewriting begins — this is why a well-designed agentic system can greet a returning user with relevant context rather than starting from a blank slate. At session end, a distillation step writes the key facts and outcomes of the completed session back to the episodic store. This write path is often absent in first-generation RAG systems and is one of the most significant architectural differentiators between a static knowledge tool and an adaptive agentic system.

Semantic or long-term memory is the knowledge substrate in the Data Layer. OpenSearch Serverless holds the vector index — the dense representation of the domain corpus that the Bedrock Knowledge Bases and Unstructured Retrieval Agent query for relevant passages. RDS, DynamoDB, and Neptune hold structured semantic facts: curated domain knowledge, entity relationships, and reference data. The Retrieval Agents in the Capabilities Layer are the access mechanism for semantic memory — they translate the Grounding Agent's retrieval intent into specific queries against these stores. The Blueprint Library in the Capabilities Layer represents a fifth memory form not in the original four-tier model: procedural memory — stored reasoning patterns, prompt templates, and agent workflow templates that encode how to approach classes of tasks rather than factual content about the domain. Procedural memory deserves explicit treatment as a peer to the four established tiers rather than as a sub-category of semantic memory because it differs along three dimensions: (a) its representation is executable rather than declarative—a blueprint encodes a procedure for solving a class of problems, not a fact; (b) its update path is curated rather than learned—blueprints are versioned artifacts published by platform engineers, not facts distilled from interactions; and (c) its access pattern is template selection rather than retrieval—the runtime selects a blueprint by task type and parameterizes it, rather than retrieving similar facts by embedding distance. In cognitive psychology this distinction tracks the established separation between declarative and procedural memory; in software architecture it tracks the separation between data and code. We argue procedural memory should be a first-class concern in any AI-native architecture that supports recurring task types, because it is the layer at which institutional best practices accumulate over time.

The state lifecycle in a single request proceeds in seven stages (Fig. 17, panel b). At session start, the Runtime reads episodic memory from DynamoDB to hydrate the initial context. During query rewriting,

the Grounding Agent reads working memory (the current query) and episodic memory (user context and preferences) to reformulate the query. During task decomposition, sub-task state is written to short-term session memory inside the Runtime. During retrieval dispatch, the Retrieval Agents read semantic memory from OpenSearch and RDS. During context assembly, all sources — session state, retrieved context, prior conversation — are combined into the assembled prompt (working memory) and written to the inference call. During model inference, Bedrock reads working memory and writes the completion result back. At session end, key facts and outcomes are distilled and written to episodic memory, and the full trace is written to CloudWatch and the optional observability store (LangFuse or Arize Phoenix). The observability layer thus serves as the system's audit memory — a complete, queryable record of every agent decision, tool invocation, and model output.

Two design implications follow from this architecture. First, the read and write paths are not symmetric. Most RAG systems implement only the read path: retrieve from the knowledge base, generate a response. The write path — distilling session outcomes back to episodic memory, updating the vector index with new documents, correcting the structured knowledge base — is what turns a static retrieval system into a learning one. Designing both paths from day one is significantly easier than retrofitting the write path onto a read-only architecture. Second, memory governance is data governance. Every episodic memory store is a record of user interactions containing potentially sensitive information; every semantic memory store is a source of truth whose provenance must be traceable. The IAM, CloudTrail, and Bedrock Guardrails components in the cross-cutting layer of Fig. 10 apply not just to model inference but to every memory read and write operation. This connection between memory architecture and compliance — particularly under GDPR's right-to-erasure provisions and HIPAA's audit requirements — must be addressed at design time, as it constrains the memory store choices and retention policies available to the system.

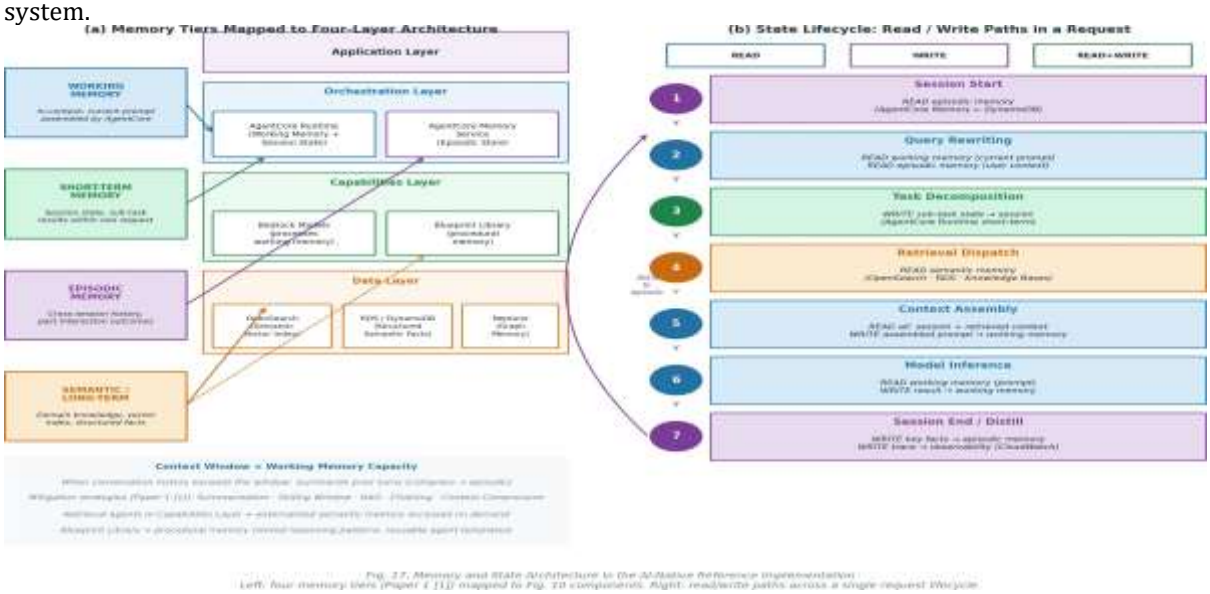


Fig. 17. Memory and state architecture in the AWS reference implementation. (a) The four memory tiers from Paper 1 [1]—Working, Short-term, Episodic, and Semantic/Long-term—mapped to specific components in Fig. 10. The Blueprint Library constitutes a fifth form:

procedural memory. The context window defines Working Memory capacity; mitigation strategies from Paper 1 are listed at the bottom. (b) The seven-stage state lifecycle showing which memory stores are read and written at each step of a single request cycle. The curved arrow shows the distillation write-back from session end to episodic memory—the mechanism that enables cross-session continuity.

F. Specializing the Framework: Two Contrasting Workloads

We claim that the four-layer framework is generic across enterprise AI-native workloads. To make that claim concrete rather than rhetorical, we describe how the same architecture specializes for two contrasting workload types: an enterprise knowledge assistant (read-heavy, retrieval-dominated, conversational) and a software engineering copilot (mixed read-write, code-execution-dominated, IDE-embedded). The exercise illustrates that the layer boundaries hold but that component choices, scaling characteristics, and which subservices dominate differ markedly.

Workload A — Enterprise Knowledge Assistant. The Application Layer is dominated by Business Copilots and conversational web UIs; MCP Servers expose only read-oriented tools (document search, definition lookup). The Orchestration Layer is comparatively simple: most queries dispatch a single retrieval call followed by synthesis, so AgentCore Runtime task decomposition is shallow (typically 1–3 sub-tasks). The Capabilities Layer is dominated by retrieval agents—Bedrock Knowledge Bases and the Unstructured Retrieval Agent carry most traffic; the Tools Registry is small (less than a dozen tools). Inference Engine traffic skews to a single mid-tier model (Claude Sonnet 4.6 in our reference) with frontier escalation rare.

The Data Layer emphasizes vector storage (OpenSearch Serverless dominates volume) and a comparatively static document corpus updated nightly. Memory architecture leans on episodic memory for user research threads and personalization, with low write-back frequency. Cost is dominated by retrieval (60–70 percent) and embedding refresh (10–20 percent), with model inference accounting for the remainder.

Workload B — Software Engineering Copilot. The Application Layer surface is the IDE: GitHub Copilot, Claude Code, Cursor, and Codex via MCP, with no general business-copilot surface. The Orchestration Layer is significantly more complex: code-generation queries routinely decompose into deeper task chains (read repository, locate symbol, plan edit, generate, validate), and AgentCore Runtime invocations carry multiple A2A coordination steps. The Capabilities Layer inverts: the Tools Registry is large and dominates traffic (file system tools, Git tools, build tools, test runners, language servers), while retrieval agents play a supporting role for semantic code search. Inference Engine traffic is bimodal—a small fast model for autocomplete (Haiku-class), a frontier model (Opus-class or Codex equivalent) for multi-file refactoring. The Data Layer is dominated by code-aware vector indices (with code-tuned embedding models) plus the actual filesystem under /workspace mounted into the Code Interpreter; document stores are minimal. Memory architecture is dominated by short-term session state (a single multi-step refactor may span dozens of tool calls) and procedural memory (the Blueprint Library entries for code patterns are heavily exercised); episodic memory plays a smaller role. Cost is dominated by inference (60–75 percent), with tool execution and repository indexing accounting for the remainder.

The framework holds across both workloads, but the cost center, dominant subservice, and architectural emphasis shift. The same layer boundaries and the same orchestration / capabilities / data separation apply; the components within each layer are sized and chosen differently. This is what we mean by genericity: a stable layer abstraction that admits substitution at the component level rather than a one-size-fits-all template.

Integrated Decision Playbook

The preceding sections treat each decision domain independently. In practice, the decisions are coupled: the model tier choice affects the implementation stack choice; the cloud choice constrains managed services; the embedding model must be compatible with the vector database. Fig. 13 presents the seven-step decision sequence that integrates these dependencies.

Memory strategy, treated in detail in Section VII.E, is assessed at three of the seven steps rather than as a single decision. Step 2 selects the memory category (short-term session, long-term persistent, or episodic), Step 3 sizes the context window for short-term capacity, and Step 5 fixes the implementation complexity through the framework choice. The minimum necessary capability principle applies: start with short-term session memory and escalate only with evidence that users need cross-session continuity.

Step-to-Artifact Mapping. Each step in Fig. 13 maps to specific architectural artifacts in the reference implementation. Step 1 (Problem classification) produces no artifact—it is a decision gate that determines whether to proceed at all. Step 2 (Capability spectrum) selects a rung from Fig. 1 and pins the orchestration complexity in the Capabilities Layer of Fig. 10. Step 3 (Model tier) selects the entries that populate the Inference Engine in Fig. 10 and configures the model gateway routing rules. Step 4 (Embedding model) selects the embedding model that drives both ingest pipelines and retrieval agents in the Capabilities Layer; this choice is locked into the vector index in the Data Layer and is expensive to change because it requires re-embedding the corpus. Step 5 (Implementation stack) selects between LangChain, LangGraph, AgentCore Runtime, or custom orchestration—the choice manifests as the implementation of the Grounding Agent in the Orchestration Layer. Step 6 (Cloud) selects which row of Fig. 12 applies, pinning all subsequent service-name decisions. Step 7 (Observability) configures the cross-cutting concerns bar in Fig. 10 (AgentCore Observability, CloudWatch and X-Ray, optional LangFuse or Arize Phoenix, IAM and CloudTrail). The mapping makes the decision sequence operational: each step ends with a specific architectural artifact that is either selected, sized, or configured, and the result of all seven steps is a fully-specified instance of Fig. 10.

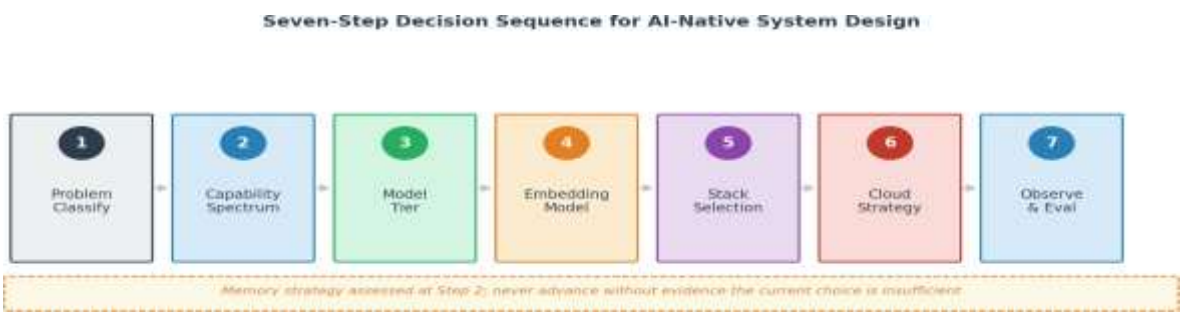


Fig. 13. The seven-step decision sequence for AI-native system design. Each step is governed by the minimum necessary capability principle: never advance to the next decision without evidence that the current choice is insufficient.

A. Worked Example: Enterprise Knowledge Assistant

Consider a financial services organization building an internal knowledge assistant for investment analysts. The system must answer natural-language questions about portfolio companies by synthesizing information from earnings transcripts, regulatory filings, research reports, and ESG data across multiple divisions.

Step 1 (Problem classification): The inputs are unstructured natural language, the knowledge domain is large and evolving, and multi-source synthesis is required. This is a clear AI-native use case. Step 2 (Capability spectrum): RAG is the minimum necessary rung. The memory assessment at this step determines that short-term session memory (conversation history in context window) is sufficient for the initial version; long-term memory for user preferences and prior research threads is deferred until usage patterns are observed. Step 3 (Model tier): Mid-tier LLM (Claude Sonnet) for production traffic; frontier model (Claude Opus) available as an escalation path for complex multi-document synthesis queries. Step 4 (Embedding model): General-purpose model (text-embedding-3-large) validated against a hand-curated financial-domain retrieval test set. Step 5 (Implementation stack): LangChain for initial prototyping, with the intent to evaluate LangGraph if multi-step workflows are introduced later. MCP-compatible tool interfaces from day one. Step 6 (Cloud): Single-cloud (AWS Bedrock for model access, OpenSearch Serverless for vector store)—mapping to Reference Implementation A from Section VII. Step 7 (Observability): LangFuse for trace logging, custom dashboards for retrieval quality and token cost per query.

B. Migration Playbook: Augmenting an Existing System

Many teams face not greenfield design but the question of how to introduce AI-native capabilities into an existing system. We propose a four-phase approach: Phase 1 (Edge augmentation): add prompt engineering at the edges—natural-language intake, summarization, exception triage. Phase 2 (Knowledge grounding): introduce RAG for specific knowledge-intensive features. Phase 3 (Supervised agents): deploy agentic workflows for specific high-value processes, with HITL checkpoints. Phase 4 (Platform integration): build shared infrastructure (model gateway, vector store, observability) and migrate point solutions onto the platform. This phased approach avoids big-bang risk and allows evaluation at each stage.

EVALUATION AND ITERATION

Every decision made in the preceding sections is provisional. Models improve, frameworks mature, cost curves shift, and business requirements evolve. The infrastructure that makes iteration systematic rather than reactive is the evaluation harness. Organizations that skip evaluation infrastructure find themselves unable to answer a basic question: did the change I just made improve or degrade quality? Without that answer, every model upgrade, prompt revision, and architecture change is a gamble.

Offline evaluation uses reference datasets to measure each component independently. Retrieval quality is measured with nDCG@10, MRR, and recall@k against a hand-curated set of query-document pairs. Generation quality is measured with factual accuracy against gold references, format compliance, and citation fidelity. End-to-end task success measures whether the system produced the correct answer regardless of how it got there [12]. For workloads involving multi-step reasoning (math, code generation, multi-document synthesis), task-success measurement should follow the chain-of-thought verification approach [27], where intermediate reasoning steps are scored alongside the final answer to distinguish correct-by-luck from correct-by-reasoning.

Online evaluation runs A/B tests in production, comparing candidate models or configurations against the current baseline on real traffic. Shadow deployment—running the new model alongside the current one without serving its outputs to users—provides evaluation data without production risk. Component-level evaluation isolates retrieval from generation so that a quality regression can be traced to its source. Model swap cadence: model tier decisions should be re-evaluated quarterly because new models release on that cadence and cost structures shift. Embedding model decisions are reviewed annually. Implementation stack decisions are reviewed annually. Regression detection uses automated guardrails that halt deployment if key metrics drop below thresholds—the AI-native equivalent of automated test suites in continuous delivery [12].

Concrete Evaluation Stack. A production-grade evaluation harness for an AI-native system has five distinct layers that must be designed together. (1) Reference datasets: a curated query-document set for retrieval evaluation (200–500 queries with annotated relevance), a generation evaluation set with gold answers (typically 100–300 representative queries with rubric-scored gold responses), and an adversarial set capturing known failure modes (jailbreaks, hallucination probes, off-topic queries). (2) Retrieval metrics: nDCG@10 and recall@10 are the two we treat as primary; MRR is supplementary; we measure them at the chunk level rather than the document level because chunk granularity correlates more directly with downstream generation quality [4], [12]. (3) Generation metrics: factual accuracy via LLM-as-judge with a separate judge model from the system under test (a frontier model judging a mid-tier production model is the standard pattern); citation fidelity via automated extraction and verification; format compliance via deterministic parsers.

(4) End-to-end metrics: task success rate, human-rated answer quality on a representative weekly sample (50–100 queries), and hallucination rate on the adversarial set. (5) Production telemetry: token cost per query, p50/p95/p99 latency by request type, retrieval cache hit rate, model-router decision distribution, and tool-invocation success rate. The evaluation harness should produce a single composite quality score that allows continuous-delivery gating: a deployment is blocked if any primary metric regresses by more than a configured threshold (commonly 2 percent on retrieval, 1 percent on factual accuracy, no regression on hallucination rate).

Numbered Evaluation Loops. Three evaluation loops run on different cadences with different consumers. The fast loop (every commit) runs deterministic tests and a 50-query subset of offline evaluation; it gates merges and runs in under 5 minutes. The medium loop (nightly) runs the full offline evaluation against the current main branch and tracks metric drift over time; it produces dashboards consumed by engineering. The slow loop (weekly or per-release) runs human-rated evaluation on a representative sample and compares against the production baseline via shadow deployment; it produces the go/no-go signal for production releases. Skipping any of the three loops produces a characteristic failure mode: skipping the fast loop allows obvious regressions to land; skipping the medium loop allows slow drift to accumulate undetected; skipping the slow loop allows quality regressions that automated metrics cannot catch (politeness drift, citation degradation, subtle factual errors) to ship to users.

Common Anti-Patterns And How To Avoid Them

Each decision domain has characteristic failure modes. Fig. 14 catalogs the six most expensive anti-patterns observed in practice. Understanding these patterns before they manifest is far less expensive than diagnosing and remediating them in production.

Capability overkill is the most pervasive anti-pattern. Teams deploy full agentic systems for tasks that prompt engineering could solve in a single model call. We have observed a representative case: a customer-service triage system was initially built as a 4-step ReAct agent with three tool calls per query; replacing it with a single well-engineered prompt to a mid-tier model reduced p99 latency from 7.2 seconds to 1.1 seconds and reduced inference cost per query by approximately 18×, with no measurable quality loss on the production query distribution. The cost of capability overkill is not just financial (agents cost 10–30× more per query); it is also operational, because agents introduce failure modes that simple prompts do not: tool invocation errors, infinite loops, inconsistent multi-step reasoning, and dramatically higher observability requirements.

Model prestige bias routes all queries to the frontier model regardless of complexity. In a typical production workload, 70–80 percent of queries are simple enough for mid-tier or small models. A representative case: a knowledge assistant defaulted all queries to a frontier model; introducing a router that dispatched simple lookups to a mid-tier model and reserved frontier for complex multi-document synthesis reduced average inference cost by approximately 65 percent with no measurable quality regression. Embedding neglect produces a system with high generator quality but poor retrieval. We have observed a financial-services system in which switching from a generic embedding model to a domain-tuned alternative improved nDCG@10 from 0.41 to 0.67 with no other architectural change—the generator was never the bottleneck. Framework lock-in makes migration impossible because the framework has been woven through every layer [20]; the remediation is to push framework calls behind a stable internal interface from day one, treating the framework as an implementation detail rather than an architectural commitment. Single-cloud assumption hard-codes infrastructure to one provider; the symptom is usually opaque service-name dependencies in code rather than the explicit four-layer abstraction discussed in Section VII.D. Skipping evaluation infrastructure is the most dangerous anti-pattern because it makes all other problems invisible: a team operating without the evaluation stack described in Section IX cannot distinguish a prompt regression from a model-version change from an upstream embedding refresh, and consequently cannot make any change with confidence.

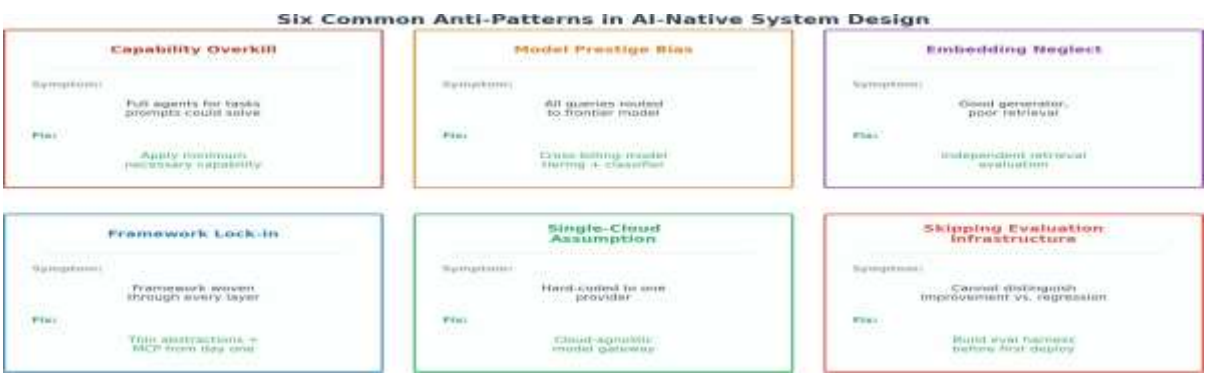


Fig. 14. Six common anti-patterns in AI-native system design, with symptoms and remediation paths. Each anti-pattern corresponds to a failure to apply the decision frameworks in Sections II–VI systematically.

Governance, Compliance, And Risk

Each decision domain introduces distinct governance implications that architects must address at design time, not as an afterthought. Governance decisions are architectural decisions—they constrain the design space in ways that, if discovered late, can require expensive rework.

Model selection and data processing: sending data to a commercial LLM API is a data-processing event under GDPR and may require specific contractual controls, including data processing agreements (DPAs) with the API provider. Organizations must verify that the provider does not retain or train on submitted data. For high-sensitivity domains, self-hosted models eliminate third-party data exposure entirely, though they shift security and maintenance responsibility to the organization.

Regulatory frameworks such as the EU AI Act classify systems by risk level and impose requirements on high-risk applications including transparency, human oversight, accuracy obligations, and conformity assessments. Industry-specific regulations add domain constraints: HIPAA in healthcare requires audit trails and access controls; FINRA and SEC in financial services require record-keeping and supervisory controls on AI-assisted communications [13]. Architects should maintain model cards describing intended use, known limitations, training data provenance, and evaluation results for every model deployed in production.

Keeping Decisions Current

The five decision domains are not one-time choices. Every decision has an expected review cadence (Fig. 15). Model tier decisions should be reviewed quarterly because new models release on that cadence and cost structures shift. Prompt and RAG strategy should be reviewed monthly or when quality monitoring detects drift. Embedding model decisions are reviewed annually or after significant domain expansion. Implementation stack decisions are reviewed annually or after major framework releases. Cloud strategy is reviewed annually or when a provider makes a material change to AI service pricing or availability.

The key organizational discipline is to assign ownership for each decision domain. Without explicit ownership, decisions made at system design time are never revisited, and the system accumulates technical debt as the ecosystem evolves around it. A model-swap readiness checklist ensures the organization can evaluate and deploy a new model within days rather than months: reference evaluation set exists, abstraction layer is in place, deployment pipeline is automated, and rollback is tested [1].



Fig. 15. Decision governance calendar. Each decision domain has a recommended review cadence and a trigger condition for ad-hoc review.

Assigning explicit ownership for each domain prevents decisions from becoming stale as the ecosystem evolves.

Conclusion

Understanding that system design has entered an AI-native era is necessary but insufficient. This paper provides the decision frameworks that translate paradigm understanding into actionable architectural choices. Five decision domains—capability spectrum, model selection, embedding model, implementation stack, and cloud strategy—are developed with structured criteria, decision trees, and illustrative trade-off analyses.

The minimum necessary capability principle governs all five domains: start with the simplest approach that works, and escalate only with evidence. This principle directly reduces the cost, latency, complexity, and governance burden that characterize over-engineered AI-native systems.

A new contribution in this version—the reference implementation section—demonstrates that the four-layer architecture from the companion paper is not merely a conceptual framework but a durable abstraction that maps cleanly onto today's major cloud platforms. The cross-platform comparison reveals that AWS, Microsoft Azure, and Google Cloud have all converged on the same architectural shape; the

differences lie in service names and pricing, not in structural patterns. This convergence validates the vendor-agnostic design philosophy advocated throughout both papers and gives practitioners confidence that investing in the four-layer abstraction is investing in durability rather than in a single provider's roadmap.

Three practical contributions give the paper immediate applicability: the seven-step decision playbook provides a canonical decision sequence; the anti-pattern catalog helps teams avoid the most expensive mistakes; and the decision governance calendar ensures choices are revisited as the ecosystem evolves.

Together with its companion paper [1], this work provides architects and engineering leaders with both the conceptual foundation and the practical decision tools needed to design, build, and govern AI-native systems that endure. Future work will focus on empirical validation of the decision frameworks across diverse enterprise deployments, formalization of the model-swap readiness checklist as a reusable operational standard, longitudinal measurement of the reference implementation patterns as cloud services evolve, and extension of the anti-pattern catalog through structured interviews with practitioners who have navigated these decisions at scale.

References

[1] A. Jain, "Evolution of systems design in AI era," Companion paper, 2025.

[2] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in Proc. 36th Int. Conf. Neural Information Processing Systems (NeurIPS), New Orleans, LA, USA, 2022, pp. 24824-24837.

[3] K. Greshake et al., "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in Proc. 16th ACM Workshop on Artificial Intelligence and Security (AISec '23), Copenhagen, Denmark, 2023, pp. 79-90.

[4] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Proc. 34th Int. Conf. Neural Information Processing Systems (NeurIPS), Vancouver, Canada, 2020, pp. 9459-9474.

[5] Y. Gao et al., "Retrieval-augmented generation for large language models: A survey," arXiv preprint, arXiv:2312.10997, Mar. 2024.

[6] E. Wong et al., "A systematic review of key retrieval-augmented generation (RAG) systems: Progress, gaps, and future directions," arXiv preprint, arXiv:2507.18910, Jul. 2025.

[7] S. Gupta, R. Ranjan, and S. N. Singh, "A comprehensive survey of retrieval-augmented generation (RAG): Evolution, current landscape and future directions," arXiv preprint, arXiv:2410.12837, Oct. 2024.

[8] E. J. Hu et al., "LoRA: Low-rank adaptation of large language models," in Proc. 10th Int. Conf. Learning Representations (ICLR), 2022.

[9] H. Li et al., "Large language model inference acceleration: A comprehensive hardware perspective," arXiv preprint, arXiv:2410.04466, Oct. 2024.

[10] N. Muennighoff et al., "MTEB: Massive Text Embedding Benchmark," in Proc. 17th Conf. European Chapter of the Association for Computational Linguistics (EACL), Dubrovnik, Croatia, 2023, pp. 2014-2037.

[11] Anthropic, "Model Context Protocol specification (revision 2024-11-05)," Anthropic Developer Documentation, 2024. [Online]. Available: https://modelcontextprotocol.io/specification

[12] A. Yehudai et al., "Survey on evaluation of LLM-based agents," arXiv preprint, arXiv:2503.16416, Mar. 2025.

[13] M. Mitchell et al., "Model cards for model reporting," in Proc. ACM Conf. Fairness, Accountability, and Transparency (FAccT), Atlanta, GA, USA, 2019, pp. 220-229.

[14] L. Wang et al., "A survey on large language model based autonomous agents," Frontiers of Computer Science, vol. 18, no. 6, Art. no. 186345, Mar. 2024.

[15] J. Luo et al., "Large language model agent: A survey on methodology, applications and challenges," arXiv preprint, arXiv:2503.21460, Mar. 2025.

[16] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. 11th Int. Conf. Learning Representations (ICLR), Kigali, Rwanda, 2023.

[17] L. Huang et al., "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," ACM Trans. Inf. Syst., vol. 43, no. 2, Art. no. 42, Mar. 2025.

[18] J. Achiam et al., "GPT-4 technical report," arXiv preprint, arXiv:2303.08774, Mar. 2023.

[19] X. Guo et al., "Large language model based multi-agents: A survey of progress and challenges," arXiv preprint, arXiv:2402.01680, Feb. 2024.

[20] D. Sculley et al., "Hidden technical debt in machine learning systems," in Proc. 28th Int. Conf. Neural Information Processing Systems (NIPS), Montreal, Canada, 2015, pp. 2503-2511.

[21] Amazon Web Services, "Architecting for AI excellence: AWS launches three Well-Architected Lenses at re:Invent 2025," AWS Architecture Blog, Dec. 2025. [Online]. Available: https://aws.amazon.com/blogs/architecture/

[22] Microsoft, "Azure AI Foundry: Your AI app and agent factory," Microsoft Azure Blog, Nov. 2025. [Online]. Available: https://azure.microsoft.com/en-us/blog/

[23] Google Cloud, "Gemini Enterprise Agent Platform (formerly Vertex AI)," Google Cloud Products, Apr. 2026. [Online]. Available: https://cloud.google.com/products/gemini-enterprise-agent-platform
[24] N. F. Liu et al., "Lost in the middle: How language models use long contexts," Trans. Assoc. Comput. Linguistics, vol. 12, pp. 157-173, 2024.
[25] G. Penedo et al., "The FineWeb datasets: Decanting the web for the finest text data at scale," in Proc. 38th Conf. Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track, Vancouver, Canada, 2024.
[26] J. Park et al., "Generative agents: Interactive simulacra of human behavior," in Proc. 36th Annual ACM Symposium on User Interface Software and Technology (UIST), San Francisco, CA, USA, 2023, pp. 1-22.
[27] K. Cobbe et al., "Training verifiers to solve math word problems," arXiv preprint, arXiv:2110.14168, Oct. 2021.