WHITEPAPER

Altimetrik

The LLM Application Stack in Production

A Practitioner's Field Guide

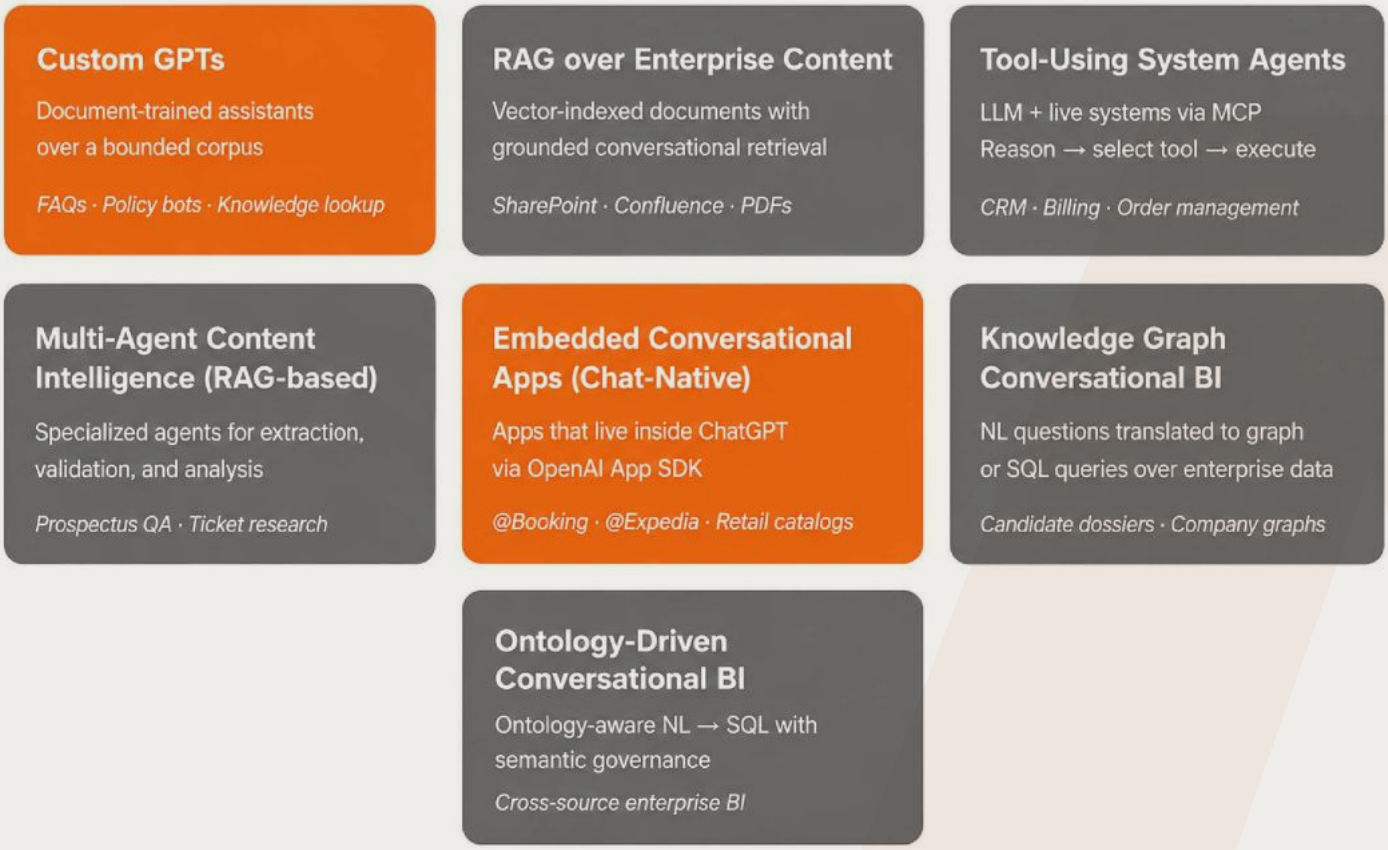
Built on OpenAI

Seven application patterns delivered across 35+ enterprise implementations

AUTHOR
Shivkumar Krishnan,
Data and AI Engineering,
Altimetrik

LLM Application Architecture Patterns

Seven patterns delivered across 30+ enterprise implementations



These patterns are not strictly ordered — they are composable and chosen based on use case requirements.

Most enterprise LLM projects stall between pilot and production. The difference is rarely the model, it is the architecture. This guide synthesizes what we have learned across 35+ enterprise implementations, organized into seven application patterns we have repeatedly delivered. The focus is on the decisions that determined success, and the ones that were made too late.

The observations here come from Altimetrik's AI practice, delivered across financial services, life sciences, retail, and wealth management, in partnership with OpenAI, Microsoft, AWS, and Databricks. Everything described in this guide is drawn from production work, not theory.

Executive Summary

Enterprise teams investing in LLM applications often struggle to bridge the gap between successful proofs of concept and production systems. This guide focuses on the patterns Altimetrik has implemented and delivered to address that challenge. Across these implementations, the root cause of that gap is almost never the model. It is where business logic lives, how the corpus is bounded, when to decompose into multiple agents, and how cross-cutting concerns like observability, access control, and human-in-the-loop are handled.

Four findings recur across the implementations behind this guide, and the seven patterns that follow serve as the evidence base for each:

- LLM applications are architecture conversation, not a model conversation.
- The seven patterns are composable building blocks, not a maturity ladder. A single enterprise solution often combines two or three for example, RAG-based retrieval inside a multi-agent content intelligence system, or a tool-using agent that surfaces ontology-driven BI results inside an embedded chat interface.
- Cross-cutting concerns observability, guardrails, identity, latency, accuracy and hallucination control are where most production engineering happens.
- Start with a clearly bounded use case, not a platform.

This guide is deliberately scoped to deliver work. Patterns we have not implemented such as voice and multimodal agents, real-time streaming systems, or fine-tuned domain models are intentionally out of scope.

A note on cross-cutting concerns

Four capabilities deserve special mention because they apply across every pattern in this guide rather than constituting a separate application type:

Human-in-the-loop (HITL):

Almost every production deployment we describe includes some form of human oversight reviewer queues, approval gates, certification workflows, or feedback loops. HITL is an architectural feature, not an architectural pattern.

LLM-as-a-judge:

Using one LLM to evaluate the output of another (for accuracy, tone, hallucination, or policy compliance) is a quality control technique that can be embedded inside any of the seven patterns. We treat it as a tool, not a separate application type.

Accuracy and hallucination control:

Every pattern in this guide can produce confident-sounding answers that are factually wrong, and the failure modes are pattern-specific. Retrieval-grounded systems hallucinate when the retrieval matches keywords but misses intent; tool-using agents hallucinate when they select the wrong tool or fabricate inputs to one; multi-agent systems compound the risk across stages; conversational BI systems hallucinate when intent-to-traversal or intent-to-SQL goes wrong. The production discipline that holds up across patterns is defense-in-depth: grounding checks that verify the response is supported by the retrieved context, automated evaluation (including LLM-as-a-judge sampling described above) against a curated golden dataset of representative questions and verified answers, deterministic validation of tool inputs and outputs where the schema allows it, and an explicit user feedback channel that converts flagged errors into adversarial test cases a practice that deserves its own treatment below. Accuracy is, in the end, not a launch milestone but a continuously measured property of the running system.

Adversarial testing:

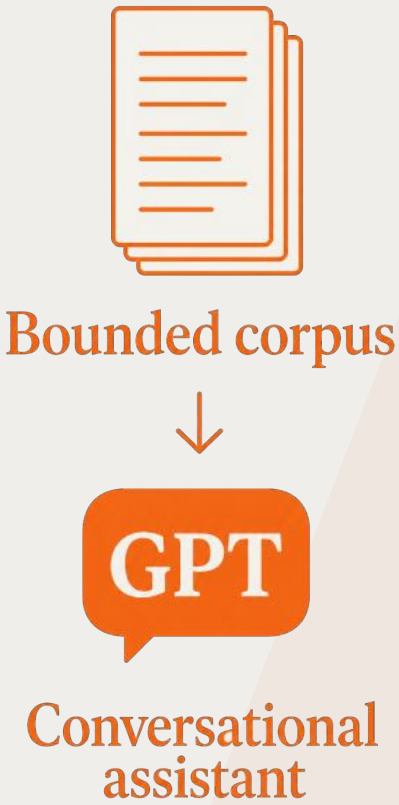
Golden-dataset evaluation and adversarial testing are often treated as a single concern, but they answer different questions. The golden dataset establishes what the system should get right; adversarial red teaming establishes what it should refuse to get wrong. The three attack vectors that matter most in practice are prompt injection (attempts to override the system's instructions), contradictory context (two retrieved documents that disagree with each other), and the void query (a question about a topic the system has no grounded knowledge of). The right behavior in a void query is to admit ignorance, not to confabulate. A system that confidently answers a void query is failing silently; one that recognizes the gap and says so is working correctly. Across our implementations, both golden-dataset evaluation and adversarial testing have typically been built on open-source libraries such as RAGAS, DeepEval, and OpenAI Evals, integrated into CI/CD pipelines and complemented by managed evaluation services where available on the underlying cloud platform.

The Seven Application Patterns

The remainder of this guide examines the seven application patterns we have repeatedly delivered across enterprise AI implementations. These patterns are not a maturity model or sequential roadmap. They are composable architectural building blocks that can be combined to address different business problems. Each pattern includes where it fits, when to use it, key architectural decisions, and lessons learned from production deployments.

1. Custom GPTs

Most accessible starting point - suited to bounded knowledge and conversational lookup



What it is

Custom GPTs are lightweight, document-trained assistants designed to answer questions from a bounded knowledge corpus or specialized assistants with a predefined system prompt. They require minimal orchestration and are well suited to FAQs, internal policy bots, and knowledge lookup. When a Custom GPT needs to reach out to other source systems, it can do so through GPT Actions, which hand control off to a backend call.

Customer Use Case

Patent filing is slow, expensive, and uncertain partly because inventors lack tools to systematically validate their ideas against prior art before investing in the process. We implemented this approach for a large healthcare customer who wanted to give inventors in the organization a tool to refine their inventions, validate them against the company's prior patents and external patent databases, and improve the likelihood of a successful patent filing. The Research Agent needed to query Google Patents, run online searches, and assemble a structured invention report.

With Custom GPTs, one of the standard mechanisms to invoke external systems is GPT Actions, where control passes from the GPT to the action and then to a backend call. This kept the orchestration footprint light while still giving us programmatic access to external data sources.

A practical note on what the Custom GPT layer is and isn't good for. The platform is optimized for conversational interaction with bounded knowledge, not for orchestrating complex backend operations. Custom GPT Actions work well for synchronous, fast-returning calls lookups, structured queries, simple writes. They are a poor fit for long-running operations, asynchronous workflows, streaming, or anything that requires holding state between calls. Likewise, the GPT layer itself does not participate in your enterprise identity model. If a user's permissions matter, that enforcement has to live in the backend API the Action calls, not in the GPT. The pattern that holds up across these constraints is to treat the Custom GPT as a thin conversational shell over a backend (commonly APP Service + API Management plus Blob Storage and Cosmos DB) that handles file processing, authentication, durable storage, and any operation that cannot complete in a single fast request.

Key architectural decision: When the corpus is small and the workflow is mostly retrieval and light reasoning, a Custom GPT plus GPT Actions is often sufficient. Reach for heavier orchestration only when the action surface grows beyond what GPT Actions can cleanly express.

When not to use this pattern: Avoid this pattern when the knowledge corpus is large enough to require chunking & vector retrieval, when you need fine-grained access controls per document, or when the workflow requires multi-step reasoning across many systems. These requirements may sometimes warrant a custom user experience beyond ChatGPT.

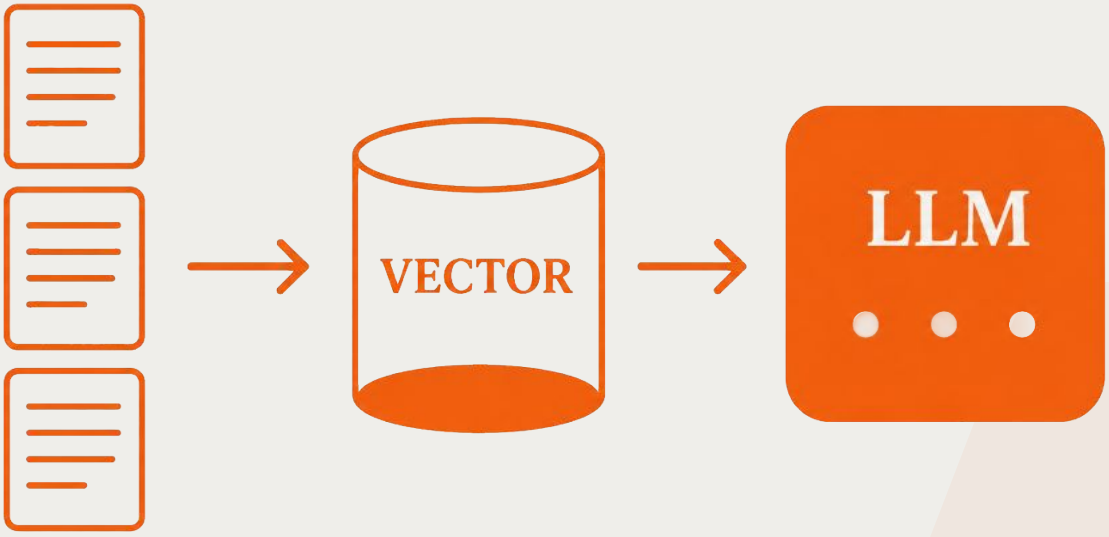
Tech Stack: OpenAI, ChatKit UI, Azure, Postgres PGVector

2. Retrieval-Augmented Generation (RAG) over Enterprise Content

Common enterprise pattern - high value for knowledge discoverability and compliance-governed content

What it is

These applications index enterprise documents (PDFs, slide decks, Word documents, SharePoint, Confluence) into vector databases and use a natural-language chat interface backed by LLMs for retrieval, summarization, and question answering. They improve discoverability of institutional knowledge while preserving source grounding through citations.



Customer Use Case

Enterprise knowledge is locked in documents that nobody can find when they need them. A global alternative investment firm had thousands of expert network transcripts in SharePoint, valuable institutional intelligence that was effectively invisible to analysts. **The client, a leading firm specializing in providing both debt and equity capital to small and mid-sized companies,** engaged us to improve internal knowledge discoverability, beginning with expert network transcripts stored in SharePoint (approximately 5,000 documents).

Earlier Microsoft connector–based integrations had been deployed, but limitations in folder scoping, manual path entry, and inconsistent tool behavior produced poor usability, low adoption, and hallucinated outputs. We delivered a Model Context Protocol (MCP) Server hosted in Azure that integrates with the customer's SharePoint environment and registers directly with ChatGPT as an MCP Server. This removes the need for Custom GPTs or manual connector configuration. Users can now query transcripts directly in ChatGPT, with grounded responses sourced only from controlled SharePoint datasets.

Key architectural decision: Choosing between Custom GPT connectors, native RAG pipelines, and MCP servers comes down to how the corpus is structured, how access control is enforced, and how the user expects to invoke the assistant. For controlled corpora with strong governance requirements, an MCP server gives you the cleanest separation between the data layer and conversational layer.

When not to use this pattern: Avoid this pattern when the LLM does not need to autonomously select tools, run multi-step operations, or write back to source systems. Retrieval-grounded generation alone is sufficient for many use cases and is architecturally lighter.

Tech Stack: Azure (MCP Server hosting), Microsoft SharePoint, Microsoft Graph API, Open AI, ChatGPT MCP integration

3. Tool-Using System Agents (++++)

High-value pattern for workflow automation requires investment in guardrails and observability

What it is

Once a use case moves beyond answering questions and starts requiring action such as placing an order, updating a record, triggering a workflow, the LLM has to reach into live enterprise systems and operate on them. Tool-using agents (often implemented as MCP Servers) connect to APIs and enterprise platforms such as order management systems, inventory, CRMs, or workflow engines. The LLM reasons, selects tools, and executes tasks on the user's behalf. These systems leverage orchestration frameworks such as LangChain, LangGraph, and OpenAI's Agents SDK.

At this layer, orchestration, observability, and guardrails become critical, small reasoning errors propagate quickly when the agent has the ability to act on live systems.

Customer Use Case

A customer support chatbot built on point-to-point integrations is a liability, every new channel or API adds fragility. A leading American multichannel video programming distributor and entertainment provider, had built an agentic customer support chatbot on Sierra and Salesforce Agentforce with point-to-point integrations across multiple internal APIs for billing and order management. The result was a fragile, spaghetti-style architecture that was difficult to extend or operate.

We built a multi-agent MCP platform that introduced a clean orchestration layer between the conversational front end and the backend systems for billing and order management. One of the most consequential technology choices in this kind of work is where the routing and business logic for backend calls lives in the calling agent (Sierra, in this case) or in the backend MCP server layer. There is no universally correct answer; the right placement depends on the organization's maturity in this space, whether it is greenfield or brownfield, the volume of reuse expected across surfaces, and how strictly the backend logic must be governed.

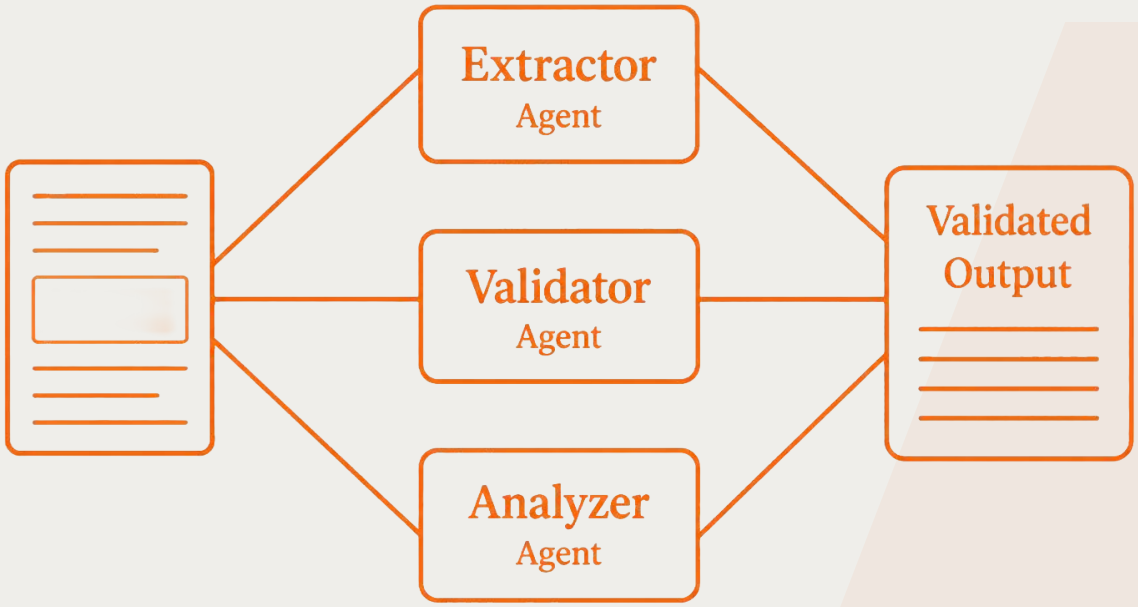
Key architectural decision: Decide intentionally where business logic lives, calling agent vs. MCP server layer. Placing logic in the calling agent is faster to iterate and works well when the behavior is surface-specific. Placing it in the MCP server layer eliminates the need to replicate logic across multiple agents or channels and guarantees consistent outcomes regardless of which agent calls it, which becomes critical when the same backend is consumed by Sierra, Salesforce Agentforce, or any future surface. The wrong default early on becomes very expensive later. There was also a design choice to be made whether to build an MCP server and deploy that into AWS Bedrock AgentCore Runtime or leverage the out-of-box features of AWS AgentCore Gateway and use that as the MCP server. Our performance tests highlighted that deploying using the AgentCore Runtime added additional latency, whereas the Gateway reduces latency as it just acts as a pass-through.

When not to use this pattern: Avoid this pattern for purely informational use cases where no system mutation is needed. The cost of guardrails, observability, and rollback infrastructure is hard to justify if the agent never needs to take action.

Tech Stack: Amazon Bedrock, AWS AgentCore Gateway, Sierra, Salesforce Agentforce

4. RAG-based Multi-Agent Content Intelligence

Powerful for document-heavy domains life sciences, financial services, legal. High build complexity justified by quality ceiling of single-agent approaches



What it is

As use cases grow more complex, single-agent reasoning often is not enough. Multi-agent systems decompose problems across specialized agents, for example: extracting structured data from unstructured content, validating outputs, and performing domain-specific analysis.

This approach is especially powerful in domains like Life Sciences, and industrial data, where documents contain dense tables, specifications, and nuanced semantics that no single prompt can reliably handle. You need specific agents tuned to extract specific forms of content.

Customer Use Case

A 100-step manual validation process was the only way a wealth management firm could verify a new fund prospectus before launch. Every page required checking for tone, numerical consistency, and regulatory language by hand, across documents that ran to over 100 pages. Financial figures referenced on different pages had to remain consistent, a 20% gross margin of reference on one page should not surface as a different number elsewhere. Tables containing metrics, and texts, logos with embedded text had to be parsed and reconciled for semantic understanding.

We built a multi-agent architecture that chunks the document using a RAG approach, then dispatches specialized agents to validate each characteristic and generate an inconsistency report.

In a separate workstream for the same customer, the team was spending significant time researching and responding to investor queries that had often already been answered. Investors filed tickets in Zendesk that required detailed research and a written response that were repetitive and inefficient. We built an agentic AI system that performs RAG over the entire Zendesk ticket history and automatically drafts a response to new investor queries.

What gets underestimated about systems like this, is the indexing pipeline that sits underneath the agents. For the investor support workstream, the agents were the visible part, but the engineering effort lived in decisions made before any agent ran: which sources to ingest and at what cadence, how to parse a dozen file formats including images, how to chunk semantically within each ticket so context did not bleed across unrelated cases, where to redact PII before embedding, what metadata to carry for filtering and access control, how to cluster related tickets for navigation, and how to enforce citation discipline so answers always pointed back to a real source. Multi-agent RAG projects stall when these decisions are made implicitly. The agents are easy; the indexing surface is where the work actually is.

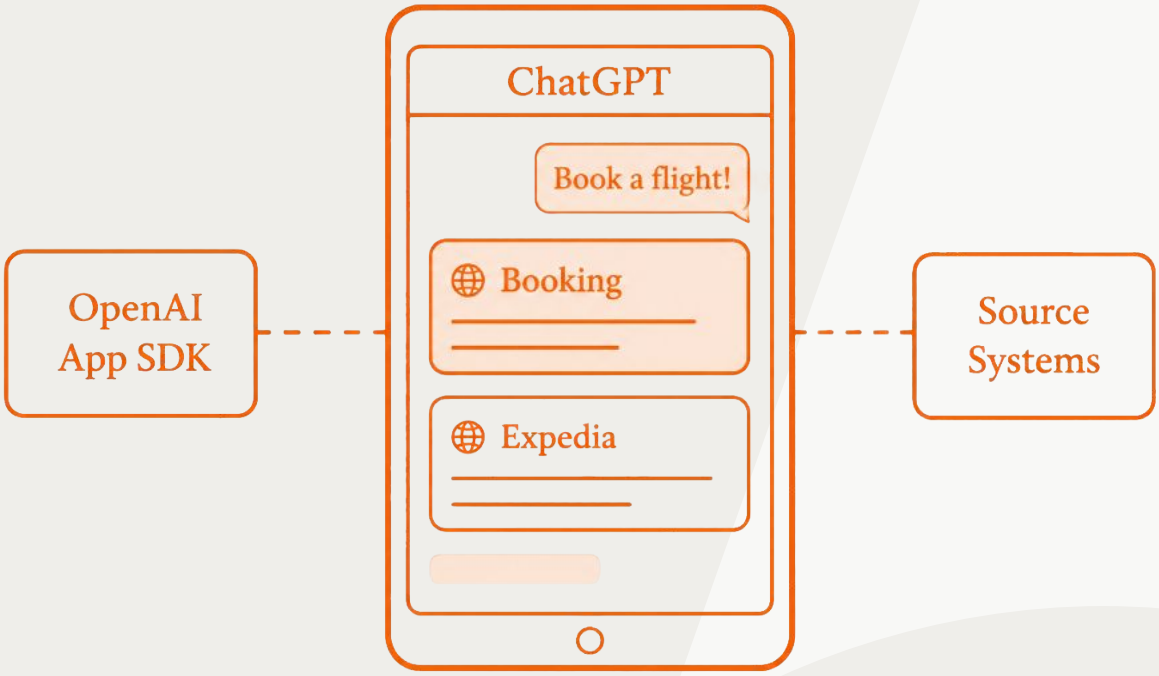
Key architectural decision: Split agents along functional boundaries (extract, validate, summarize) rather than document sections. Functional decomposition gives you reusable agents; document-section decomposition gives you a brittle pipeline that breaks the moment the document layout changes.

When not to use this pattern: Avoid multi-agent decomposition when a single well-prompted agent reliably solves the task. Multi-agent systems multiply latency, cost, and failure modes — they should be reached for only when single-agent approaches have demonstrably hit a quality ceiling.

Tech Stack: Azure AI, Microsoft Graph API, Azure OpenAI Vision model, Zendesk

5. Embedded Conversational Apps (Chat-Native App Model)

Emerging pattern - high reach, lower control. Best suited to transactional consumer or enterprise use cases with clear handoff boundaries



What it is

This emerging class of applications is integrated into the ChatGPT experience using OpenAI's App SDK. Examples include @Booking, @Expedia, and @Instacart. These apps act as transactional conversational interfaces inside ChatGPT itself and are currently platform-native.

They represent an App Store-style approach for both consumer and enterprise applications, giving users access to internal data sources and transactional capabilities directly through ChatGPT. For example, a retailer can build a customer interface inside ChatGPT to look up item availability, ask product questions, and complete a transaction, with the GPT serving information that is accurate up to the moment by interacting directly with backend source systems.

Customer Use Case

This is an emerging pattern we are seeing increased interest in across both consumer and enterprise contexts. Instead of building a destination application, the enterprise extends its capabilities into a conversational surface owned by a third party. This changes how identity, authorization, observability, and branding need to be designed. This also makes the user experience seamless for business users as they can avoid setting up MCP Connectors.

For enterprises evaluating this pattern, the key questions are: which transactions are appropriate to expose through a third-party chat surface, how user authentication and entitlement are handled, and how the experience degrades gracefully when the user wants something the embedded app cannot do.

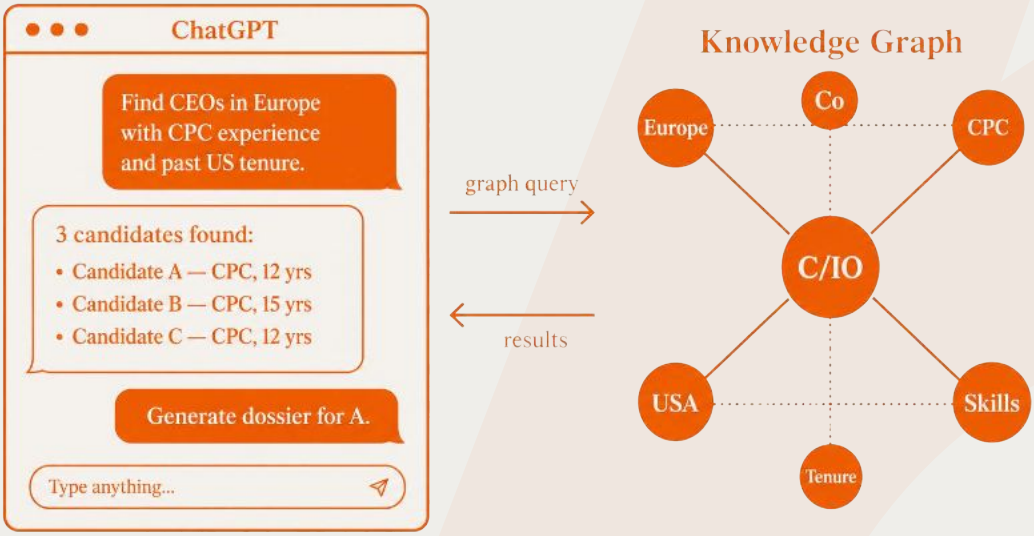
Key architectural decision: Treat the embedded conversational app as an extension of your product surface, not a replacement for it. Decide upfront which user journeys live entirely inside ChatGPT and which must be handed off to a richer destination experience.

When not to use this pattern: Avoid this pattern when the user journey requires complex multi-screen workflows, heavy data input, or branded visual experiences that a chat surface cannot reproduce. Avoid it also when regulatory or compliance requirements limit where customer interactions can occur.

Tech Stack: OpenAI App SDK, ChatGPT

6. Knowledge Graph Conversational BI

High-value for relationship-rich data - justified when the connections across entities are themselves the insight



What it is

These systems ingest enterprise datasets into a data lake and construct knowledge graphs representing relationships across entities. LLMs interpret natural-language user questions and generate structured graph queries or SQL, often by classifying user intent through semantic or orchestration layers enabling a conversational BI experience.

Customer Use Case

Executive recruiters work with datasets that are too large, too interconnected, and too dynamic for traditional search to serve well. Finding a CMO with a European base, CPG experience, and US tenure is a multi-dimensional query that no spreadsheet handles gracefully. A 100+ year old large executive placement firm relied on this network knowledge that was stored in the legacy databases that was hard to converse with. We ingested datasets covering companies, industries, and candidate information (PII, qualifications, work experience) into Databricks and augmented the schema with metadata. This dataset was used to construct a graph database, loaded with 150 million nodes.

The customer had two primary use cases: a Candidate Dossier and a Company Dossier. The conversational interface allows recruiters to search through the dataset of candidates by location, qualifications, and past experience, and everything about their profile then generate a dossier that can be shared with potential employers. The Company Dossier serves the inverse purpose, generating a company profile that can be shared with candidates with information about the funding of the company, financial metrics, board members, etc.

For example, a recruiter can ask: "Find me Chief Marketing Officers based in Europe who have experience working in the Consumer Goods industry with prior experience working in the USA." The LLM parses the intent, walks the knowledge graph to identify candidates that match, and the recruiter can then select a candidate and ask the LLM to generate a dossier from a provided template.

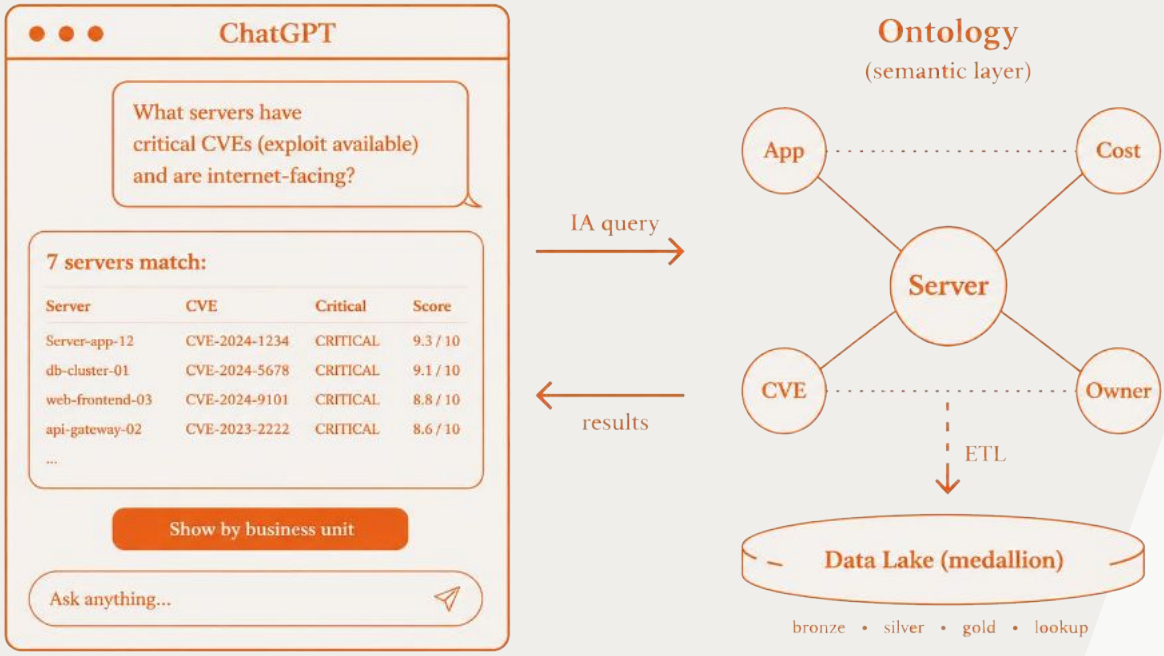
Key architectural decision: Schema modeling: what becomes a node, an edge, or an attribute determines which questions are answerable in a single hop versus expensive traversals. Spend the time before you load; reshaping a populated graph at 150 million nodes is painful. Pair this with the data-placement decision discussed at the end of Section 7.

When not to use this pattern: Avoid this pattern when the underlying data is small, simple, or already well served by traditional BI tools. The investment in graph modeling, ingestion pipelines, and ontology metadata is only justified when the relationships across the data are themselves the source of insight.

Tech Stack: Azure AI, Neo4j, Microsoft Graph API, Microsoft Fabric, ChatGPT, Open AI

7. Ontology-Driven Conversational BI

Most architecturally complex pattern - justified when semantic governance and cross-source consistency are more important than query latency



What it is

This is the most architecturally involved pattern we have delivered. It formalizes enterprise semantics through ontologies. The LLMs map natural-language questions to ontology-aware queries, ensuring semantic consistency, governance, and high-precision analytics across complex data estates.

The main concept here is the use of an ontology-based knowledge graph to drive a conversational BI system. Interconnected data across various legacy enterprise systems is ingested into a data lake, augmented with LLM-generated metadata, and then used to construct a knowledge graph that represents relationships at the level of an ontology. This ontology is then used to translate the user's conversational queries into SQL.

The fundamental difference between pattern 6 and pattern 7 is whether the knowledge graph contains real data or the ontology.

Loading the ontology into the knowledge graph introduces another layer of indirection that can affect latency. A choice between these two approaches should be made based on the size of the data, the performance needs, and the complexity of the user queries

Customer Use Case

A large financial intelligence company had server estate data, vulnerability data, and FinOps data sitting in four disconnected legacy systems — none of which could be correlated. The result was that infrastructure decisions were being made without a complete picture of risk or cost. This was one of the more complex implementations we delivered for a large financial services customer - a large public company that offers financial intelligence including credit ratings, analytics, and benchmarks to the financial and commodity markets.

The customer asked us to bring visibility into their disparate legacy data sources. They had data spread across four legacy systems: a server estate of thousands of nodes in ServiceNow CMDB, vulnerability data about those servers in two different systems, and FinOps data for those servers in yet another system - none of which could be correlated.

We built a medallion architecture that brought all the data together in Databricks and used agentic AI to generate metadata across the entire schema and construct an ontology, with a human-in-the-loop step for certification. This was further enhanced using Databricks Genie multi-agent orchestration and LLM based metadata generation to support a conversational BI interface. The result gave the customer a real understanding of their data across previously disconnected sources and enabled materially better decisions.

Key architectural decision: Choose between pattern 6 (graph holds data) and pattern 7 (graph holds ontology, data stays in the lake) based on data volume, query complexity, and acceptable latency. Pattern 7 trades latency for semantic precision and governance.

When not to use this pattern: Avoid this pattern when the data estate is small, when the semantic model is simple, or when the organization does not yet have the data engineering maturity to support ontology curation and human-in-the-loop certification.

Tech Stack: Databricks (medallion architecture), Databricks Genie, agentic metadata generation, Azure, AWS

Choosing between Patterns 6 and 7

The two patterns make the same set of decisions and answer the middle one differently. Three decisions matter, in order.

1. Schema Modelling (applies to both patterns) What becomes a node, edge, or attribute determines which questions are inexpensive. This decision applies to both patterns and should be made before loading.

2. Data Placement (where the patterns diverge)

	Pattern 6	Pattern 7
Graph holds	The data	The ontology
Strength	Fast traversal, direct queries	Semantic precision, stronger governance
Trade-off	Performance tied to data volume and refresh cadence	Extra layer of indirection costs latency

3. Intent-to-Traversal (the hard part, regardless of pattern) Parsing a user query like "CMOs in Europe with CPG experience and prior US tenure" into an executable plan is the hard part of conversational BI over either substrate. Invest in this layer neither the graph nor the ontology will make the experience feel natural on its own.

Conclusion

Looking across these 35+ implementations, a few patterns become hard to ignore.

LLM applications are an architecture conversation, not a model conversation

Every meaningful decision in the projects above came down to architecture: where does business logic live, how do you bound the corpus, when do you decompose into multiple agents, when does the knowledge graph hold data versus ontology, where do you place the human-in-the-loop checkpoint, and how do you think about RBAC. The choice of model rarely determines success or failure. The choice of architecture almost always did.

The patterns are composable, not sequential

The seven patterns are not a maturity ladder you climb in order. They are building blocks that can be combined with several of them a tool-using agent that calls a RAG system, a multi-agent content intelligence pipeline that surfaces results through an embedded conversational app, an ontology-driven BI system whose answers are validated by a downstream LLM-as-a-judge before being returned. Treating these as composable shifts the design conversation from "which pattern do we pick" to "which combination of patterns matches the use case."

Cross-cutting concerns are where most of the engineering happens

Once the pattern is chosen, the bulk of the work is in concerns that span every pattern: observability, evaluation, guardrails, identity and entitlement, source-of-truth governance, latency budgets, fallback behavior, token use and human-in-the-loop design. None of these are pattern-specific, and underinvesting in them is the single most common reason production LLM applications underperform.

Start with a clearly bounded use case, not a platform

Across these implementations, the projects that delivered fastest and held up longest started with a narrow, well-bounded use case one workflow, one user persona, one source of truth and grew from there. Projects that started by building a "general-purpose enterprise AI platform" almost always struggled. The patterns in this guide are tools to deliver against a specific use case, not a blueprint for building everything at once.

What this guide deliberately does not cover

This guide covers patterns that my team has delivered in the Open AI ecosystem, other LLM platform based solutions will be part of a different playbook in future. There are other meaningful production patterns video RAG, voice and multimodal agents, real-time streaming systems, fine-tuned domain-specialized models, on-device deployment, code-generation, developer-assist tooling that are excluded in order to have a contained scope.

Final thought

Enterprises should view LLM applications not as isolated tools but as a portfolio of patterns to be combined. Each pattern carries a different rigor in data architecture, governance, and user experience design. The discipline is not in mastering any single pattern it is in choosing the right combination, drawing the architectural lines in the right places, and investing in the cross-cutting concerns that make the chosen combination work in production.

Acknowledgments

I am grateful to Altimetrik leadership for the opportunity to lead these initiatives across multiple industries and customer engagements. I am also thankful to OpenAI for their strong partnership and guidance during many of these implementations. The customer organizations we worked with shaped each project through their use cases and their willingness to push on hard problems. Within Altimetrik, I am grateful to colleagues across Architecture, Data Engineering, ML Engineering, Program Management, UX design, Sales, Client Partners and Training & Enablement, teams whose work on these projects made this synthesis possible. Individual contributions were acknowledged within each project as it was delivered.

altimetrik.com

AUTHORS
Shivkumar Krishnan,
Altimetrik

Altimetrik