Journal of Computer Science and Technology Studies
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

Enterprise Conversational AI Platform Architecture: Orchestration Systems, Retrieval-Augmented Generation, Plugin Ecosystems, Streaming Infrastructure, and Latency Optimization at Enterprise Scale

Prasun Bhattacharyya

Sr. Technical Program Manager, Enterprise AI Platforms, Generative AI, Amazon, USA

Corresponding Author: Prasun Bhattacharyya**, E-mail:** prasunbhattacharyya12@gmail.com

ABSTRACT

Enterprises deploying conversational AI at scale confront a structural question that predates any specific model choice: where should responsibility for deciding what to do with a user's request architecturally reside? As organizations layer chatbots, digital assistants, and internal search tools across dozens of business domains, knowledge remains fragmented across disconnected systems, and routing logic proliferates in ad hoc ways. This paper examines three candidate placements for orchestration responsibility in enterprise conversational AI platforms: embedding reasoning inside the inference layer, augmenting static intent classifiers with machine-learned confidence scoring, and situating a reasoning-based orchestrator at the application layer, coordinating a plugin ecosystem through tool-calling. Drawing on architectural patterns observed across large-scale production deployments serving millions of employees, we propose a five-layer reference architecture spanning user interaction, orchestration, plugin/service, data and retrieval, and governance layers, detail the token-budget economics that constrain plugin scope descriptions within a shared context window, and show how streaming delivery reshapes both governance enforcement and the latency accounting for orchestration overhead. We argue that application-layer orchestration, despite an apparent latency penalty relative to tighter inference-layer coupling, better serves multi-domain enterprise deployments because it preserves extensibility, auditability, and governance separation. Results suggest the architectural placement of reasoning, not the underlying model, is the primary determinant of whether an enterprise conversational AI platform can scale governance and extensibility together.

KEYWORDS

Conversational AI orchestration; retrieval-augmented generation; plugin ecosystems; enterprise governance; latency optimization; multi-agent systems

ARTICLE INFORMATION

ACCEPTED: 20 October 2026 **PUBLISHED:** 20 November 2026 **DOI:** 10.32996/jcsts.2026.8.19

1. Introduction

Enterprise adoption of generative AI shifted from experimentation to infrastructure in the space of roughly two years. By 2024, most organizations surveyed by Stanford's AI Index reported using AI in at least one business function, up from just over half the year before, and the average enterprise reported relying on more than five distinct generative AI applications (Maslej et al., 2025). That growth did not arrive through a single unified assistant. It arrived through a scattering of chatbots, internal search tools, ticketing copilots, and domain-specific agents, each built by a different team, on a different roadmap, with little coordination between them.

The result is a familiar enterprise pathology given new urgency by generative AI: knowledge silos. An employee asking a conversational assistant about expense policy, then benefits enrollment, then a software deployment status, may in practice be routed to three unrelated backend systems, each with its own retrieval pipeline, its own authentication scope, and its own idea of what "helpful" looks like. Consolidating these domains behind a single assistant is now a stated goal for many enterprise platform teams, but consolidation raises an architectural question that model selection alone cannot answer: where should responsibility for deciding what a request needs, and which service should handle it, actually live?

Three answers compete in current practice. One embeds routing and reasoning directly inside the inference layer, coupling model serving with intent resolution so that a single forward pass produces both an answer and a downstream action. A second augments a conventional static intent classifier with machine-learned confidence scoring, keeping a rules-based backbone but softening its

Copyright: © 2026 the Author(s). This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC-BY) 4.0 license (https://creativecommons.org/licenses/by/4.0/). Published by Al-Kindi Centre for Research and Development, London, United Kingdom.

edges with statistical judgment. A third treats orchestration as a distinct application-layer concern: a reasoning engine, separate from any individual model deployment, that inspects a request, consults a registry of available tools and plugins, and issues calls through a structured tool-calling interface. Each placement carries different consequences for latency, for how easily new capabilities can be added, and for how governance teams audit what the system did and why.

This paper asks which of these three placements is best suited to enterprise-scale, multi-domain conversational AI, and argues for the third. Application-layer orchestration, instantiated as a reasoning-based orchestrator sitting above a plugin ecosystem, offers a better balance of extensibility and governance than either inference-layer embedding or augmented static classification, even though it introduces an additional network hop that a tightly coupled inference-layer design avoids. The stakes of this choice compound over time: an architecture that is merely adequate at ten integrated domains can become the binding constraint on a platform's growth once that number reaches fifty or a hundred, long after the original design decision has been forgotten by whoever made it. The remainder of the paper develops this position through a five-layer reference architecture, drawn from patterns observed across production enterprise conversational systems serving millions of users, and defends it against the strongest available counterargument: that tighter coupling at the inference layer should, in principle, reduce round-trip latency enough to outweigh the governance costs of collapsing orchestration into the model layer.

2. Related Work

Static intent classification has served as the backbone of enterprise chatbots since well before the current generation of large language models. A classifier trained on labeled utterances maps a request to one of a fixed set of intents, then hands the request to a corresponding handler. The approach is cheap to run, easy to audit, and predictable in its failure modes, but it degrades quickly outside the intents it was trained on, and adding a new intent typically means retraining rather than simply registering a new capability. Confidence-thresholded variants soften this rigidity by routing low-confidence predictions to fallback handlers or human agents, but the underlying architecture still treats routing as a closed classification problem rather than an open reasoning task.

The transformer architecture that underlies essentially all current large language models reframed what a routing decision could look like (Vaswani et al., 2017). Once a model could attend over long, heterogeneous context and generate structured output, routing could be treated as a generative act rather than a lookup. Retrieval-augmented generation extended this further by pairing a parametric language model with a non-parametric retrieval index, allowing the model to ground its output in documents rather than only in what it had memorized during training. This combination outperformed both purely parametric sequence-to-sequence models and task-specific retrieve-and-extract pipelines on open-domain question answering (Lewis et al., 2020), and the pattern has since become closer to a default than a novelty: a majority of enterprise RAG implementations now rely on GPT-based models paired with standard retrieval frameworks, according to a recent systematic review (Akbulut, 2026). Retrieval is not uniformly beneficial, however. Adaptive strategies that decide when to retrieve, rather than retrieving indiscriminately on every turn, have been shown to improve performance on short-form open-domain questions (Zhang et al., 2024).

A parallel line of work treated the language model itself as a reasoning agent capable of invoking external tools rather than only generating text. ReAct interleaved reasoning traces with actions, and this interleaving outperformed imitation- and reinforcement-learning baselines by a wide margin on interactive decision-making benchmarks, according to Yao et al. (2023). Toolformer pushed further by having models learn, largely on their own, when and how to call external APIs during generation (Schick et al., 2023). These tool-calling patterns became the substrate for multi-agent frameworks: two or more language-model agents could coordinate through structured role-play to complete tasks neither could complete alone, as CAMEL demonstrated (Li et al., 2023), and AutoGen generalized this into a configurable framework for multi-agent conversation across a wider range of enterprise-relevant tasks (Wu et al., 2024). A related but distinct problem, namely how an agent retains state across a conversation that exceeds any single context window, was addressed by Packer et al. (2024), who proposed an operating-system-like memory hierarchy in MemGPT. The Model Context Protocol has more recently begun to standardize how tools describe themselves to a calling model; a taxonomy of the protocol decomposed its lifecycle into sixteen distinct activities across four phases while cataloguing sixteen associated threat scenarios (Hou et al., 2026).

What this body of work has not settled, and what a recent survey of multi-agent systems makes explicit in calling for further architectural study (Guo et al., 2024), is where orchestration responsibility should sit relative to model serving when an enterprise is not building one agent but coordinating dozens of narrow, governed capabilities behind a single conversational surface. That gap is the starting point for the architecture developed in the next section.

3. Platform Architecture

Across the production enterprise conversational AI systems we have observed and helped build, a common structure emerges once a platform crosses from single-domain chatbot to multi-domain assistant. We organize this structure into five layers: user interaction, orchestration, plugin/service, data and retrieval, and governance.

3.1 A Five-Layer Reference Architecture

Interaction with the platform begins at a surface the employee actually sees: a chat window, a voice interface, a command embedded in another application. Its job is narrow by design. This user interaction layer captures input, renders output, and

preserves just enough session state to support a turn-based conversation, deferring almost every substantive decision to the layer beneath it.

Sitting immediately below is the orchestration layer, where we locate the reasoning-based orchestrator (RBO) described in section 3.2. This layer receives a normalized request from the interaction layer, decides which plugins or services are relevant, sequences calls to them, and assembles a response. Its architecture is deliberately independent of any single model deployment: the orchestrator can be backed by different underlying models over time, or even route different requests to different models, without the interaction layer or the plugin layer needing to change.

Individual capabilities live one level further down, in a plugin/service layer that hosts the tools the orchestrator can invoke: a policy lookup tool, a ticketing system connector, a benefits calculator, a code-search utility. Each plugin exposes a scope description, a set of callable functions, and an expected input/output contract, discussed further in section 3.3.

Grounding comes from a fourth layer, data and retrieval, which supplies context rather than commands. Rather than a single monolithic index, enterprise deployments typically maintain multiple retrieval sources, scoped by domain and by access permission, that plugins or the orchestrator itself can query. This is where retrieval-augmented generation patterns (Lewis et al., 2020) are operationalized at the platform level rather than within any single model call.

Cutting across all four of these, rather than sitting cleanly above or below them, is governance. This layer logs orchestration decisions, enforces access control on which plugins a given user or request can reach, and provides the audit trail that compliance and security teams require. We return to the governance implications of this layered separation in section 4.

Table 1: Architecture layer responsibilities

Layer	Primary responsibility	Representative components	Governance touchpoint
User interaction	Capture input, render output, hold turn-level session state	Chat window, voice interface, embedded command surface	Input logging, session boundary enforcement
Orchestration	Reason over requests, select and sequence plugin calls	Reasoning-based orchestrator (RBO), plugin registry	Decision logging, model-swap isolation
Plugin/service	Execute narrow, owned capabilities	Policy lookup, ticketing connector, benefits calculator	Per-plugin access scoping
Data and retrieval	Supply grounded context to plugins and orchestrator	Domain-scoped retrieval indexes	Permission-scoped retrieval, data residency controls
Governance	Cut across all layers to enforce access control and auditability	Audit log, access-control policy engine	Cross-layer decision trail for compliance review

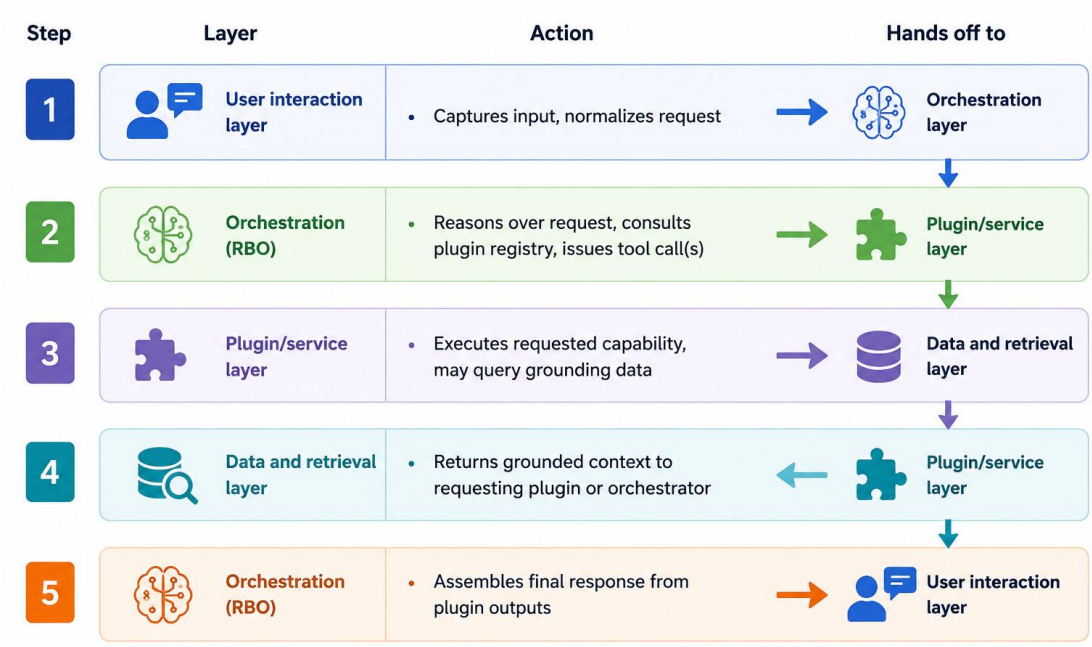


Figure 1: Request flow through the five-layer reference architecture

The governance layer logs and enforces access control continuously across steps 1-5 rather than occupying a discrete step in the sequence.

3.2 Reasoning-Based Orchestration and Tool-Calling

The reasoning-based orchestrator sits at the center of this architecture and performs a function analogous to what ReAct-style interleaving does inside a single agent loop (Yao et al., 2023), except at the scale of an entire enterprise plugin ecosystem rather than one task. On receiving a request, the orchestrator does not classify it into a fixed intent. It reasons over the request in natural language, consults a registry of plugin scope descriptions, and issues one or more structured tool calls, in the pattern established by Toolformer-style tool-use training (Schick et al., 2023). Where a request spans domains, for instance a question that requires both a policy lookup and a personalized calculation, the orchestrator can sequence multiple plugin calls within a single turn, coordinating them in a way that resembles the multi-agent conversational patterns described by Wu et al. (2024), though here the "agents" are narrow deterministic services rather than additional language models. This distinction carries governance weight as well as engineering convenience, since a deterministic service is far easier to constrain, log, and roll back than an additional autonomous reasoning agent would be.

Extensibility follows directly from this design. Adding a new capability to the platform means registering a new plugin and its scope description, not retraining a classifier or rewriting a monolithic prompt. In our experience operating platforms of this kind, this is the single largest driver of how quickly a platform team can respond to new business requirements once dozens of domains are already live.

3.3 Plugin Design and the Token-Budget Constraint

A reasoning-based orchestrator can only consider plugins it can see, and it can only see what fits in its context window alongside the conversation history, retrieved context, and its own reasoning trace. This creates a real economic constraint on plugin design, one that becomes visible only once a platform hosts more than a handful of plugins. A scope description budget of roughly 500 tokens per plugin has proven a workable ceiling in practice: descriptions much longer than this crowd out conversation history and retrieved content, while descriptions much shorter tend to under-specify what the plugin does, leading the orchestrator to invoke it incorrectly or fail to invoke it when it should. In practice, this ceiling tends to surface first as an operational symptom rather than a deliberate design choice: response quality degrades gradually as more plugins are registered, well before anyone traces the cause back to context-window arithmetic.

This constraint is not a minor implementation detail. It is a design discipline that shapes the entire plugin ecosystem. A plugin's scope description must state, within that budget, what the plugin does, when it should be called, what inputs it needs, and what its outputs mean, without exhaustively documenting every parameter or edge case. Standardization efforts around how tools describe themselves to a calling model, such as the Model Context Protocol, respond to exactly this pressure (Hou et al., 2026), and platform teams that fail to enforce a token discipline of this kind tend to discover it only after registering enough plugins that the orchestrator's effective context budget for conversation history has quietly collapsed. Techniques for reusing or caching portions of a prompt across calls, such as modular attention reuse, can partially relax this constraint by reducing the marginal cost of including a plugin description in every call, but they do not eliminate the underlying trade-off between plugin catalog size and available context for everything else (Gim et al., 2024).

Deriving the token budget. The figure is a working ceiling rather than a measured optimum, and the reasoning behind it is worth making explicit so that readers operating under different constraints can re-derive it for their own deployments. The orchestrator's context window must simultaneously hold five competing claims: the plugin registry, whose size grows as the product of catalog size and per-plugin scope-description length; enough conversation history to remain coherent across turns; retrieved context supplied by the data and retrieval layer; the framing that establishes the orchestrator's role and output contract; and headroom for its own reasoning trace and emitted tool calls. Only the first of these grows with platform adoption, which makes it the term that must be bounded by policy rather than left to accumulate. Fixing a target catalog size and reserving the majority of the window for conversation history, retrieved context, and reasoning headroom leaves a registry envelope from which a per-plugin ceiling follows by division. Across the deployments informing this paper, that arithmetic consistently landed in the low hundreds of tokens, and 500 was adopted as a round working ceiling at the upper end of that range. The figure was then checked against observed behavior rather than derived and left alone. Where individual scope descriptions were allowed to exceed it materially, the first symptom was not a hard failure but a gradual loss of multi-turn coherence as older conversational context was displaced, followed by degraded grounding as retrieved passages competed for the same space. Where descriptions were compressed well below it, the failure inverted: the orchestrator began selecting plugins incorrectly or omitting relevant ones, because invocation conditions and output semantics had been abbreviated past the point of usefulness. We therefore offer 500 tokens as a calibrated design discipline for platforms in this class, not as a value derived from controlled experimentation, and we expect the specific number to move with context-window size, catalog size, and the retrieval intensity of a given deployment.

Table 2: Plugin design constraint summary (approximate 500-token scope-description budget)

Description element	Approx. token share	Purpose
Capability summary (what the plugin does)	30-35%	Lets the orchestrator match the plugin to relevant requests
Invocation conditions (when to call it)	20-25%	Reduces incorrect or missed invocations
Required inputs and parameters	15-20%	Specifies what the orchestrator must supply
Output semantics	15-20%	Tells the orchestrator how to interpret and use the result
Reserved margin	~10%	Buffers against prompt-formatting overhead and future edits

3.4 Dual-Context Memory

Enterprise conversational platforms also need to remember things across turns and, in many cases, across sessions. We use a dual-context memory model that separates two kinds of state. Session memory holds the immediate conversational context, what was asked, what was retrieved, what plugins were called, valid for the duration of a single interaction and discarded, or summarized and archived, once it ends. A user-defined personalization layer, by contrast, persists across sessions and holds information the user has explicitly established as durable, such as a preferred communication style, a home department, or standing preferences for how certain requests should be handled.

This separation matters architecturally because the two kinds of memory carry different governance requirements and different consistency needs. Session memory can be relatively ephemeral and locally scoped; the personalization layer needs stronger consistency guarantees and clearer user control over what is retained. A closely related problem arises within a single agent, where conversation state can grow past what any one context window holds; an operating-system-like paging mechanism between working context and long-term storage was one proposed answer to that problem (Packer et al., 2024). The dual-context split described here can be understood as a domain-specific instance of that same underlying idea, extended to a setting where personalization data must also satisfy enterprise data-retention and privacy policy, and where a user, not just an administrator, can inspect and revise what has been remembered about them.

3.5 Streaming Delivery and the Layer Boundary

Nothing in the layered separation described so far requires a response to be assembled before it is delivered, and in production deployments it is not. The response is emitted progressively, token by token, from the point at which the orchestrator begins synthesising plugin outputs. We treat this as an architectural commitment rather than a transport detail, because it constrains what each layer above the plugin/service layer is permitted to do: no layer may buffer a complete response in order to inspect it, since by the time a complete response exists the user has already read most of it.

That constraint falls most heavily on the governance layer. Where a non-streaming architecture can screen a finished response against access-control and data-handling policy before releasing it, a streaming architecture has no such checkpoint. Enforcement must therefore happen upstream of emission rather than downstream of generation, which is why the permission-scoped retrieval and per-plugin access scoping described in section 3.1 are load-bearing rather than defensive redundancy: they are the only points at which the platform can still decide what a user is entitled to see. The same constraint shapes orchestration sequencing. When the orchestrator invokes several plugins within a turn, the sequence must be arranged so that at least one produces emittable content early; otherwise the user waits on the slowest branch with nothing on screen, and the parallelism the architecture makes available is spent without being felt.

Streaming also changes how the latency cost of application-layer orchestration should be accounted for, which bears directly on the objection taken up in section 4. The additional hop between orchestrator and plugin is paid once, before the first token is emitted, and is not repeated across the remainder of the stream. Its user-visible cost is therefore bounded by first-byte latency rather than by total response time, and its proportional weight falls as responses lengthen. Total-latency accounting, which is what the inference-layer coupling argument implicitly assumes, overstates the penalty for exactly the multi-domain queries the architecture exists to serve. This is an argument about where the cost lands, not a claim that it disappears; the techniques for reducing the underlying latency itself, including prompt caching and reuse of processed prompt segments across calls (Gim et al., 2024), operate independently of where orchestration sits.

3.6 Voice and Multi-Modal Access

Nothing in this architecture depends on the interaction layer being text-based. Voice input, once transcribed, and multi-modal input, once encoded into a form the orchestrator can reason over, enter the same orchestration layer through the same tool-calling interface as typed text. What changes at the interaction layer is latency tolerance: voice interactions are typically less forgiving of round-trip delay than a typed chat window, which raises the stakes of the latency question addressed directly in section 4. This has consequences beyond raw user experience, since voice deployments often demand the tightest achievable latency

budget precisely where the governance layer's audit and access-control work still has to happen, just without becoming visible to the person speaking.

4. Architectural Analysis and Discussion

The strongest case against application-layer orchestration is a latency argument, and it deserves to be taken seriously rather than dismissed. Embedding routing and reasoning inside the inference layer collapses what would otherwise be two network round trips, one to the orchestrator and one to the plugin or model it selects, into a single forward pass. Recent advances in inference serving make this coupling increasingly attractive on paper. PagedAttention-based serving achieves substantially higher throughput than earlier serving systems at comparable latency (Kwon et al., 2023), building on distributed serving architectures like Orca that were themselves designed to reduce per-request overhead (Yu et al., 2022). Speculative decoding can accelerate autoregressive generation by a factor of two to three without any change to model architecture (Leviathan et al., 2023), and semantic caching approaches have reported substantial improvements in cache hit ratio and token savings relative to earlier caching schemes (Li et al., 2024). A recent survey of efficient LLM serving suggests, taken together, that the inference layer has become the primary site of latency engineering in production language-model systems (Miao et al., 2025). Set against a real-time latency budget measured in a few hundred milliseconds, these are not small gains, and they explain why serving-layer engineering absorbs a disproportionate share of platform investment in many organizations. If the inference layer is already where the serious latency optimization happens, the argument goes, why not also do routing there?

This argument, though not wrong on its own terms, appears not to hold once a platform moves from a small number of well-integrated capabilities to the dozens of loosely coupled, independently owned domains typical of a large enterprise. Inference-layer embedding assumes that the reasoning needed to route a request is stable enough, and small enough, to bake into the serving path for a given model or model family. That assumption may hold for a platform with five tightly scoped domains. It tends not to hold once new plugins are added weekly, ownership of individual domains is distributed across separate engineering teams, and the underlying model itself may be swapped, upgraded, or run in parallel with alternative models for evaluation. Under those conditions, coupling orchestration logic to inference means that every new capability, every ownership change, and every model migration touches the same latency-critical serving path the counterargument is trying to protect.

Augmented static classifiers fail for a related but distinct reason. Softening a rules-based backbone with confidence scoring keeps the classifier's fundamental limitation intact: it can only route to intents it already knows about, and each new intent still requires a labeling and retraining cycle rather than a plugin registration. This makes the augmented-classifier approach slower to extend than either of the reasoning-based alternatives, even when it matches or beats inference-layer embedding on raw latency for the intents it already covers.

There is also a measurement question folded into the counterargument that is worth separating out. The serving-layer results cited above are reported as throughput at matched latency or as end-to-end generation speedups, both of which are total-response-time quantities. Streaming decouples the metric that governs perceived responsiveness from that total, as section 3.5 sets out: the orchestrator's additional hop is a fixed cost incurred once before the first token and never again within the turn. Compared on first-byte latency rather than on total response time, the gap between application-layer orchestration and inference-layer coupling narrows to a single constant, and narrows further as responses lengthen. We do not claim the constant is negligible, and its size will vary with deployment topology; the point is that the counterargument's strength depends on a metric choice it does not make explicit, and that the choice favours the coupled design for reasons unrelated to the routing decision itself.

Three properties appear to matter more at enterprise scale than shaving milliseconds off a single round trip, and application-layer orchestration accepts a modest latency cost, an additional hop between the orchestrator and the services it calls, in exchange for them. First, it decouples plugin development from model deployment, so a team can ship a new capability without coordinating a serving-path change, and a platform team can migrate or evaluate models, including using an independent model as a judge of orchestration quality in the way Zheng et al. (2023) describe for general LLM evaluation, without touching plugin code. Second, it produces a natural audit boundary: every orchestration decision is a discrete, loggable event, distinct from the token-level computation inside any individual model call, which aligns with risk-management frameworks that ask organizations to document how an AI system's decisions can be traced and reviewed (Tabassi, 2023). Third, it supports the kind of access control that regulated enterprises increasingly need to demonstrate, since the orchestration layer is a natural place to enforce which plugins a given user, role, or request context is permitted to reach, consistent with recent governance-framework proposals for enterprise AI operating under regulatory scrutiny (Otunlape, 2026). Taken together, these three properties turn orchestration from a routing mechanism into a control point through which the rest of the platform can be governed.

None of this makes the counterargument disappear. Where an enterprise deployment is narrow, single-domain, and latency-critical in a way end users can perceive directly, for instance a real-time voice interface handling a small, stable set of well-understood requests, inference-layer coupling may still be the more defensible choice. It is not yet clear whether the token-level latency savings observed in narrow serving benchmarks would translate into a perceptible user-facing difference once network and retrieval latency elsewhere in the pipeline are accounted for. Our position is scoped deliberately to the multi-domain, multi-owner, governance-sensitive case, which describes the large majority of enterprise conversational AI platforms operating today. Within that scope, the evidence favors keeping reasoning at the application layer even at some cost to raw round-trip speed. A platform

serving a handful of stable, high-volume intents may reasonably choose differently, and that difference in choice, rather than any flaw in the underlying models, is what separates the three architectural placements considered here.

Table 3: Comparative analysis of orchestration approaches

Dimension	Inference-layer embedding	Augmented static classifier	Application-layer orchestration (RBO)
Extensibility	Requires serving-path changes to add capability	Requires relabeling and retraining for new intents	Requires only a new plugin registration
Coupling to model deployment	Tight; routing logic lives inside the served model	Loose but rigid; classifier is separate but static	Loose and flexible; orchestrator is model-independent
Latency profile	Lowest round-trip latency in principle	Low latency for known intents	One additional network hop versus inference-layer embedding
Governance and auditability	Difficult to isolate routing decisions from generation	Simple to audit but limited in scope	Discrete, loggable orchestration events
Failure mode under scale	Serving-path bottleneck as domains multiply	Coverage gaps outside trained intents	Context-window pressure from plugin catalog growth
Best-fit scenario	Narrow, single-domain, latency-critical deployments	Stable, well-defined, low-change intent sets	Multi-domain, multi-owner, governance-sensitive platforms

Table 4: Latency and governance trade-off summary

Architectural placement	Typical latency profile	Governance and auditability	Extensibility	Best-fit scenario
Inference-layer embedding	Lowest (single forward pass)	Weak; routing entangled with generation	Low; tied to serving-path changes	Narrow, stable, latency-critical single domain
Augmented static classifier	Low for known intents	Moderate; classifier decisions are simple to log	Low; requires retraining per new intent	Stable, low-change intent sets
Application-layer orchestration (RBO)	Moderate (one added hop)	Strong; discrete, loggable decisions	High; plugin registration only	Multi-domain, multi-owner, governance-sensitive platforms

5. Conclusion

This paper set out to answer a placement question rather than a modeling question: not which model an enterprise conversational AI platform should run, but where responsibility for deciding what to do with a request should architecturally sit. Application-layer orchestration, implemented as a reasoning-based orchestrator coordinating a plugin ecosystem through tool-calling, better serves multi-domain enterprise deployments than either inference-layer embedding or augmented static classification, notwithstanding a real and acknowledged latency cost relative to tighter coupling. The contribution offered here is less a new algorithm than a synthesis: a five-layer reference architecture that draws together two operational constraints, a roughly 500-token scope-description budget per plugin and a dual-context memory model separating ephemeral session state from durable personalization, into a single reusable framework, grounded in patterns observed across production platforms serving millions of users rather than in laboratory benchmarks.

Some boundaries around this argument are worth stating plainly. The position defended here concerns architectural placement under multi-domain, multi-owner conditions; it does not claim that any particular reasoning model, prompting strategy, or plugin protocol is superior, and it does not resolve the narrower question of when inference-layer coupling might still be preferable for constrained, latency-sensitive single-domain deployments. The token-budget figures and architectural patterns described here reflect what has proven workable across the systems observed rather than a value derived from controlled experimentation, and readers building smaller or differently governed platforms should expect the specific numbers to shift even where the underlying trade-offs do not.

Future orchestration research might usefully examine how governance and auditability requirements should shape multi-agent coordination protocols as they mature beyond the two-agent and small-team settings studied so far, how token-budget constraints on tool descriptions interact with emerging standards for tool interoperability, and whether hybrid placements, in which narrow, latency-critical sub-flows are permitted to bypass the orchestrator under tightly scoped conditions, can preserve most of the governance benefit argued for here while recovering some of the latency the pure application-layer design gives up.

6. Ethics Statement

6.1 Statement of Human and Animal Rights

This study did not involve human participants, animal subjects, or identifiable personal data requiring ethics board review. The work is an architectural and analytical treatment of enterprise AI systems.

6.2 Informed Consent

Not applicable.

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers.

References

[1] Maslej, N., Fattorini, L., Perrault, R., Gil, Y., Parli, V., et al. (2025). The AI Index 2025 annual report. Stanford Institute for Human-Centered AI.

[2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*, 5998-6010. https://doi.org/10.48550/arXiv.1706.03762

[3] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, *33*, 9459-9474.

[4] Akbulut, A. (2026). Retrieval-augmented generation (RAG) and large language models (LLMs) for enterprise knowledge management and document automation: A systematic literature review. *Applied Sciences*, *16*(1), 368. https://doi.org/10.3390/app16010368

[5] Zhang, Z., Fang, M., & Chen, L. (2024). *RetrievalQA: Assessing adaptive retrieval-augmented generation for short-form open-domain question answering*. Findings of the Association for Computational Linguistics: ACL 2024, 6963-6975.

[6] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). *ReAct: Synergizing reasoning and acting in language models*. International Conference on Learning Representations. https://doi.org/10.48550/arXiv.2210.03629

[7] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, *36*, 68539-68551.

[8] Li, G., Hammoud, H.A.A.K., Itani, H., Khizbullin, D., & Ghanem, B. (2023). CAMEL: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, *36*, 51991-52008.

[9] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., Liu, J., Awadallah, A.H., White, R.W., Burger, D., & Wang, C. (2024). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. First Conference on Language Modeling.

[10] Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S.G., Stoica, I., & Gonzalez, J.E. (2024). MemGPT: Towards LLMs as operating systems. First Conference on Language Modeling.

[11] Hou, X., Zhao, Y., Wang, S., & Wang, H. (2026). *Model Context Protocol (MCP): Landscape, security threats, and future research directions*. ACM Transactions on Software Engineering and Methodology. https://doi.org/10.1145/3796519

[12] Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N.V., Wiest, O., & Zhang, X. (2024). Large language model based multi-agents: A survey of progress and challenges. Proceedings of the 33rd International Joint Conference on Artificial Intelligence, Survey Track, 8048-8057.

[13] Gim, I., Chen, G., Lee, S., Sarda, N., Khandelwal, A., & Zhong, L. (2024). Prompt cache: Modular attention reuse for low-latency inference. Proceedings of the 7th Conference on Machine Learning and Systems.

[14] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C.H., Gonzalez, J., Zhang, H., & Stoica, I. (2023). *Efficient memory management for large language model serving with PagedAttention*. Proceedings of the 29th ACM Symposium on Operating Systems Principles, 611-626. https://doi.org/10.1145/3600006.3613165

[15] Yu, G., Jeong, J.S., Kim, G., Kim, S., & Chun, B. (2022). Orca: A distributed serving system for transformer-based generative models. 16th USENIX Symposium on Operating Systems Design and Implementation, 521-538.

[16] Leviathan, Y., Kalman, M., & Matias, Y. (2023). Fast inference from transformers via speculative decoding. Proceedings of the 40th International Conference on Machine Learning, PMLR 202, 19274-19286.

[17] Li, J., et al. (2024). *SCALM: Towards semantic caching for automated chat services with large language models*. 2024 IEEE/ACM 32nd International Symposium on Quality of Service. https://doi.org/10.1109/IWQoS61813.2024.10682957

[18] Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Jin, H., Chen, T., & Jia, Z. (2025). Towards efficient generative large language model serving: A survey from algorithms to systems. *ACM Computing Surveys*, *58*(1). https://doi.org/10.1145/3754448

[19] Zheng, L., Chiang, W., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E.P., Zhang, H., Gonzalez, J.E., & Stoica, I. (2023). Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. Advances in Neural Information Processing Systems 36.

[20] Tabassi, E. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. NIST AI 100-1. https://doi.org/10.6028/NIST.AI.100-1

[21] Otunlape, O. (2026). Governance frameworks for enterprise AI systems operating in regulated environments. *International Journal of Computer Applications*, *187*(74), 13-21. https://doi.org/10.5120/ijca2026926244