DZone Trend Report March 2026

Generative AI

From Prototypes to Production, Operationalizing AI at Scale

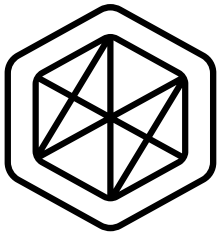
Brought to you in partnership with

denodo

Contents

Editor's Letter

Welcome to the DZone 2026 Generative AI Trend Report



For the past few years, much of the AI discourse has revolved around the models themselves: which is larger, faster, or more capable than the last. New releases arrive almost weekly, benchmarks shift, and the sense of rapid progress is hard to miss. It feels like a lifetime ago since ChatGPT debuted in 2022. What a dizzying pace of change we've seen since.

For engineering teams, the conversation is moving in a different direction. Generative AI, once a topic of experimentation or curiosity, is becoming embedded in everyday development workflows — from generating code and test cases to summarizing logs, supporting internal search, and powering customer-facing apps. For better or for worse (and we're sure that we have readers in both camps!), AI is now increasingly prevalent across the SDLC.

At the same time, the reality of enterprise adoption is more nuanced than headlines might suggest. While most organizations are already using LLMs in some capacity, few have fully integrated GenAI capabilities into enterprise systems. Many teams now find themselves in the uncanny valley between experimentation and operational maturity — AI is unlocking productivity gains, but its true potential is yet to be fully realized.

Building a single AI-powered feature is relatively straightforward. However, building systems that can evaluate model behavior, manage costs and latency, govern data access, and operate reliably in production environments is a far more complex challenge — costly, time-consuming, iterative, and dogged by rigorous governance throughout.

In DZone's 2026 *Generative AI* Trend Report, we explore how organizations are navigating this transition. Through original research and expert perspectives, we take a deep dive into the architectural and operational practices emerging around enterprise GenAI, including retrieval systems, vector databases, and the growing discipline of LLMOps.

Our community contributors share their perspectives from the front lines of enterprise AI, exploring everything from operational maturity and responsible governance to the realities of deploying agentic systems at scale. Together, their insights reflect a shared experience across the industry, and turning promising AI experiments into dependable systems is where the real work begins.

Looking ahead, as GenAI becomes further entrenched in the modern software stack, the real question isn't whether teams will adopt it; it's how they will engineer it. We hope the analyses within prove insightful for you along your AI journey.

Warm regards, *DZone Content and Community*

Your Content and Community team can be found reviewing contributor articles, collaborating with Publications authors, engaging DZone's Core members, and partnering with sponsors to deliver valuable, high-quality learning content that keeps global DZone-ians on top of emerging technologies, tools, and practices.

dzonestaff dzone dzone.com/trendreports

2.1

Key Research Findings

An Analysis of Results From DZone's 2026 Generative AI Survey

Key Research Findings

An Analysis of Results From DZone's 2026 Generative AI Survey

From Pilots to Platforms: The State of Enterprise Generative AI

Enterprise AI is no longer experimental. Our research finds that almost 8 in 10 organizations said that they are now using large language models (LLMs) in some capacity. In large enterprises and among engineering practitioners, this adoption rate approaches universality. Yet overall maturity tells a different story.

64%

of organizations said their GenAI maturity is in the pilot or operational phase

Source: DZone 2026

Only 22% of respondents described their generative AI (GenAI) capabilities as enterprise-integrated or transformational. A clear majority remain in pilot or operational phases. The surface narrative is one of momentum, but structural reality is more uneven: AI is being deployed faster than it's being engineered.

For the DZone audience — the developers, architects, and DevOps and platform engineers reading this report — we believe that distinction matters. Our research suggests the constraint you are facing is no longer model access; it is architectural discipline.

Horizontal Adoption, Vertical Integration

The spread of LLMs across organizations is broad and largely decentralized. Teams are embedding code assistants into continuous integration workflows, experimenting with retrieval-backed chat interfaces, and integrating summarization tools into internal support systems. Of the 79% actively using LLMs, 43% are still in pilot programs, while 36% reported production deployments.

Splitting the data by company size and scale exposes some coordination gaps. The capabilities inside smaller firms may be proliferating at the team level, but they are not always consolidating into a shared infrastructure.

Deploying a single AI-powered feature is relatively simple. However, it is far more complex to standardize model selection, establish versioning discipline, centralize embeddings, implement regression testing, and monitor probabilistic outputs across environments. In turn, the research reaffirms that AI adoption is horizontal, with features appearing everywhere, but that integration is vertical — and requires deliberate platform design.

This is where the key engineering challenge emerges. Without shared inference gateways, standardized retrieval layers, and consistent evaluation pipelines, AI remains fragmented. The results of this fragmentation are duplicated effort, inconsistent guardrails, and uneven observability.

Our findings suggest the difference between experimentation and industrialization is not capability. In fact, it is cohesion.

AI as a Productivity Layer

The distribution of use cases among respondents reinforces this view. GenAI is not concentrated around a single breakthrough application. Code generation is the most common use case at 56%, followed by content generation at 51%, chatbots at 49%, and process automation at 44%. Enterprise search and early-stage agentic workflows lag behind in the mid-30% range.

> For now, AI is augmenting workflows more often than it is replacing them

This clustering itself is revealing. LLMs are functioning less like specialized tools and more like horizontal accelerators embedded across workflows and the SDLC. For many, AI is compressing time: Developers use it to scaffold services, generate test cases, refactor legacy code, and summarize logs. Support teams ground chatbots in internal documentation. Operations teams automate repetitive ticket classification. These are incremental gains, but they unlock greater advantages as they compound.

Delivery models reflect the same pragmatism. A third of respondents primarily use AI through internal employee tools, while 26% deploy it in customer-facing applications. Only 16% rely on shared cross-team platforms, and just 10% reported autonomous multi-step systems operating at scale.

For now, AI is augmenting workflows more often than it is replacing them. Re-architecting core enterprise systems around agentic orchestration remains limited, with the dominant trend among our audience being one of acceleration rather than transformation.

LLMOps is Becoming a Core Discipline

As the use of LLMs expands, governance is no longer peripheral. Our findings show that human-in-the-loop reviews appear in 62% of deployments, while automated evaluation for quality, bias, and hallucination risk stands at 58%. Prompt versioning and tool-level guardrails hover around 50%, and runtime monitoring and drift detection lag at 36%.

These rates suggest that organizations are formalizing review mechanisms, but lifecycle observability is still maturing.

In practice, production-ready AI systems require more than manual oversight. They demand regression testing across curated "golden" datasets, structured output validation against defined schemas, and systematic evaluation whenever prompts or models change. They require canary rollouts for model upgrades, visibility into token usage and cost behavior, and drift monitoring — not only for model outputs but also for embedding quality as documentation evolves.

Our research shows that security teams are most likely to emphasize evaluation rigor, while DevOps teams lean toward deployment discipline and guardrails. The more experienced the practitioner, the more likely they are to skew more strongly toward assurance-first controls.

It is a pattern that mirrors the early DevOps era. Rapid iteration without instrumentation works . . . until it doesn't.

There is perhaps a need to more actively acknowledge how probabilistic systems behave differently from deterministic code. Minor prompt changes can produce divergent outputs. Model upgrades can alter behavior in subtle ways. Without structured evaluation and monitoring, variability becomes risk. In turn, LLMOps is emerging as infrastructure rather than merely an add-on.

The Returns of GenAI Are Operational and Compounding

Despite uneven maturity, respondents said the reported benefits of GenAI are tangible and cover multiple bases. These include faster time-to-market leads (68%), reduced manual work (62%), increased developer productivity (61%), improved access to knowledge (59%), and a better customer experience (51%).

79%

of organizations said they are already using LLMs

Source: DZone 2026

These gains are all pragmatic, functional, and materially useful rather than performative. AI is accelerating execution before it reshapes strategy.

For DevOps teams, our findings suggest that improvements in delivery speed are especially visible with AI-generated test cases reducing bottlenecks, automated code scaffolding shortening development cycles, and log summarization accelerating incident response. Faster knowledge retrieval can also reduce context-switching friction.

Individually, these changes may appear modest, but when repeated across teams and sprints, they can significantly sharpen velocity curves. Our data suggests that the early advantage lies in throughput optimization rather than business model reinvention. The takeaway is that structural efficiency, once embedded, tends to persist.

Reliability Is Driving Architectural Decisions

Retrieval-augmented generation (RAG) represents the clearest signal of infrastructure evolution among respondents, with 69% of organizations having implemented RAG, split between pilot and production deployments. Adoption increases sharply among the largest enterprises.

Our research shows that the leading RAG use cases — chatbot accuracy, research and data analysis, customer support, and code search — all share a common theme: Trust is under scrutiny.

RAG shifts the system contract, so instead of relying on a model's internal knowledge, organizations ground outputs in controlled data sources. This shift introduces new engineering considerations that include how:

- Chunking strategy affects retrieval quality
- Embedding freshness impacts accuracy as documentation evolves
- Latency tradeoffs emerge between retrieval depth and response speed
- Hybrid search models combine vector similarity with keyword scoring to improve precision

Vector databases reinforce this direction of travel. Three quarters of respondents reported using them, though most remain in experimental stages. This is no longer fringe thinking but mainstream infrastructure, with adoption among larger enterprises and engineering roles approaching ubiquity.

The emerging AI stack is taking shape and becoming recognizable: model inference layered over retrieval, backed by vector search, and instrumented with governance and evaluation.

Architectural tradeoffs are unavoidable. Centralization can deliver consistency at the cost of agility, while decentralization can increase flexibility but invite fragmentation. Smaller models, when paired with robust retrieval, can rival larger systems on performance per dollar — but only if the surrounding architecture is carefully engineered.

These are not abstract debates; they are crucial considerations that shape performance, cost, and reliability in production systems.

The Emerging Maturity Gradient

Our research findings reveal a consistent gradient, wherein larger organizations are more likely to productionize retrieval infrastructure and vector services, while smaller firms experiment more freely but lack consolidated platforms. Engineering practitioners reported higher adoption rates than business leadership, while senior respondents emphasized evaluation and risk.

These results suggest a structural tension between velocity and control. Organizations that reconcile these forces — i.e., enabling rapid experimentation while institutionalizing governance and shared infrastructure — will be those that move beyond the pilot phase in a more meaningful way. Those that do not risk fragmentation: isolated use cases, duplicated embedding pipelines, and inconsistent guardrails.

> The message to bear in mind is that while AI adoption is inevitable, operational excellence is not

The message to bear in mind is that while AI adoption is inevitable, operational excellence is not.

Nearly every organization is using or evaluating LLMs, even if the majority remain in transitional maturity states. Our findings suggest a clear rise in governance practices, greater consolidation in infrastructure layers, and increasingly measurable productivity gains. The competitive edge is shifting accordingly.

The winners will not be the teams that deploy AI features fastest but those who design cohesive systems with shared retrieval layers, disciplined evaluation pipelines, structured output validation, cost-aware inference strategies, and observable model lifecycles. Treat AI as a feature, and you gain efficiency. Treat AI as infrastructure, and you gain resilience.

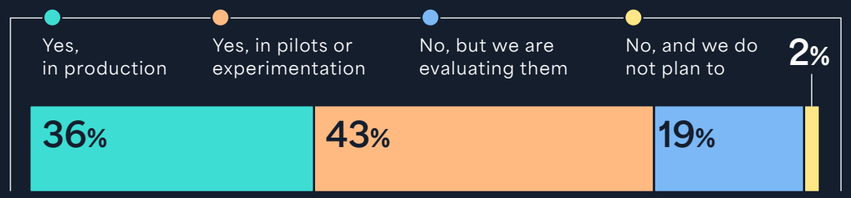
It's clear that GenAI adoption is now mainstream. The next frontier is industrialization — and that frontier is defined by engineering.

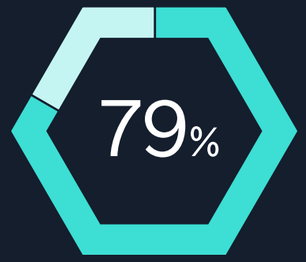
Inside the Enterprise AI Shift: From Pilots to Production

LLM ADOPTION IS NOW TABLE STAKES

For most engineering teams, LLMs have shifted from novelty to standard tooling. Competitive advantage no longer comes from model access – it comes from how effectively teams integrate AI into real development and delivery pipelines.

Does your organization use large language models (LLMs)?





of respondents say their organization already uses large language models internally

MOST TEAMS ARE STILL BRIDGING THE PRODUCTION GAP

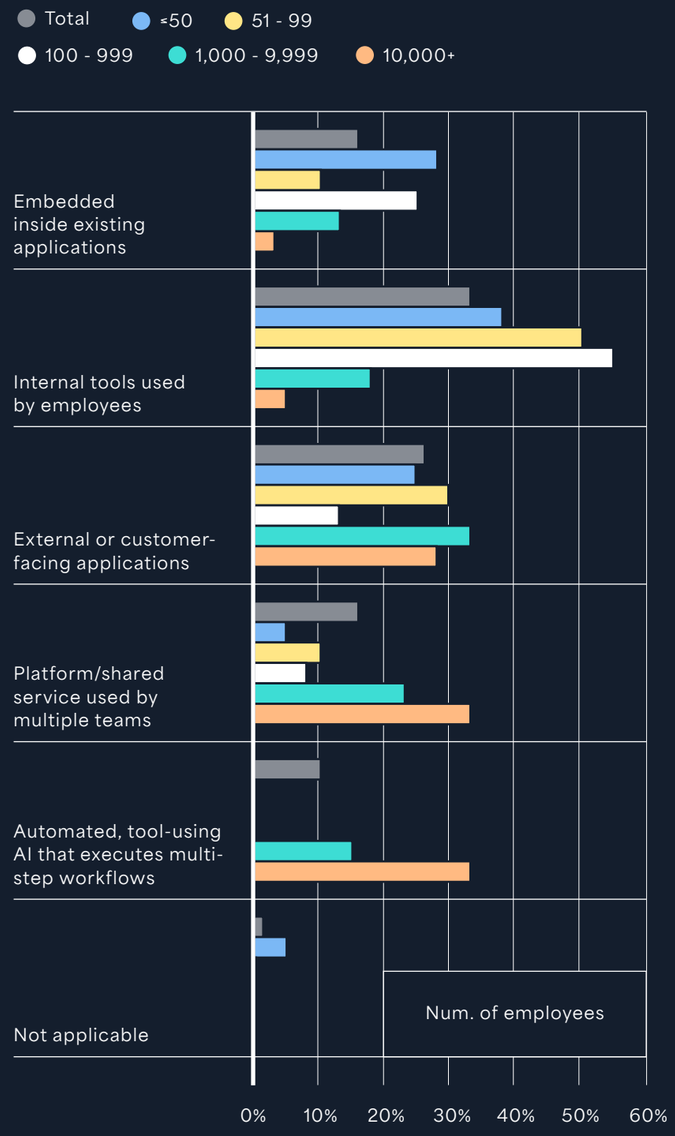
Experimentation is easy; scaling is hard. Many organizations have working pilots, but far fewer have repeatable patterns for deploying AI across environments. The constraint isn't model capability – it's production engineering discipline.

How would you describe your organization's overall GenAI maturity?

AI IS ACTING AS A PRODUCTIVITY LAYER, NOT A REPLACEMENT

GenAI is enhancing existing workflows more than replacing systems. It's improving how developers write code, automate tasks, and support users. However, widespread rearchitecting of enterprise applications remains limited.

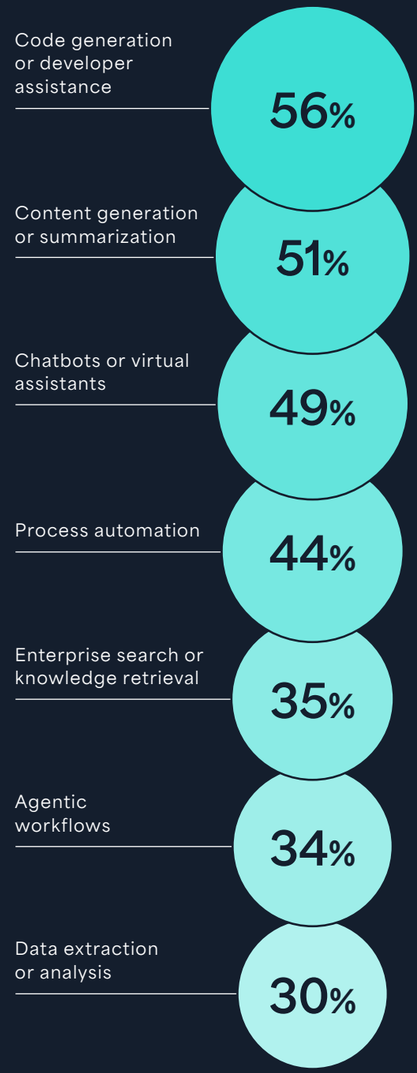
How would you describe your organization's overall GenAI maturity?



ADOPTION IS HORIZONTAL, NOT SINGULAR

There's no single breakthrough application. Engineering teams are applying LLMs across coding, automation, search, and customer support simultaneously. Our findings signal that GenAI is becoming a general-purpose capability.

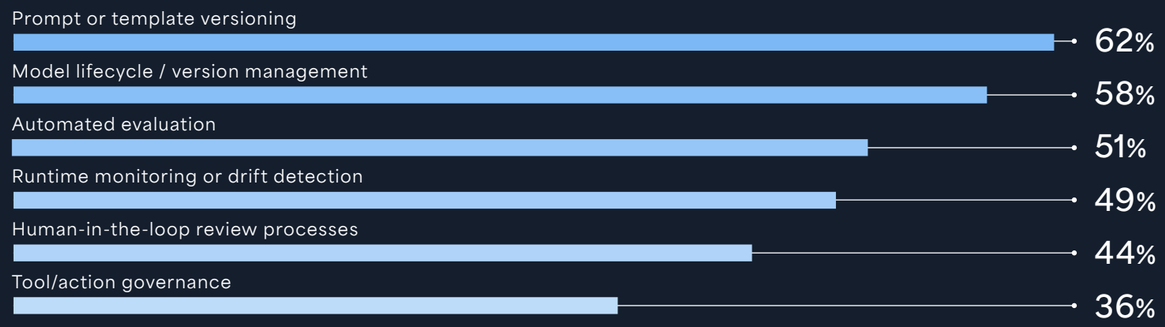
In what ways does your organization primarily use LLMs? [Top 3 use cases]



LLMOPS IS EMERGING AS THE NEW DEVOPS

As adoption scales, evaluation, monitoring, and governance become mission-critical. The most mature teams treat AI systems like any production service – one that is tested, observable, versioned, and governed – rather than simply an API call.

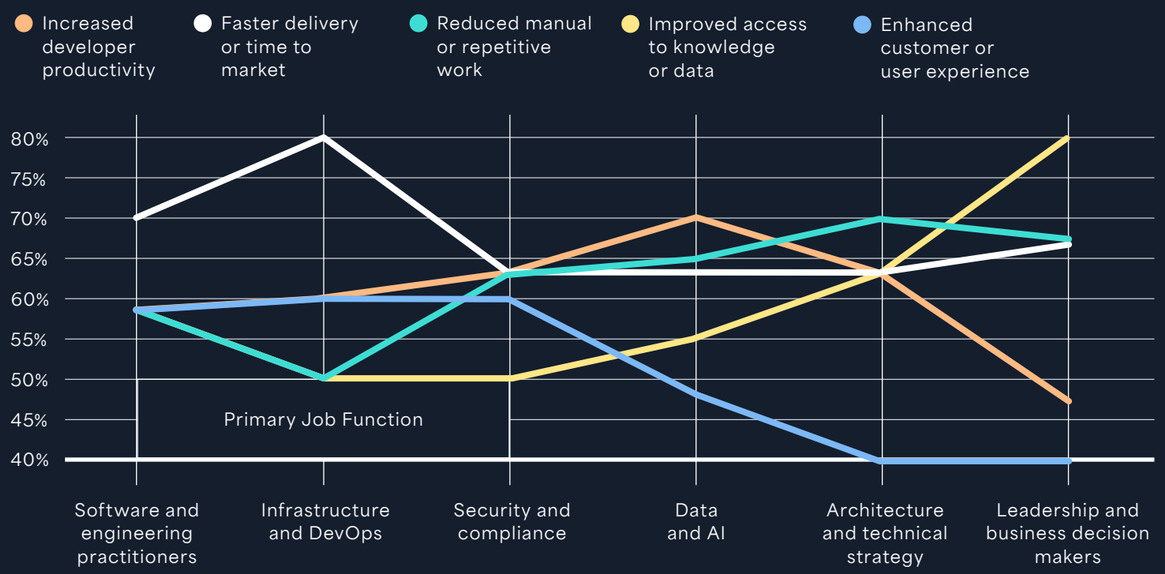
Which of the following LLMOps practices does your organization have in place? [Top 3 use cases]



ROI IS SHOWING UP IN DELIVERY VELOCITY

The most visible gains aren't headline-grabbing transformation. Instead, they are faster release cycles, reduced repetitive work, and higher developer throughput. For many, GenAI is already functioning as an acceleration layer across the SDLC.

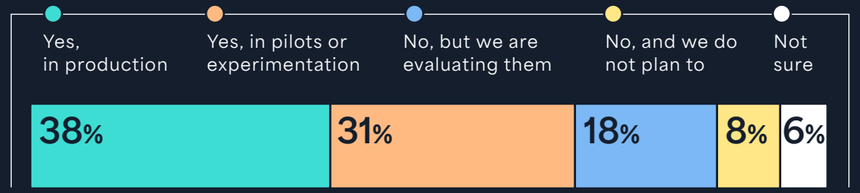
How would you describe your organization's overall GenAI maturity?

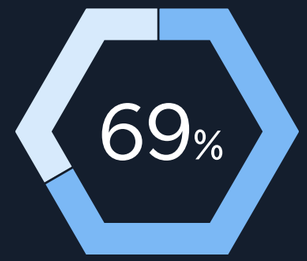


RAG REFLECTS A SHIFT TOWARD TRUST ENGINEERING

Teams are grounding models in internal data to improve reliability and reduce hallucination risk. The lesson is a practical one: Intelligence without context is not useful. Accuracy, traceability, and data control are becoming architectural priorities.

Has your organization implemented retrieval-augmented generation (RAG) or intelligent search?



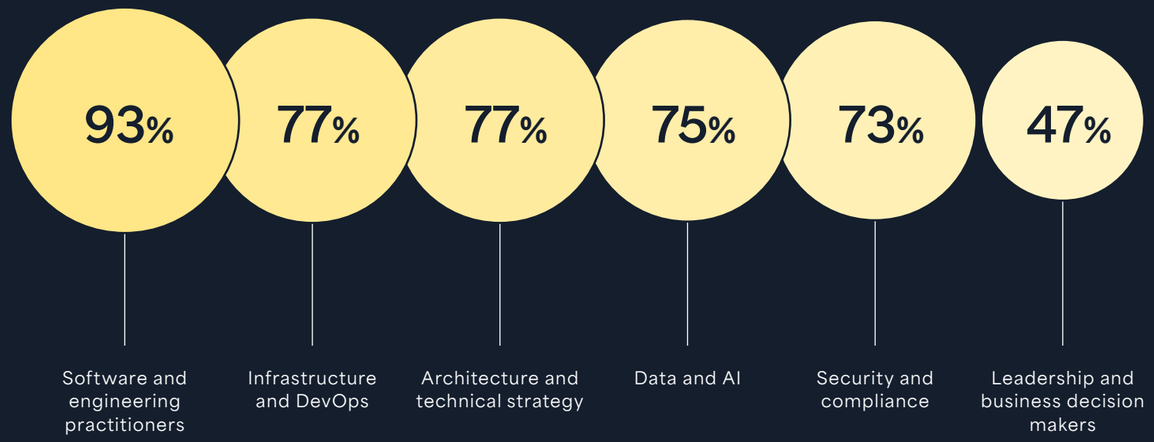


of respondents say their organization has implemented RAG or intelligent search capabilities

THE ENTERPRISE AI STACK IS TAKING SHAPE

Vector search, retrieval pipelines, and governance tooling are solidifying into core infrastructure. The next wave of enterprise systems won't be defined by which model they use, but by how well teams engineer the supporting architecture.

Has your organization used vector databases in GenAI or search-related projects? [Yes, in production or experimentally]



Our research shows that GenAI adoption is no longer the differentiator – engineering maturity is. As AI becomes infrastructure, the winners won't be the teams that experiment fastest, but the ones that operationalize best: building scalable, governed, production-grade systems.

Engineering Trust Into Agentic AI

Why the Data Layer Determines Success

In partnership with

denodo

Kevin Bohan
Director of Product Marketing at Denodo

Agentic AI is moving quickly from prototypes to production systems. Engineers are building agents that can reason over enterprise data, interact with tools, and trigger actions across business workflows. The models are improving. Orchestration frameworks are maturing. Tooling for evaluation, observability, and LLMOps is expanding rapidly. Yet many deployments fail.

The reason is often misunderstood. Most failures are not caused by weak models or flawed prompts. They happen because the underlying data foundation is not designed for autonomous systems. Gartner notes that many organizations **underestimate the expanded data requirements** of AI agents.

Across engineering teams the pattern is clear: Most agent failures are data failures. Agents hallucinate because they rely on stale snapshots. They misclassify entities because definitions differ across systems. They expose sensitive information because governance was added after the prototype worked. These are architectural problems, not prompt problems.

From Batch Intelligence to Live Autonomy

Traditional machine learning systems were built around batch workflows. Data is consolidated into warehouses or lakehouses, models are trained on curated datasets, and results are refreshed periodically. Agentic systems operate very differently. They are expected to query operational and analytical systems in real time, combine structured and unstructured information, make context-dependent decisions, and act within business workflows.

This shift from passive analytics to active autonomy fundamentally changes data requirements. Static snapshots and loosely governed pipelines are not enough. Agents need live, policy-aware, semantically consistent access to distributed data. In practice, an answer might combine information from a lakehouse table, an operational application, and a document repository. Without a reliable way to unify these sources, autonomy becomes fragile.

Separate the Brain From the Data Layer

A clean architectural separation between reasoning and data access is essential. The agent layer should focus on planning, reasoning, memory, and tool orchestration. This is where LLM selection, prompts, and execution strategies live. The data layer should focus on connectivity, abstraction, semantics, security, and auditing. **Logical data management platforms** can provide this capability.

When business rules, schema interpretation, or access policies are embedded directly into prompts or custom tools, the system becomes tightly coupled. Model upgrades become risky. Policy changes require rewriting agents. Observability becomes inconsistent. Separating orchestration from governed data access allows both layers to evolve independently. Models can change without reengineering governance, and policies can be updated without touching agent logic. This separation is often the difference between a proof of concept and a production-ready system.

Live Data for Situational Awareness

If autonomous agents reason over yesterday's data, they make yesterday's decisions. Replication-heavy architectures introduce latency, data drift, and duplicated governance controls. In regulated or operational environments, those gaps create risk.

Replication-heavy architectures introduce latency, data drift, and duplicated governance controls. In regulated or operational environments, those gaps create risk.

A zero-copy approach allows agents to access data where it lives through a logical abstraction layer. This reduces duplication and ensures that decisions reflect the current state of the business. Platforms such as Denodo implement this model through data virtualization, enabling governed real-time access across distributed systems. This approach does not replace lakehouses, warehouses, or streaming platforms. Those systems remain critical for analytics and storage. Instead, a governed access layer complements them by unifying live access across environments. For agentic systems, live access is not simply a performance improvement. It is a requirement.

Semantics Reduce Fragile Prompt Engineering

LLMs excel at language but struggle to interpret enterprise schemas. When agents query raw tables with inconsistent naming conventions and hidden business logic, prompt complexity increases quickly. Developers compensate by adding instructions and examples to teach the model what the data means.

A unified semantic layer shifts that responsibility into the platform. By standardizing business definitions, relationships, and metrics across systems, the semantic layer provides consistent context. Instead of guessing meaning from column names, agents interact with clear business abstractions. This reduces hallucinations caused by schema ambiguity and makes agent behavior more predictable. Semantic tagging also enables consistent classification of sensitive attributes such as PII or financial data. Once tagged, policies can be enforced automatically regardless of where the underlying data resides.

Governed Data Products as Agent Interfaces

Another common failure pattern occurs when agents interact directly with raw data systems. A more resilient approach is exposing curated, governed data products as the interface between agents and enterprise data. A data product encapsulates domain logic, quality rules, defined service expectations, access controls, and metadata and lineage.

Engineering teams must be able to trace what data was accessed, which policies applied, and how the result influenced the agent's decision.

To an agent, the data product appears as a simple tool. To the platform, it acts as a contract that enforces consistency and governance. When these data products are exposed through standards such as Model Context Protocol (MCP), they become reusable tools across different agent frameworks. Denodo supports this approach by exposing governed data products as secure data services. The execution flow becomes predictable: The agent plans a task, invokes a governed tool, retrieves live data with policies enforced, and reasons over consistent results. Credentials remain protected and query logic stays outside the agent code.

Governance and Guardrails by Default

As agents gain more autonomy, the risk surface grows. They access more systems, execute more queries, and trigger more actions. Governance must therefore be enforced at the data layer. Fine-grained access controls, row- and column-level masking, policy enforcement, and full audit logging should apply consistently whether the consumer is a human analyst, a BI tool, or an AI agent.

Observability is equally important. Engineering teams must be able to trace what data was accessed, which policies applied, and how the result influenced the agent's decision. Embedding these guardrails into the data layer ensures that autonomous systems inherit enterprise governance standards automatically. Treating the data layer as a first-class component delivers cleaner agent code, faster production, and improved observability.

A Practical Starting Point

For teams implementing this architecture, Denodo offers a free Developer version of its logical data management platform. Start with a single, well-defined agent use case. Identify the required data, define a governed data product for that domain, apply semantic definitions and policy controls, expose it as a data service, and integrate it into the agent workflow.

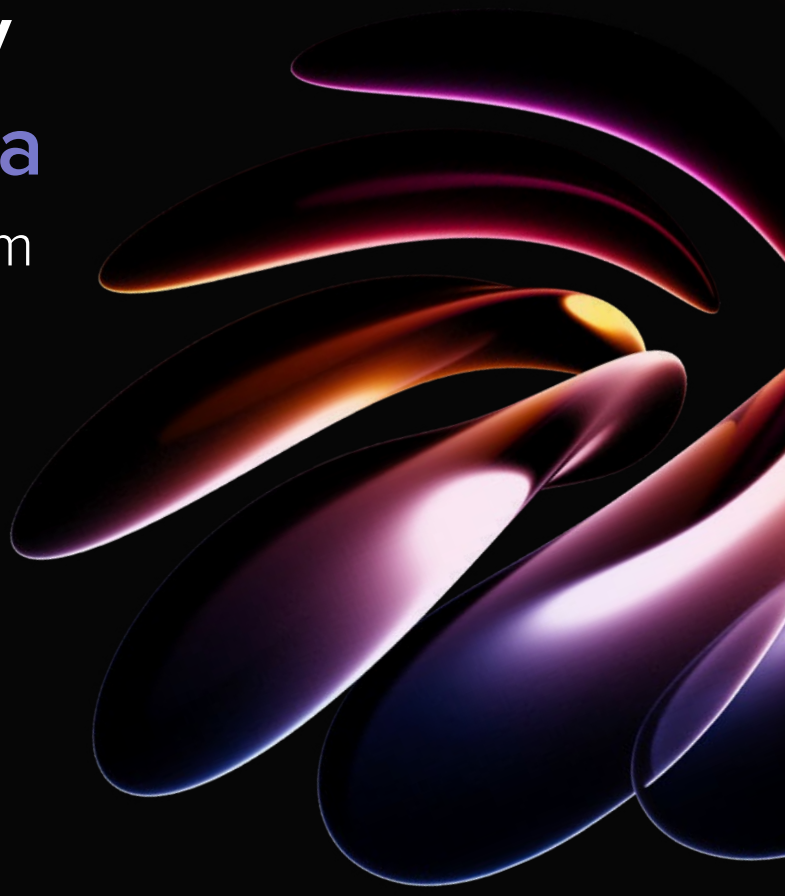
When the data layer provides live access, shared semantics, and built-in governance, autonomous systems become far more reliable. ⬢

denodo

Scan to learn more about the Denodo AI SDK

Your Agent Is Only as Good as Its Data

Engineer trustworthy AI from the data layer up



Agentic AI demands more than powerful models.

It requires:

- Live data access
- Shared business context
- Automated governance
- Secure, policy-aware execution

Denodo is the intelligent data platform built for this reality

Designed for AI Builders

Real-time, zero-copy data access
Unify operational systems, lakehouses, and cloud platforms without adding more pipelines.

Consistent semantics across systems
Give agents shared meaning, not raw tables.

Guardrails by default
Row-level security, masking, and auditing enforced automatically, even for autonomous agents.

Trusted data products for AI tools
Expose governed data as standardized interfaces that integrate cleanly into agents and MCP-compatible frameworks.

Denodo does not replace your data infrastructure. It works alongside it, providing a governed access layer that simplifies agent architecture and reduces production risk.

If you are building AI agents, the data layer is not optional. It is foundational.

Try it yourself

Denodo Developer Edition is free and not a time-limited trial. Download today at denodo.com/try-denodo

3.1

DZone Core Member Project Spotlight

Operationalizing Agentic AI in Enterprises

A Problem-Constraints-Tradeoffs Case

CORE

Dr. Tuhin Chattopadhyay
Professor of AI and Analytics, Area Chair, Analytics Department at Jagdish Sheth School of Management

Dr. Tuhin Chattopadhyay is a highly esteemed figure in the AI and data science fields, commanding immense respect from both the academic and corporate fraternities. He has been recognized as one of India's Top 10 Data Scientists by *Analytics India Magazine,* showcasing his exceptional skills and profound knowledge in the field. Outside of his professorship at JAGSoM, Bengaluru, Tuhin spearheads his own global AI consultancy organization.

tuhinc
tuhinai
tuhin.ai

DZone

Operationalizing Agentic AI in Enterprises

A Problem–Constraints–Tradeoffs Case

Our problem did not show up as a lack of intelligence. It appeared as instability.

In early enterprise deployments of multi-agent systems, instability surfaced in a specific way: Individual agents behaved reasonably in isolation, but the overall system became fragile under real operating conditions. Decisions that looked correct locally produced cascading effects globally.

Signals we saw in practice:

- Latency spikes with no clear triggering change
- Outputs that were valid but inconsistent across runs
- Escalations from downstream teams about trust in automated decisions
- Post-incident ambiguity, with multiple plausible causes but no provable root

This was the point where agentic AI stopped being an architectural idea and more an operational risk. The challenge was not whether agents could act autonomously but whether that autonomy could survive enterprise reality.

Constraints

Experimentation could not come at the cost of reliability. Several constraints shaped every decision that followed, influencing implementation as well as setting the boundary conditions for what safe agent autonomy could mean in an enterprise environment across teams mixed in skill and familiarity with agent architectures.

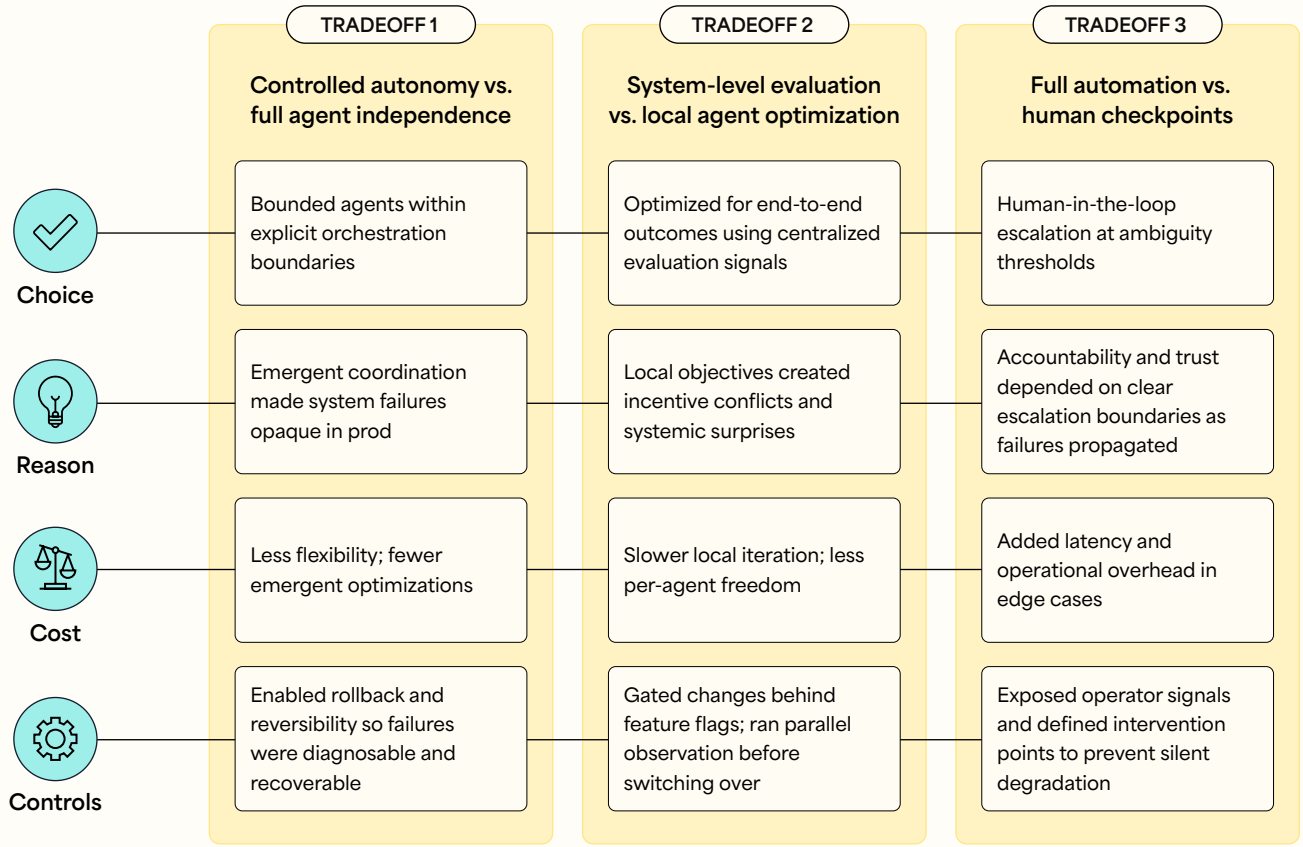
Non-Negotiables	Preferences
- **Governance:** explainable, traceable, reversible decisions - **Reliability:** 24/7 uptime and existing SLAs - **Operability:** mixed-experience support - **Incremental change:** evolve alongside workflows	- Higher autonomy and emergent coordination - Cleaner architecture with fewer guardrails - Faster iteration with less overhead

What we were forced to build:

- Clear rollback paths and access controls
- Early warning signals operators trust
- A simple operating model (debuggable without specialist knowledge)
- Safe rollout mechanics (phased change, parallel observation)

Tradeoffs

Within these boundary conditions, we made three tradeoffs that prioritized predictability and accountability over unconstrained autonomy.



Operational note: We treated new agent behaviors as rollouts, not releases. Changes shipped behind feature flags, were observed in parallel against system-level signals, and were designed to be reversible so we could learn in production without treating production like a lab.

Outcomes

Our outcomes were mixed and instructive. System stability improved and failures became easier to diagnose, though operational overhead increased initially. On-call noise went up before it came down. The biggest surprise was not technical but organizational; adoption lagged until teams understood *how* and *why* decisions were made.

Three lessons learned:

1. Let constraints drive architecture, not ideals
2. Design control surfaces as carefully as intelligence
3. Treat rollout and governance as first-class system components

In enterprises, agentic AI succeeds less by maximizing autonomy and more by engineering accountability.

3.2

Responsible AI Playbook

A Security, Governance, and Compliance Checklist for Safe Adoption

CORE

Pratik Prakash
Principal Solutions Architect
at Capital One

Pratik, an experienced solutions architect and open source advocate, expertly blends hands-on engineering and architecture with multi-cloud and data science expertise. Driving tech simplification and modernization, he specializes in scalable, serverless, and event-driven applications and AI solutions for digital transformation.

scanpratik
pratik-prakash
PratikPrakash_
scanpratik

DZone

Responsible AI Playbook

A Security, Governance, and Compliance Checklist for Safe Adoption

This playbook provides a tactical framework for engineering, security, and product leaders to deploy generative AI responsibly. Safe adoption requires clear boundaries, repeatable controls, and verifiable evidence rather than case-by-case approvals. The following checklist applies to internal productivity tools, customer-facing features, and custom LLM-integrated applications. Teams should use this as a baseline gate before moving any AI use case into production and revisit the criteria quarterly to account for evolving model capabilities and regulatory expectations.

Usage Inventory and Ownership

Before scaling, organizations must identify where AI is currently being used and who is accountable for its behavior. Ensure this step accounts for shadow AI, the unauthorized or unvetted use of AI tools and applications by employees within an organization, bypassing formal procurement and security oversight.

- ○ **Maintain a centralized AI service registry** that logs all sanctioned LLMs, third-party wrappers, and internal experimentations
- ○ **Assign a named business owner** for every AI use case to ensure accountability for output quality and risk
- ○ **Define risk tiers** (e.g., Low, Medium, High) for each application based on data sensitivity and the degree of human oversight
- ○ **Map data inputs and outputs** to visualize exactly where prompts originate and where the resulting generated content is stored
- ○ **Establish a formal exception process** for using non-sanctioned models or experimental features
- ○ **Publish acceptable-use guidance** and mandate developer training to reduce the reliance on shadow AI for production tasks

Model and Data Boundaries

Defining what data can and cannot interact with an LLM is the primary defense against catastrophic data leakage.

- ○ **Classify prohibited data types** (e.g., PII, PHI, internal secrets) that must never be used in prompts or context windows

- ○ **Enforce environment separation** between dev, staging, and prod AI environments to prevent live data from leaking into test models
- ○ **Configure data retention policies** that align with corporate governance, ensuring prompt logs are purged accordingly
- ○ **Require grounding and citations** for all knowledge-based outputs to ensure models pull from verified internal sources
- ○ **Restrict third-party data sharing** by verifying that model providers do not use organizational data for training their base models
- ○ **Verify data residency requirements** to ensure model processing occurs within approved geographic boundaries

Role-Based Access, Privacy, and Controls

AI features should follow the principle of least privilege, ensuring models only access the information necessary for a specific task.

- ○ **Implement RBAC for AI connectors** to ensure models cannot access data sources beyond the user's existing permissions
- ○ **Apply data masking or anonymization** to any sensitive information before it is passed to an external LLM provider
- ○ **Enforce separation of duties** for high-risk changes, such as modifying the primary model provider or altering system prompts
- ○ **Require explicit user notice** for customer-facing features, clearly indicating when an interaction is AI-generated
- ○ **Audit API keys and credentials** used for LLM integrations, rotating them according to standard security protocols
- ○ **Limit connector write permissions** to prevent AI agents from taking unauthorized actions in downstream systems

Auditability and Change Management

To move fast without losing control, teams must be able to reconstruct what the AI did, why it did it, and who authorized the configuration.

- ○ **Log all prompt/response pairs** with associated metadata for forensics and incident investigation
- ○ **Version control all system prompts** and model configurations to prevent silent shifts in application behavior
- ○ **Maintain an immutable audit trail** of who approved specific model changes or routing logic updates

- ○ **Enable traceability of context sources** so that users can verify the specific documents used to generate a response
- ○ **Perform quarterly access reviews** for all users and services with administrative control over AI infrastructure
- ○ **Retain evidence of security testing** and red teaming results for high-risk AI deployments

Quality, Hallucination, and Bias Safeguards

Generative outputs are probabilistic; therefore, teams must implement technical guardrails to monitor and mitigate hallucinations or drift.

- ○ **Define acceptable output criteria** for each use case, distinguishing between creative drafts and factual assertions
- ○ **Implement hallucination filters** or automated fallbacks (e.g., "I don't know") when model confidence scores fall below a set threshold
- ○ **Establish a user reporting path** that allows end users to flag inaccurate, biased, or harmful outputs directly
- ○ **Set automated regression triggers** to re-test critical workflows whenever a model version or prompt is updated
- ○ **Monitor bias and quality signals** through recurring sampling and human-in-the-loop (HITL) reviews
- ○ **Enforce review-before-apply rules** for any AI output that automates high-impact decisions or code execution

The necessary level of control scales with risk. The following table provides a minimum control baseline based on the application's risk tier.

MINIMUM CONTROL BASELINE BASED ON THE APPLICATION'S RISK TIER

Risk Tier	Minimum Controls Required	Review Frequency
Low (internal drafts)	Prompt logging, basic RBAC	Annual
Medium (customer support)	Grounding, PII masking, user reporting	Quarterly
High (financial/medical)	HITL review, red teaming, full audit trail	Monthly

Regulatory Readiness and Third-Party Risk

Compliance is built on documentation, and teams must prepare to prove that their AI stack meets emerging global standards.

- ○ **Document all data flows**, specifically identifying where processing occurs and for what specific business purpose
- ○ **Review vendor security posture** and contractual terms to ensure they meet organizational compliance standards
- ○ **Maintain a current risk summary** for each AI application, detailing known limitations and mitigation strategies
- ○ **Obtain internal sign-offs** from legal and security teams before launching customer-facing or high-stakes AI features
- ○ **Map AI activities to generic frameworks** (e.g., NIST AI RMF, ISO/IEC 42001) to ensure readiness for future audits

Incident Response and Containment

Standard incident response plans rarely account for the unique failures of AI, such as prompt injection or model toxicity.

- ○ **Define AI-specific incident categories,** including data exposure, harmful output generation, and model outages
- ○ **Build a kill switch mechanism** to instantly disable specific AI features without taking the entire application offline
- ○ **Establish a triage workflow** that includes security, engineering, and product owners for rapid escalation
- ○ **Conduct breach simulations** involving prompt injection and tool use failures to test detection capabilities
- ○ **Formalize post-incident reviews** to update system prompts and safety filters based on real-world failures

Closing

Responsible AI is a moving target. Treat this playbook as a living document to be integrated into your SDLC, ensuring that every update is as safe as the initial release. For further guidance, consult the industry-standard resources below.

ADDITIONAL RESOURCES

- → NIST AI Risk Management Framework
- → OWASP Top 10 for LLM Applications
- → ISO/IEC 42001 (Artificial Intelligence Management System)

3.3

AI Maturity Is the New Differentiator

Why Operationalization Matters More Than Model Capability

CORE

Frédéric Jacquet
Technology Evangelist
at AI[4]Human-Nexus

Frédéric Jacquet is an AI and data expert. In 2021, he led a research program on conversational AI, leveraging AI and crowdsourcing, and shared the resulting insights at the 2021 International Databricks AI+Data Summit. He spoke at the AI Paris 2023 conference on Adaptive AI. With over 30 years of experience, he blends cutting-edge technologies with business results while keeping human innovation at the forefront.

aFredNotAfraid
jacquetfred

AI Maturity Is the New Differentiator

Why Operationalization Matters More Than Model Capability

In 2026, the frantic race for the ultimate language model, the one that would be THE most powerful, is becoming irrelevant, if it ever was. As LLM capabilities converge, access to superior raw intelligence is no longer enough to guarantee a competitive edge. The real divide now lies in operationalization, the ability for an organization to transform a fragile prototype into a robust production solution. Achieving this requires a structural shift. It is time to move beyond isolated experiments toward a true stage of systemic maturity, which requires treating AI not as a mere technological curiosity but as a critical production dependency. This scaling relies on a rigorous discipline of reliability, measurement, governance, and engagement, and it requires turning operational maturity into the new strategic pivot.

The Mirage of Model-Based Advantage

Today, the most common mistake is believing that choosing the highest-performing model constitutes a winning bet in itself. This vision overlooks the technical reality of model convergence. Whether proprietary or open source, the performance gaps in standard reasoning tasks are narrowing to the point where generative AI is becoming a sophisticated commodity.

In fact, relying exclusively on a provider's raw intelligence is becoming a delusion. In a production environment, AI must be viewed and treated as a critical dependency, not an isolated project. It is essential to understand that for a company, a model-based competitive advantage loses all value as soon as a competitor updates their API or a new "small language model" surpasses last year's giants, for example. Differentiation no longer stems only from what the model can do; in reality, it also (and now primarily) comes from how the company masters its execution, reliability, and integration into the business value stream.

Symptoms of Operational Immaturity

This phenomenon recalls the early days of big data. Remember, without control over upstream data quality, pipelines propagated silent errors until they rendered management indicators completely useless. Once trust was broken, no one dared to use the reports anymore, leaving the system running empty. Today, the risk is identical since, without rigorous monitoring, models can end up producing hallucinations or subtle biases that degrade user trust without technical teams being alerted.

Added to this is an uncontrolled volatility of costs. Without a true LLMOps approach integrating a FinOps discipline, a simple prompt optimization or an increase in traffic can transform an API bill into a financial nightmare. Finally, immaturity manifests through data

opacity. In particular, the company loses control over systems where one can neither audit the source of information nor guarantee the isolation of sensitive data. These organizations find themselves trapped in a cycle of "perpetual prototyping," where every move to production reveals security or performance flaws that should have been anticipated by a robust architecture.

Five Shifts Toward Operational Maturity

To cross the threshold of industrialization, organizations must execute five strategic shifts. This scaling phase requires trading the sometimes permissive flexibility of a sandbox mode for the rigor of battle-tested industrial standards.

- **Experimentation → ownership:** Maturity begins with clarity. Every AI system must have a defined business owner. It is no longer an IT topic but a business dependency where responsibility for outputs and their impact is explicitly assigned.
- **Subjective validation → systematic measurement:** The era of the "vibe check" is ending. Relying on subjective gut feelings is not viable and should be replaced by automated evaluation pipelines. A mature organization leverages "LLM-as-a-judge" frameworks and rigorous benchmarks to quantify quality and detect regressions before they ever reach the end user.
- **Fragility → a reliability posture:** AI is probabilistic by nature. Maturity consists of accepting this uncertainty by designing architectures capable of managing failures. This involves fallback systems and guardrails to filter hallucinations, as well as proactive latency management.
- **Blind consumption → cost discipline:** Scaling requires a FinOps vision. This means actively arbitrating between the performance of a large model and the efficiency of a smaller specialized model, while implementing quotas and budget visibility per business unit.
- **Monolith → modular architecture:** Mature teams isolate AI behind standardized interfaces. This modular approach allows for replacing one model with another without rewriting the entire application, thus limiting technical debt and excessive vendor lock-in.

AI OPERATIONAL MATURITY DIAGNOSTIC: SYMPTOMS VS. SIGNALS

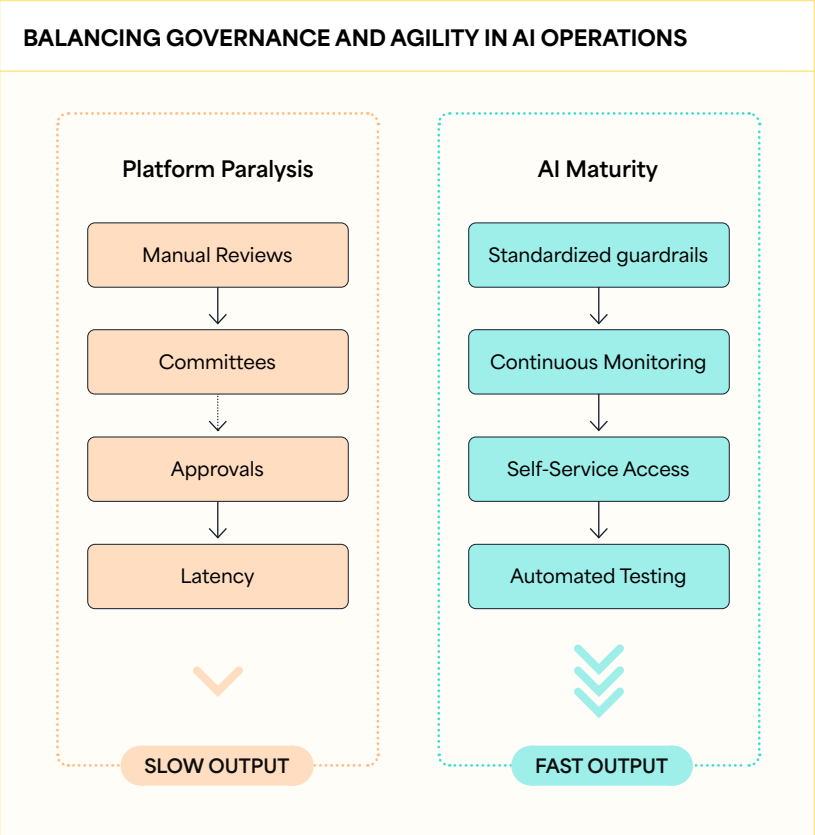
Maturity Dimension	Immaturity Symptom	Mature Signal
Ownership	Shadow AI and ambiguity regarding output responsibility	Defined business owner for every system
Measurement discipline	Subjective manual validation ("vibe check")	Automated benchmarks and drift monitoring
Reliability posture	Fragility in the face of hallucinations or latency	Design for failure modes

Maturity Dimension	Immaturity Symptom	Mature Signal
Cost discipline	Unpredictable invoices disconnected from value	Active arbitration between quality, latency, and cost
Data boundaries	Inconsistent permissions and leakage risks	Access governance and continuous auditability
Architecture	Model changes with unpredictable side effects	Modular architecture limiting cascading failures
Change management	Forced updates causing system breakages	Phased deployments and clear expectations

Use this diagnostic table to identify your current maturity stage and prioritize your operational investments in the short term.

Standardize vs. Localize: Scaling Without Platform Paralysis

To scale without sacrificing speed, operational maturity requires a subtle balance between centralized control and local autonomy. Mature organizations standardize the elements that reduce risk and duplication. This includes elements such as measurement language, security protocols, interface conventions, and production-readiness expectations.



Conversely, everything related to user experience and business expertise is localized. Development teams must remain free to iterate on their workflow UX and on the context strategy specific to their domain. The golden rule is simple: You must standardize what protects the company and localize what preserves relevance and execution speed. Figure 1 illustrates how a mature and standardized architecture unlocks local innovation, whereas rigid governance creates bottlenecks that force the use of shadow AI.

Two Failure Modes That Hinder AI Efficiency

The path to maturity is often hindered by two extremes. The first is perpetual prototyping, when projects never move beyond the pilot stage due to a failure to build the operational muscles necessary for production. The second is platform paralysis. Excessive centralization creates bottlenecks where teams wait for endless approvals for every new prompt. These frictions inevitably push developers toward shadow AI solutions to maintain their pace, ruining any governance efforts.

Take the example of a product team wanting to adjust the "temperature" of the prompt to reduce a customer assistant's verbosity. In an organization constrained by its platform, this minor change requires opening a change ticket, a two-week security review, and approval from a centralized architecture committee. Faced with this bottleneck, the team ends up using a personal OpenAI account and a private API key to bypass the queue. While this shadow AI scenario does not stem from bad intentions, it remains the result of a platform that confused governance with inertia, where teams were forced to choose between strict compliance and pragmatic efficiency.

Conclusion: Pick One Maturity Investment

In the context of massive adoption, operational maturity becomes the sole guarantor of sustainable value creation. So instead of trying to solve everything, adopt an approach that consists of identifying your most glaring symptom of immaturity through the maturity diagnostics table. This is only a starting point, a compass to guide your initial efforts. For instance, start by committing to a single first pillar, such as measurement automation or modularity. Remember that maturity is not a static destination but rather a continuous effort.

In 2026, the difference between a leader and a follower will not be measured by the number of models tested but instead by the robustness of the systems running them and the business value they deliver.

ADDITIONAL RESOURCES

- Artificial Intelligence Risk Management Framework, NIST
- Agents, Large Language Models, and Smart Apps, AI Infrastructure Alliance
- OWASP Top 10 for LLM Applications, OWASP
- "The Illusion of Deep Learning: Why 'Stacking Layers' Is No Longer Enough" by Frédéric Jacquet
- "The Rise of Shadow AI: When Innovation Outpaces Governance" by Frédéric Jacquet

3.4

Shipping GenAI Into an Existing App

How to Integrate AI Features Without Rewriting Your Stack

CORE

Naga Santhosh Reddy Vootukuri
Principal Software Engineering Manager
at Microsoft

As a seasoned professional with 17+ years working at Microsoft and specialized skills in cloud computing and AI, I lead a team of SDEs focused on initiatives in the Azure SQL deployment space, where we emphasize high availability for SQL customers during critical feature rollouts. I review technical books for Apress, Packt, Pearson, and Manning publications; judge hackathons; mentor junior engineers; and write for DZone. I am also a senior IEEE member working on multiple technical committees.

sunnynagavo

DZone

Shipping GenAI Into an Existing App

How to Integrate AI Features Without Rewriting Your Stack

Most production applications already have an architecture, release process, and users who depend on predictable behavior. While adding a generative AI (GenAI) feature into an existing app doesn't change any of that, it introduces uncertainties like non-deterministic outputs, increased latency due to model responses, and failure modes that look like successes but contain hallucinated responses. We tend to treat AI as a special case; however, the goal isn't to build an AI platform but to ship a bounded feature that can improve a specific workflow without creating technical debt across the codebase.

This article provides a repeatable integration pattern that includes how to pick a workflow, define a strict contract, create a clear division of responsibilities between your app and AI logic, design for latency, build a fallback ladder that keeps the app running, and plan a staged rollout for fast iteration and rollback in case of any regressions.

Pick a Workflow That Can Ship

Not all workflows are good candidates for GenAI integration. "Shippable workflows" share three characteristics:

1. Bounded inputs (what goes in)
2. Reviewable output (human can verify and assess results)
3. Verifiable result (if it worked or not)

In these scenarios, even if the AI service goes offline, the user should still be able to complete the workflow using manual steps. For example, consider a support ticketing app where a support representative (rep) writes a summary of the mitigation/resolution that was performed after closing an incident ticket. The AI feature generates a draft reply based on order history and other common resolution patterns, and automatically writes a summary from the conversation transcript. The rep reviews, edits if needed, and submits. If AI is unavailable, the rep can still write the summary manually, exactly as they do today.

This example has the above characteristics:

1. Bounded inputs: order ID, customer messages, transcript in, and summary out
2. Reviewable output: AI agent reviews and edits before submitting
3. Verifiable result: draft accepted, edited, or discarded

If the AI agent fails, it's still low risk because the bad draft takes only a few minutes to fix, it's a slower process, but the workflow still completes, and it's not a customer-facing mistake.

Before committing to a workflow, run it through these filters:

WORKFLOW FILTERS TO CHECK	
Approach	**Techniques**
Input clarity	Can you specify what goes into the model? Is the data structured or easily parsed? Avoid workflows that require unbounded context.
Output shape	Is the generated response structured, or at least constrainable? Can the model return a schema (JSON) rather than prose?
Latency tolerance	Will users wait *X* seconds, or does this need a sub-second response?
Risk tier	What's the blast radius? Does hallucination cause a minor inconvenience or a financial/legal catastrophe?

Define the Feature Contract Before You Touch the Model

The contract is the most important artifact in this integration; it defines what your app sends, what it expects back, and what happens when things go wrong. Before writing a single prompt, define the contract between your app and the AI layer. This allows your frontend and backend teams to build against a scheme while the AI engineer fine-tunes the prompt. Skipping this step leads to common integration failures.

Your contract should define the following:

Inputs:

- **Required fields:** The minimum context the AI needs (e.g., `incident_id` `conversation_transcript`).
- **Optional fields:** Context that improves the summary but isn't mandatory (e.g., `ticket_category`, `customer_tier`, `order_history`, `any previous_resolutions` to consider).
- **Defaults:** When optional fields are missing, apply appropriate default values (e.g., `ticket_category` defaults to `general`, `order_history` defaults to empty `[]`).
- **Max sizes:** Put a ceiling on transcript length; token budgets also matter (e.g., `conversation_transcript` capped at 60,000–70,000 chars, `order_history` limited to last 10 orders).

Outputs:

- **Structured fields:** Define the exact shape (e.g., `draft_summary`, `confidence_score`, `suggested_resolution_steps[]`, `resolution_category`).
- **Draft vs. final:** AI outputs are always drafts so that the rep reviews and edits the summary before submitting. Never let AI post it to the incident ticket automatically.
- **Metadata:** For debugging and auditing, include sources like transcript message IDs, order records, matched resolution patterns, `model_version`, and `processing_time_ms`.

Uncertainty rules:

- **Confidence is below threshold:** Return status as `needs review` with clarifying questions. For example, if the transcript mentions multiple issues, AI asks, "Which resolution should be highlighted?"
- **Context is missing:** Proceed with available data and flag incomplete context as `true`. For example, if order history is unavailable, generate a summary from the transcript (prompt) only.
- **Request is out of scope:** Return status as `unsupported` rather than generate hallucinated responses.

Error categories: Errors are unavoidable. the table below shows different error types, their status codes, and how the end user experience should look.

ERROR TYPES, HTTP STATUS CODES, AND USER EXPERIENCES

Error Type	HTTP Code	Retry	User Experience
Timeout	504	Yes, only once	"AI summary generation taking longer than expected": show fallback UI where rep can write manually
Model unavailable	503	Yes, with exponential back-off policy	"AI summary generation is temporarily unavailable": write summary manually
Rate limited	429	Yes, after a delay	"AI is throttled due to multiple requests": wait for *X* time or write summary manually
No context found	422	No	"Input prompt is too short": add more details to generate a summary

Versioning: Include both `Contract_version` and `Behavior_version` in every generated response. Contract version changes when schema changes, and behavior version changes when input prompt or model selection changes. This prevents silent shifts and makes it easy for your team to trace if the generated summary quality changes due to contract version or behavior version changes.

Contract template:

```
{
  "contract": "incident-resolution-summary",
  "contract_version": "1.3",
  "behavior_version": "2025-06-01",

  "input": {
    "incident_id":           { "type": "string", "required": true },
    "conversation_transcript": { "type": "string", "required": true, "max_
length": 60000 },
    "ticket_category":       { "type": "string", "required": false, "default":
```

Code continues on next page >

```
"general", "enum": ["billing","technical","account","general"] },
    "customer_tier":         { "type": "string", "required": false, "default":
"free", "enum": ["free","pro","enterprise"] },
    "order_history":         { "type": "array",  "required": false, "default":
[], "max_items": 10 },
    "previous_resolutions":  { "type": "array",  "required": false, "default":
[], "max_items": 5 }
  },

  "output": {
    "draft_summary":              { "type": "string",  "description": "Generated
resolution summary -- always a draft, never auto-applied" },
    "confidence_score":           { "type": "enum",    "values":
["high","medium","low"] },
    "status":                     { "type": "enum",    "values":
["complete","needs_review","partial","unsupported"] },
    "review_required":            { "type": "boolean" },
    "resolution_category":        { "type": "string",  "description": "AI-
suggested category (e.g., refund, config_change)" },
    "suggested_resolution_steps":  { "type": "array",   "items": "string" },
    "clarifying_questions":       { "type": "array",   "items": "string",
"description": "Populated when status is needs_review" },
    "source_message_ids":         { "type": "array",   "items": "string" },
    "sources_used":               { "type": "array",   "items": "string",
"description": "KB articles, order records, resolution patterns consulted" },
    "incomplete_context":         { "type": "boolean", "description": "true if
optional inputs were missing or a tool call failed" },
    "model_version":              { "type": "string" },
    "processing_time_ms":         { "type": "integer" }
  },

  "errors": {
    "TIMEOUT":           { "http": 504, "retry": "once",              "fallback":
"retry_then_manual",  "user_message": "Summary taking longer than expected...
Write manually below." },
    "NO_CONTEXT":        { "http": 422, "retry": "no",               "fallback":
"prompt_user",        "user_message": "Transcript too short -- add more detail."
},
    "MODEL_UNAVAILABLE": { "http": 503, "retry": "backoff, max 3",   "fallback":
"queue_or_manual",    "user_message": "AI summaries temporarily unavailable --
write below." },
    "RATE_LIMITED":      { "http": 429, "retry": "after Retry-After", "fallback":
"throttle_or_manual", "user_message": "AI summaries limited right now -- please
write manually." }
  },

  "acceptance": {
    "actions": ["accept", "edit", "discard"],
    "telemetry_event": "summary_draft_outcome",
    "track_per_confidence": true
  }
}
```

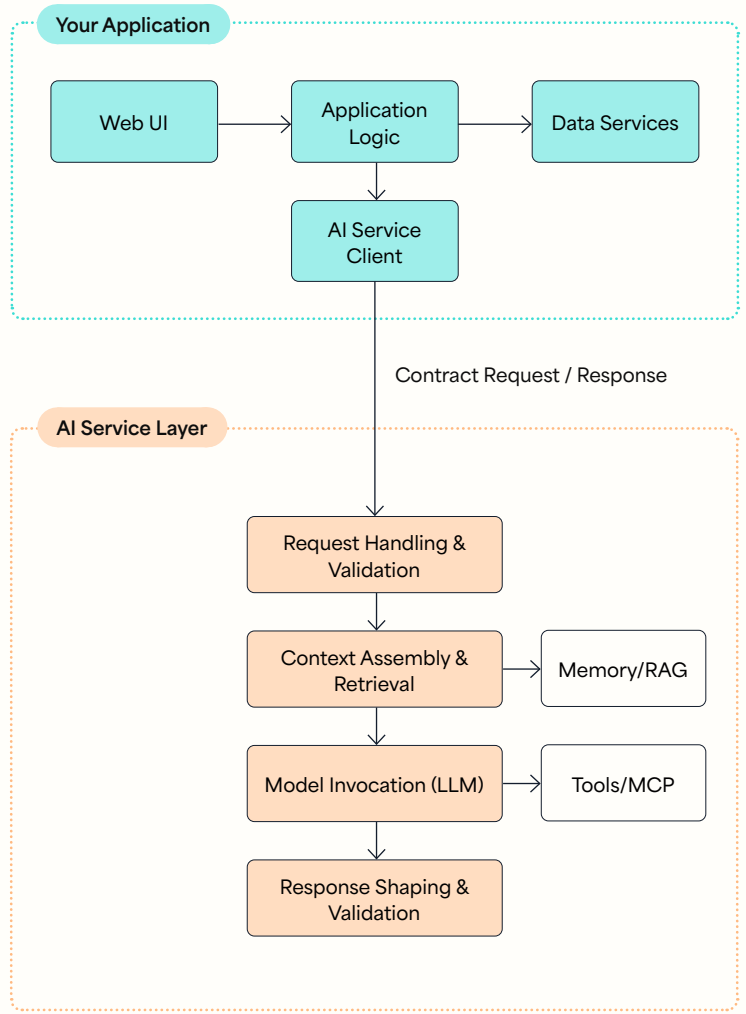
Choose Where AI Logic Lives: In-App vs. a Dedicated AI Service

There are two options for where to keep the AI logic: integrate the AI layer directly into your application code (in-app) or extract it to a dedicated AI service.

When in-app works:

- Single app consumes AI feature
- Small team owns both app and AI logic
- Latency from an extra network call matters
- Minimal operational overhead

AI SERVICE AND APPLICATION BOUNDARIES



When a dedicated AI service is better:

- Multiple components consume similar AI capabilities
- Separate teams own applications versus teams having AI knowledge
- Different release cadences where you want to swap models without redeploying the entire app
- Centralized cost tracking, rate limiting, and audit logging

Once you add one GenAI feature, it's easy to use the same pattern to add multiple features within the same AI service.

Either way, the boundary matters more than the deployment topology. Your app should never construct prompts, parse raw model output, or handle tool orchestration directly. That logic belongs behind the contract interface as shown in Figure 1. General guidance: If it depends on the model, put it into the AI layer, and if it depends on the user, it goes in the application layer.

The **application layer** should never see a raw model response and similarly, the **AI layer** (black box) should never make user experience decisions. This responsibility split allows the app to survive model swaps and prompt rewrites without changing product code.

RESPONSIBILITY CHECK BETWEEN AI AND APP LAYERS		
Responsibility	**App**	**AI**
UX states (loading, error, fallback)	●	○
Permissions and access controls	●	○
Workflow state and persistence	●	○
Presentation and formatting	●	○
Prompt construction and management	○	●
Model selection	○	●
Tool orchestration	○	●
Response shaping to contract schema	○	●
Context gathering	○	●

Add Context and Tools Without Tangling the Product

The most useful AI features usually need context beyond the user's immediate inputs (e.g., knowledge base articles, user history), and they may also invoke tools to look things up or take actions. All context retrieval and tool orchestration stays behind the AI boundary, and your app only sends the inputs defined in the contract. The AI layer decides what else it needs to complete the workflow.

Every tool the AI layer calls should have its own mini-contract (inputs, expected outputs, a failure mode). Tool outputs should use stable identifiers like record IDs, document IDs, and URLs so that the AI layer can pass citations back through the contract's output schema. When a tool fails, the AI layer decides whether to proceed without that context, try an alternate source, or return a degraded response; this decision should never be part of the app code.

Keep Context Optional and Traceable

Design every context source as optional. If the knowledge base is down, the AI layer should still attempt a response from the direct input. However, flag the result as lower confidence, which should reflect the reduced context. If retrieval returns nothing relevant, proceed rather than block.

When the response uses external context, include source metadata in the output to show which documents or records were referenced and what tool calls were made. This makes results auditable, debuggable, and trustworthy to the rep who reviews. A `sources_used` field in your contract output lists this information.

Design for Latency and UX From Day One

As compared to traditional database queries, GenAI calls are usually slow. A typical LLM response takes anywhere between 2–10 seconds. Design your UX around this reality; otherwise, your application will underperform.

STREAMING VS. SYNCHRONOUS VS. ASYNC RESPONSE MODES

Mode	How It Works	UX Pattern
Streaming	Display text upon arrival so user can read immediately	Typewriter effect, as user sees progress instantly
Synchronous	Wait for full response, then display all at once	Loading indicators, and user simply waits
Async	Long operations that process in the background and notify user when complete	High latency, "processing state" and notify user on completion (notifications)

Timeouts are a UX decision, not an infrastructure decision. Define how you want to handle the UX for each threshold value:

- < 2 seconds: show a brief loading indicator; no special handling is needed
- 2–5 seconds: show progress context or text like "drafting summary…"; provide end users the option to cancel
- 5–10 seconds: show a "taking longer than usual…" message; offer an option to fallback or continue to wait
- > 10 seconds: trigger the fallback path automatically; log the timeout accordingly

User controls matter. Let users cancel a pending request or retry on failures. If streaming, let them stop generation early and show partial results when it's safe. If the result is a draft, always let them review and edit it. If the user is not satisfied with the output, provide an option to regenerate accordingly.

Define the UX States Up Front

Map out every state the feature can be in before you build the UI. The table below shows variable states with options for the end user to see and take actions.

VISUAL INDICATORS AND USER ACTIONS

UX State	Visual Indicator	User Actions Available
Loading	Spinner or progress message like "generating draft…"	Cancel
Partial	Streaming text appears incrementally	Cancel, Accept early

UX State	Visual Indicator	User Actions Available
Complete	Full draft shown with confidence score	Accept, Edit, Discard, Regenerate
Fallback	Message displays "AI is unavailable..."	Complete task manually
Error	Display error message and guidance	Retry, Complete task manually

Every state needs a defined exit, or at least one action the user can take. If the user enters a state, they should be able to exit without reloading the page or losing their work.

Build a Fallback Ladder

Fallbacks are not just a single safety net; they are also a ladder that ensures a model failure isn't a dead end, providing users with an option so that they never hit a wall.

- **Clarify:** If the input is ambiguous or incomplete, ask the user for more information before calling the model to prevent wasted calls and low-quality results.
- **Degrade:** If the model returns a low-confidence or partial result, include an appropriate message (e.g., "draft needs review").
- **Alternate path:** If the primary model or tool is unavailable, try a simpler fallback using a smaller model, a template-based fill, or a cached similar result.
- **Handoff:** If nothing works, transition gracefully by showing the user a manual path.

EXAMPLE FALLBACK LADDER

Trigger	User Experience	System Action
Input too short/ ambiguous	"Add more details to generate a summary"	Clarify: prompt user before calling AI layer
Low confidence	"Please review this draft"	Degrade: return with confidence score indicator and flag for manual review
Model timeout	"Taking longer than usual..."	Alternate path: switch to a smaller, faster model before retrying once
API/model down	"AI summary is temporarily unavailable"	Alternate path: return cached template or structure form for manual option
Rate limits /quota exceeded	"High demand right now; your request is queued"	Alternate path or handoff: notify operations team

Make sure there is always guaranteed forward progress, where every fallback ends with a user action, and the workflow can complete without waiting indefinitely.

Cases with low confidence do not mean blocking the workflow or not showing the result. Use draft language (e.g., "Here's a starting point") and visual indicators (e.g., yellow border), and require a manual confirmation from the user before the draft can be applied safely.

Ship Safely With Staged Releases

Don't ship to 100% of end users on day one. Staged releases let you validate at each step in a production-like environment first before blasting the radius to all regions and users. This makes it easy to catch regressions instead of releasing and rolling back at a larger scale.

EXAMPLE ROLLOUT PHASES WITH DURATION AND VALIDATION METRICS

Phase	Audience	Duration	Validation
Internal	Internal teams, "dogfooding"	1-2 weeks	Use it for internal testing, fallback schemes, and latency within thresholds to find obvious breaks
Canary	5-10% of opt-in users	1-2 weeks	Validate at scale and monitor metrics
Expanded	25-40% of users	2-3 weeks	Confirm metrics hold, no latency degradation, no incidents or regressions
Generally Available (GA)	100% of users	Ongoing	Full release, validate models, steady-state metrics

Every phase needs a way to disable the AI feature almost instantly, without requiring a full release or partial deployment (e.g., feature flag that reverts UI to manual path). Test the **kill switch** before you start the incremental rollouts.

Define and establish your **rollback signals** before launch. You need hard thresholds that trigger an immediate rollback, such as fallback rate climbs above a set threshold, increased latency, or escalating model costs. Beyond infrastructure or technical performance, closely look at user interaction metrics as well as whether the user's "discard" rate is significantly higher than the "acceptance" rate.

Feature Telemetry and Pre-Release Checks

Collect telemetry and instrument these metrics from day one, not at the GA phase. Before sign-off, make sure to simulate failures (e.g., model returns garbage JSON, context is empty) and ensure that your telemetry tracks the success-to-failure ratio. Below are the key telemetry signals you should track consistently, starting from your first pre-production rollout:

- **Latency:** p50, p90, and p99 for the full AI service round-trip
- **Timeout rate:** percentage of requests exceeding threshold limits
- **Fallback rate:** how often users hit an issue before fallback
- **User actions:** accept/edit/discard ratios per confidence level

- **Error distribution:** counts by error category from the contract
- **Cost per call:** track token usage or API cost per every request

Initiate pre-release validation before incrementally deploying to the next phase:

- **Contract validation:** Always validate the undesirable path by sending requests with invalid edge-case inputs. Verify whether the error responses match the contract.
- **Error path testing:** Simulate tool failures, timeouts, and model errors. Confirm whether the fallbacks activate correctly.
- **Load testing:** Verify behavior under concurrent requests, particularly rate limit handling. Confirm whether latency holds at the next phase's expected traffic volume.
- **Fallback experience:** Manually trigger each fallback scenario (or disable AI using kill switch). Confirm whether users can complete workflow without any issues.

Security and Compliance Touchpoints

Integrating GenAI into your app introduces specific security and compliance considerations that are worth addressing during early development stages:

- **Data boundaries:** Ensure that the sensitive data (e.g., PII, financials) handled by the AI layer follows your existing app's data classification policies. Don't send restricted data to external model APIs.
- **Access controls:** Make sure to route AI service calls through your app's access control so that permissions are enforced consistently.
- **Audit logging:** Log every AI request and response at the boundary (inputs, outputs, error category, latency) with enough structure for auditing. Also include timestamps, user IDs, and model versions.

The Minimum Viable Integration Pattern

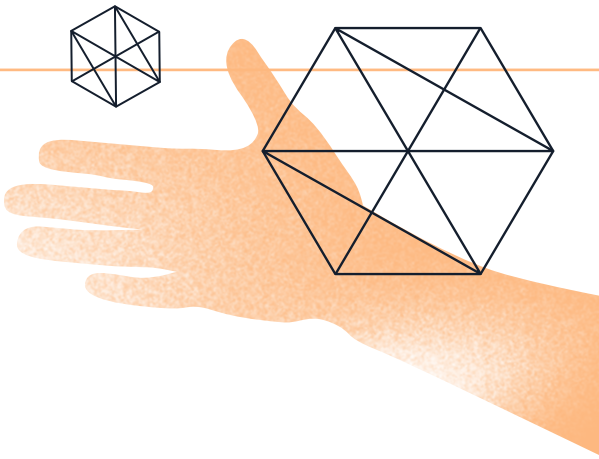
No longer an experiment, shipping a GenAI feature into an existing app is becoming a baseline capability for all teams that want to deliver smarter user experiences without rewriting their entire stack. You will have a repeatable way to ship your next GenAI features faster, safer, and with far fewer surprises by applying the integration pattern outlined in this article. This isn't just an AI platform. It's a service boundary with a contract, the same integration pattern you'd use for any external dependency that is slower and less predictable than your own code. Start with one workflow first, release it and learn from telemetry, and then decide whether the next feature needs the same boundary or a new one.

4.1

Leaders in Tech

Insights From Jeff Hollan, Partner Director of Product at Microsoft

Jeff leads the Agent Platform in Microsoft Foundry, where his team builds the Agent Service, Agent Framework and SDKs, workflows, and developer experience that enable any developer to build, deploy, and manage enterprise AI agents at scale. Previously, he was Director of Product at Snowflake, where he launched Snowflake Intelligence and led the Cortex AI, Snowpark, and Developer Platform efforts. Earlier at Microsoft, he directed Azure's serverless and PaaS business, including Azure Functions, Container Apps, App Services, and Static Web Apps, and is a co-creator of KEDA, an open-sourced CNCF graduated project for event-driven autoscaling in Kubernetes.



Leaders in Tech

Insights From Jeff Hollan, Partner Director of Product at Microsoft

Q In practical engineering terms, what separates an "AI agent" from a chat interface with tools, and what minimum capabilities make something truly agentic?

A The lines definitely blur, and we get excited to call everything an "AI agent" when a new trend emerges. In practical terms, a chat interface is reactive — it responds to a prompt and maybe calls a tool once or twice. What makes something truly agentic is the reasoning capability and ability to work toward a concrete goal. An agent breaks work into steps, reasons through decisions, tracks its progress, and continues until it feels it has adequately met that goal — or knows it should stop and ask for help.

Q What are the most credible enterprise agent use cases you're seeing move from pilot to production, and what's making those succeed?

A The agents that ship are the ones that automate tasks requiring a lot of effort that are repeatable and well-bounded. Triaging support cases, performing first-level analysis, doing research, preparing for meetings, or prepping for sales engagements — these are things that require significant manual labor today and where agents deliver real results. What's consistent across these examples is that the scope is clear, the agent is anchored to trusted data, and the team designs a safe "handoff to human" path from the start. It's not about replacing a role; it's about reliably taking chunks of repetitive work off people's plates.

Q What are the top blockers preventing agents from reaching production at scale, and what are the most effective ways teams are getting past them?

A For an agent to work in the enterprise, it needs access to the right context. That's often where things get hard. Work context, data, and knowledge can be scattered across office docs, knowledge bases, and data lakes. Pulling all of that together on top of a platform that meets security and compliance needs, like private data handling and private connectivity, is a real challenge. It's easy to build a prototype, but doing it in a way that's effective and production-ready is difficult. This is why I encourage teams to leverage platforms that pull these things together for you. Beyond data access, teams that succeed treat evaluation and quality as a priority from the get-go, not a side project. They start with "recommend and draft" before "take action," putting identity, access, and logging in place early. This defines what "good" looks like so they can ship improvements steadily.

> Teams that succeed treat evaluation and quality as a priority from the get-go, not a side project

Q Where do agent deployments fail most often in the real world — and what's the best "first fix" you recommend?

A Without a clear problem outlined for the agent to solve, the agent will lack needed context, return mixed signals, and you end up with confident-sounding output that's hard to trust. The first fix is tightening the definition of the task and what "done" means. Give the agent a concrete objective, make sure it's pulling from the right sources, and normalize it pausing to request missing information rather than guessing.

Q When should teams choose a single generalist agent vs. a multi-agent system vs. an agent plus deterministic workflows — and what decision criteria do you use?

A Start as simple as possible. A single generalist agent is great when you have a human heavily in the loop who can guide and monitor the agent that's acting as an assistant or copilot. Generalist agents fall short when the task is highly consequential and you don't have a human as closely in the loop. That's where purpose-built agents playing defined roles in a multi-agent system, potentially with deterministic workflows, become a better fit. The distinction comes down to how much human oversight is present and how critical the outcomes are.

Q What does "good" evaluation and monitoring look like for agents in production?

A Good evaluation means you can look at real use cases and confirm that as your models evolve, prompts change, and you add new tools, you're not regressing. A healthy set of evaluations, especially around tool selection and answer correctness, is critical. This is core to Microsoft Foundry: allowing teams to continuously measure and observe their agents while staying in control. Speed matters, but most importantly, they need to run safely, predictably, and better each day.

Q If an agent can take actions, what are the non-negotiable controls that enable trust and compliance without slowing delivery?

A Identity must be strictly managed, and you need to be able to track and trace every action so you understand who did what and where. Explicit approval gates and enforceable guardrails are essential, and over time, we'll trust agents to do more on their own. One constraint I see is bounding an agent too tightly, which limits the return. There's a balance to giving agents enough flexibility while maintaining the right guardrails. This is exactly why we built the Foundry control plane: a single place to define and enforce policies so teams can move fast without losing control.

Q Looking ahead 12–24 months, what's the most important shift you expect in agent platforms or developer practice?

A The biggest shift is that both how you build agents and how agents interact with your organization will become far more goal-oriented. Instead of hand-crafting every behavior, you'll be coordinating things like setting the right goals and specifications while leveraging platforms that can do more optimization on their own, and often, improve themselves over time. We're seeing this already in software engineering, where there's far less manual effort and more focus on intent and outcomes. The winning teams will be the ones who can run agents like dependable systems because the models will keep getting better, but the differentiator will be the engineering discipline around them.

5.1

Solutions Directory

Tools to assist with building, deploying, and operating GenAI applications

This directory provides a curated list of widely used developer-focused solutions, limited to free-to-access entry models. Information is sourced from official product documentation and project repositories. Inclusion is based on standardized signals, including industry prominence, adoption and ecosystem traction, maintenance and release cadence, and the availability of verifiable public information.

Solutions Directory

Explore tools used to build and ship generative AI

Product	Type	Purpose	Access Model
2026 PARTNERS			
Apryse Server SDK	SDK	Embed advanced document processing into backend apps	By request
Apryse Mobile SDK	SDK	Introduce fast, precise document viewing to mobile apps	By request
Apryse WebViewer	SDK	Render and process documents within the browser's security	By request
Denodo Platform	Software	Logical data management for real-time enterprise data access and AI	Free tier
Unstructured	Software	Transform documents and PDFs into AI-ready data	Free tier
Vertesia Platform	Software	Transform business workflows with trusted AI agents	Trial period
Accelerate	Library	Run training/inference across CPU/GPU/TPU	Open source
AutoGen	Framework	Build and coordinate multi-agent LLM apps	Open source
BentoML	Software	Package and serve models/LLMs as APIs	Open source
DeepSpeed	Library	Optimize large-scale training and inference	Open source
Diffusers	Library	Build and run diffusion model pipelines	Open source
DSPy	Framework	Program and optimize LLM pipelines	Open source
FAISS	Library	Vector similarity search and indexing	Open source
Gradio	Framework	Create web UIs for ML/LLM demos	Open source

Product	Type	Purpose	Access Model
JAX	Framework	Autodiff and XLA-compiled numerical computing	Open source
KServe	Software	Kubernetes-native model inference serving	Open source
LangChain	Framework	Build LLM apps with tools, agents, and retrieval	Open source
llama.cpp	Software	Local LLM inference on CPU/GPU	Open source
LlamaIndex	Framework	Connect private data to LLMs for RAG	Open source
LiteLLM	Software	Proxy/gateway for multiple LLM provider APIs	Open source
MLflow	Software	Track experiments and manage model lifecycle	Open source
Ollama	Software	Run and manage LLMs on local machines	Open source
ONNX	Spec/ format	Interchange format for exporting ML models	Open spec, open source
ONNX Runtime	Software	Inference engine for ONNX models	Open source
OpenAI Python SDK	SDK/library	Python client for OpenAI-compatible APIs	Open source
PEFT	Library	Parameter-efficient LLM fine-tuning methods	Open source
PyTorch	Framework	Train and deploy deep learning models	Open source
Qdrant	Software	Vector database for embedding search and RAG	Open source, free tier
Ray	Framework	Distributed compute for ML training and serving	Open source
Semantic Kernel	SDK/ framework	Orchestrate LLM agents and workflows	Open source
Streamlit	Framework	Build data and AI web apps in Python	Open source, free
TensorFlow	Framework	Build and deploy deep learning models	Open source
Transformers	Library	Use and fine-tune pretrained models and LLMs	Open source
Triton Inference Server	Software	Serve models at scale across backends	Open source
TRL	Library	Post-train LLMs with RLHF and preference methods	Open source
vLLM	Software	High-throughput LLM serving with batching	Open source

DZone, 3343 Perimeter Hill Dr, Suite 100, Nashville, TN 37211

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community. We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.