Retrieval-Augmented Generation Playbook

2026: Live Customer Data Integration Strategies

TEALIUM®

Contents

Most enterprise RAG systems are retrieving from snapshots.
A vector store loaded last Tuesday. A knowledge base that syncs overnight.
A customer profile that reflects who someone was twelve hours ago.

That works for internal documentation search. It does not work when a customer is mid-session, their cart value just crossed a threshold, and your AI agent needs to decide whether to offer free shipping or a loyalty perk. By the time your batch-indexed data catches up, the moment is gone.

This playbook is for teams building RAG systems that need to operate on live customer data, not stale copies of it. That could be for specific applications or even as part of an agentic harness. We'll cover architecture decisions, retrieval stack selection, real-time sync patterns, privacy controls, evaluation frameworks, and phased rollout strategies. Every recommendation is grounded in production benchmarks and real enterprise deployments.

If you're building RAG against static documents, you have plenty of good resources already.
If you're building RAG against customer data that changes by the minute, keep reading.

Define Scope and Service-Level Agreements

Before selecting a vector database or arguing about chunking strategies, answer one question: what customer interactions depend on retrieval accuracy, and how fresh does the data need to be?

This sounds obvious. In practice, most teams skip it. They build a general-purpose RAG pipeline and then discover that their support automation use case needs sub-second data freshness while their product recommendation use case tolerates minutes of lag. Those are fundamentally different architectures.

Map your use cases to SLAs. For each RAG-powered interaction, define three targets:

- **Retrieval latency:** How fast does the system need to fetch relevant context? Production RAG systems at DoorDash operate at 2.5-second end-to-end response times across hundreds of thousands of daily support queries [1]. Your target should be specific to each use case.
- **Data freshness:** How quickly must a change in the source system (a new ticket, an updated profile, a recent purchase) be available for retrieval? Real-time support agents need seconds. Weekly reporting summaries can tolerate hours.
- **Accuracy threshold:** What retrieval precision is acceptable before the system should escalate to a human? LinkedIn's hybrid RAG-knowledge graph system achieved a 28.6% reduction in resolution times by setting clear accuracy gates [2].

Build a mapping table. One column for the customer interaction (billing inquiry, order status, product troubleshooting). One for the data sources required (CRM, order management, ticketing). One for latency and freshness SLAs. One for the expected outcome.

This table becomes your architecture's spec sheet. Every technical decision downstream should trace back to it.

Inventory and Normalize Live Customer Data Sources

RAG quality is bounded by the quality of what gets retrieved. If your customer data lives in six systems with six different schemas, your retrieval layer inherits all that inconsistency.

Start with an inventory. List every data source your RAG system needs to access: CRM records, support tickets, product analytics events, knowledge base articles, transaction histories, behavioral event streams. For each source, document the schema, update frequency, data format, and any access constraints.

Then normalize. Normalization means structuring varied customer data formats into a consistent schema so your embedding and retrieval layers receive uniform inputs regardless of where the data originated. A "customer name" field should mean the same thing whether it comes from Salesforce, Zendesk, or your internal CDP.

Unified ingestion layers make this manageable at scale. Integration platforms like Boomi, Merge, or a CDP like Tealium's AudienceStream can aggregate and standardize inputs across sources, reducing the connector-by-connector complexity that kills RAG projects in their first quarter [10]. Tealium's approach is particularly relevant here because it handles this unification in real time rather than through batch ETL jobs, which means your RAG system retrieves from a profile that reflects the customer's current state, not yesterday's.

Three normalization priorities:

First, resolve identities. A customer who emails from one address, logs in with another, and calls from a phone number registered to a third needs to be one record. Identity resolution is a prerequisite infrastructure for customer-facing RAG, not a nice-to-have.

Second, standardize event schemas. Whether a "purchase completed" event comes from your mobile app, web checkout, or POS system, the schema your retrieval layer sees should be identical. This is where event stream platforms (EventStream, Segment, mParticle) earn their keep.

Third, enforce data quality at ingestion. Garbage in, hallucinations out. Validate required fields, flag anomalies, and reject malformed records before they reach your vector store. Sixty percent of new RAG deployments now include systematic evaluation from day one, up from less than 30% in early 2025 [3]. That trend exists because teams learned the hard way that post-hoc quality checks don't catch ingestion problems fast enough.

Select the Optimal Retrieval Stack

Your retrieval method determines what your LLM sees. Choose wrong, and even the best model generates responses grounded in irrelevant context.

Three retrieval approaches, each with trade-offs:

Vector search (semantic retrieval) converts queries and documents into embeddings and finds the closest matches by meaning. It excels when customers phrase questions in unpredictable ways because it captures intent, not just keywords. The downside: it can retrieve semantically similar but factually wrong context if your embedding model isn't well-tuned to your domain.

Keyword/BM25 search (lexical retrieval) matches on exact terms. It's fast, predictable, and excellent when precision matters more than flexibility. Policy lookups, SKU searches, and compliance queries where specific terminology must match exactly are BM25's territory.

Hybrid retrieval combines both. This is where most production enterprise systems land. A hybrid approach runs both semantic and keyword retrieval, then merges and re-ranks the results. It balances the recall strength of vectors with the precision of lexical matching.

Framework comparison for enterprise RAG:

Framework	Key Strength	Integrations	Best For
LangChain	Workflow flexibility, debugging (125K+ GitHub stars) [17]	700+ [17]	Complex multi-step retrieval chains
LlamaIndex	Multi-source indexing, managed infrastructure [17]	Extensive	Data-heavy RAG with diverse source types
Haystack	UsModular, real-time-ready pipelines [17]	Growing	Production systems needing pipeline flexibility
Meilisearch	Blazing-fast hybrid search, typo tolerance [17]	REST API	Applications requiring sub-50ms search

Vector database selection matters equally. Current early 2026 benchmarks show meaningful performance differences [4]:

Framework	Key Strength	Integrations
Pinecone	30ms @ 1M vectors	Fully managed, minimal ops overhead
Weaviate	45ms (p95) @ 500K vectors	Hybrid search with knowledge graph relationships
Qdrant	50ms @ 1M vectors	Self-hosted, complex metadata filtering

Pick your vector database based on your operational model, not just raw latency numbers. If your team doesn't want to manage infrastructure, Pinecone's managed service saves months of ops work. If you need sophisticated metadata filtering on customer attributes during retrieval, Qdrant's Rust-based engine handles that well. If your retrieval needs to traverse relationships between entities (customer to account to product to support history), Weaviate's graph-aware approach fits.

Implement Real-Time Data Sync and Incremental Indexing

This is where most RAG-with-customer-data projects either differentiate or stall. Static RAG indexes documents once and retrieves forever. Customer-facing RAG needs continuous synchronization between source systems and the retrieval layer.

Change Data Capture (CDC) is your foundation. Log-based CDC monitors database transaction logs to capture inserts, updates, and deletes as they happen. Unlike bulk-load approaches that re-process entire tables, CDC transmits only changed data, which minimizes network bandwidth, processing overhead, and (critically) indexing lag. Tools like Debezium paired with Kafka provide this for most relational databases.

The data flow looks like this:

Source system change → CDC capture → streaming pipeline (Kafka, Kinesis) → embedding generation → vector store upsert → available for retrieval.

Each step introduces latency. Your SLA from Step 1 determines how aggressively you need to optimize each stage. For sub-second freshness, you need streaming embedding generation (not batch) and a vector database that supports real-time upserts without locking the index [13].

Incremental indexing is the key architectural pattern. Rather than rebuilding your entire vector index when data changes, incremental indexing updates only the affected chunks. A customer's profile update re-embeds and upserts that customer's vectors. A new support ticket adds new chunks. A resolved ticket gets its status metadata updated.

This matters at scale. An enterprise with 10 million customer profiles can't afford to re-embed everything hourly. Incremental indexing keeps retrieval current without the compute cost of full rebuilds.

Tealium's EventStream handles the ingestion side of this equation by capturing behavioral events in real time and making them immediately available to downstream systems [10]. When paired with a streaming pipeline to your vector store, this creates a path from "customer clicked a product" to "that interaction is retrievable by an AI agent" in seconds, not hours.

Architecture considerations for production:

Keep your hot-tier index (recent, frequently accessed data) separate from your cold-tier archive. Research on the LiveVectorLake pattern demonstrates that dual-tier temporal architectures enable real-time semantic search on current knowledge while maintaining complete version history [9]. This matters when your RAG system needs to answer "what's happening now" and "what happened last quarter" from the same pipeline.

Apply Structured Context Injection and Privacy Controls

Retrieval gets data to the model. Context injection determines what the model actually sees. This is where privacy, compliance, and response quality converge.

Structured context injection means sending only necessary, validated, and auditable data elements into the LLM prompt. Not raw database dumps. Not entire customer records. Precisely the fields relevant to the query, validated against a schema, with sensitive data redacted or masked before the prompt is assembled.

The Model Context Protocol (MCP) is emerging as the standard interface for this. MCP defines how external data sources expose structured context to AI models, including schema definitions, access controls, and audit trails. Tealium's Moments API now supports MCP integration, which means consented, real-time customer data can flow directly to AI models through a governed protocol rather than ad-hoc API calls with hand-rolled privacy logic [10] [11].

Privacy controls are non-negotiable for customer data RAG. A system that retrieves and injects customer data into LLM prompts without proper safeguards is a compliance incident waiting to happen. Especially when external LLM providers are involved. According to Tonic.ai's analysis, chatbots processing personal data under GDPR require explicit consent management and data minimization at every stage of the pipeline, from retrieval through generation [14].

Required controls:

Field-level redaction. Strip PII that isn't necessary for the response before any customer data reaches the prompt. Social security numbers, full payment details, medical record identifiers. None of these should ever appear in a prompt sent to an external model. De-identification approaches that replace sensitive entities with synthetic alternatives maintain data utility while eliminating exposure risk [15].

Schema validation. Every context injection should be validated against a predefined schema. If a field isn't in the approved schema, it doesn't get injected. This sounds like overhead until an upstream data model change adds a new sensitive field and it starts flowing through your pipeline unredacted. Schema validation catches that automatically.

Access controls gate which data elements are retrievable based on three factors: the requesting agent's permissions, the customer's consent status, and the use case. A marketing agent shouldn't retrieve support case details. A support agent shouldn't access financial data unless the query demands it. Role-based (RBAC) or attribute-based (ABAC) models both work. Pick based on your existing identity infrastructure.

Audit logging. Log what data was retrieved, what was redacted, what was sent to the model, and what response was generated. Every injection event, every time. This audit trail is essential for GDPR's right to explanation and HIPAA's access tracking requirements [14]. It also saves your team hours of debugging when a response goes wrong in production.

Encryption protects data in transit (TLS) and at rest (AES-256). For highly sensitive use cases, confidential RAG architectures keep data encrypted throughout the entire retrieval-to-inference lifecycle, so even the compute environment never sees plaintext [15].

Use a checklist before going to production: validate schema, confirm redaction rules, verify consent status, enforce access controls, enable audit logging, test with synthetic PII.

Monitor, Evaluate, and Optimize Retrieval Performance

A RAG system that works on launch day and degrades silently over the following months is worse than no RAG system at all. Users learn to distrust it, and rebuilding that trust is harder than building it the first time.

Core metrics to track:

Retrieval quality measures whether the right context gets retrieved. Use Precision@k (what fraction of retrieved chunks are relevant), Recall@k (what fraction of relevant chunks get retrieved), and MRR (Mean Reciprocal Rank, how high the first relevant result ranks). These tell you whether your retrieval layer is doing its job before the LLM even gets involved.

Generation quality measures whether the LLM produces faithful, relevant responses from the retrieved context. The RAGAS framework (Retrieval Augmented Generation Assessment) evaluates this without requiring ground-truth labels, using three core metrics: faithfulness (is the answer supported by the context?), recall (does the context match the expected answer?), and relevance (does the answer address the query?) [7].

End-to-end metrics connect retrieval performance to business outcomes. Response latency, hallucination rate, escalation rate, and user satisfaction scores. These are the numbers your stakeholders care about.

Dual evaluation approach. Automated LLM judges handle the volume. DoorDash's production system uses a multi-metric LLM judge that evaluates every response in real time, catching hallucinations and relevance failures before they reach customers [1]. But automated judges have blind spots. Supplement them with periodic human review cycles where domain experts assess a sample of responses for accuracy and appropriateness.

Tools like LangSmith provide trace-level observability into RAG chains, letting you debug exactly where a bad response went wrong. Was the retrieval off? Was the context correct but the generation unfaithful? Was the query ambiguous? Arize Phoenix adds real-time monitoring for retrieval latency and generation quality with root cause analysis [5]. Pick the observability stack that fits your existing infrastructure, but pick one. Operating a customer-facing RAG system without observability is flying blind.

Execute a Phased Rollout

Don't launch RAG across every customer-facing channel simultaneously. You'll learn more from a focused pilot than from a broad rollout that generates too many signals to interpret.

Phase 1: Pilot (Weeks 1-4). Pick one high-impact, well-bounded use case. Billing inquiries, order status lookups, or product troubleshooting all work well because the expected answers are verifiable and the data sources are clearly defined. Deploy RAG alongside your existing system (shadow mode or A/B test), not as a replacement.

Phase 2: Measure and Iterate (Weeks 5-8). Evaluate retrieval accuracy and latency against the SLAs you defined in Step 1. If precision is low, experiment with chunking strategies. If latency is high, profile your pipeline to find the bottleneck (embedding generation, vector search, or LLM inference usually dominate). If relevance is inconsistent, test re-ranking approaches. This phase is where you tune, not where you scale.

Phase 3: Expand (Weeks 9-16). Once your pilot use case meets SLA targets consistently over two or more weeks, add use cases incrementally. Each new use case likely needs its own data source inventory and normalization work from Step 2. Don't assume the infrastructure that works for billing support will work unchanged for product recommendations.

Phase 4: Production scale. Full deployment across all target use cases with monitoring, alerting, and escalation paths in place. At this point, your focus shifts from "does it work" to "does it keep working" and "can we make it faster."

DoorDash followed a similar pattern. They started with a focused support automation use case, validated their three-component architecture (RAG + LLM guardrail + LLM judge), achieved a 50% reduction in development time, and then expanded to broader support coverage handling hundreds of thousands of daily queries [1].

Scale Retrieval Accuracy with Re-Ranking and Knowledge Graphs

As you move beyond pilot use cases into complex, multi-domain customer interactions, basic vector retrieval starts showing its limits. A query about a customer's billing dispute might require context from their account history, recent support interactions, product documentation, and company policy. Vector similarity alone doesn't know how to weigh those sources.

Re-ranking adds a scoring layer between retrieval and generation. After your retriever returns the top-k candidate chunks, a re-ranker (often a cross-encoder model) evaluates each candidate against the original query with more computational depth than the initial retrieval. This post-processing step catches cases where a semantically similar but contextually irrelevant chunk would have been included.

Cross-encoder re-rankers are more accurate but slower than bi-encoder retrieval. For customer-facing applications, the latency trade-off is usually worth it. A 50ms re-ranking step that improves precision by 15-20% means fewer hallucinated responses and fewer escalations [5].

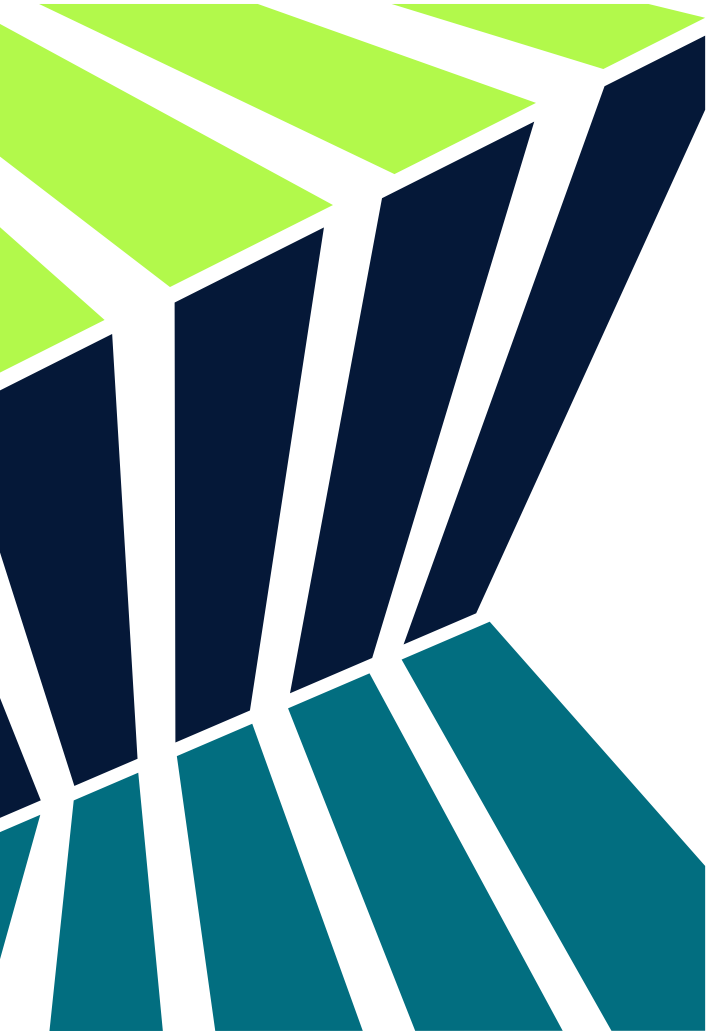
Knowledge graphs go further. Where vector search treats each chunk independently, knowledge graphs encode relationships between entities: customers, products, accounts, interactions, policies. This structural information lets the retrieval system traverse connections rather than just matching similarity.

LinkedIn's hybrid RAG-knowledge graph system demonstrates the impact. By combining traditional RAG with a knowledge graph that captures entity relationships across their platform, they achieved a 28.6% faster resolution rate for support queries [2]. The knowledge graph provides the relational context that pure vector retrieval misses.

GraphRAG, the combination of graph-based retrieval with augmented generation, is gaining serious academic and production traction. Research accepted to ICLR 2026 (LinearRAG, GraphRAG Benchmark) and AAAI 2026 (LogicRAG) is pushing the boundaries on how graph structure can improve retrieval for complex, multi-hop queries [8]. Microsoft Research's open-source GraphRAG implementation provides a practical starting point for teams exploring this approach [16].

For customer data RAG, knowledge graphs are especially valuable because customer relationships are inherently relational. A customer is connected to accounts, subscriptions, orders, support cases, and interaction histories. Traversing those relationships during retrieval produces more complete and accurate context than similarity search alone.

Enforce Security, Compliance, and Data Governance



Every component of your RAG pipeline that touches customer data is a potential compliance surface. The retrieval layer, the context injection layer, the LLM itself, and the response delivery layer all need security controls.

Governance checklist for production RAG:

Access controls at every layer. Your data ingestion pipeline, vector store, context injection service, and LLM endpoint should each enforce independent access controls. A compromised retrieval layer shouldn't expose data that the context injection layer's RBAC would have blocked. Defense in depth applies to RAG just as it does to any data system.

Certifications and compliance. Enterprise RAG deployments handling customer data typically require SOC 2 Type II certification at minimum. If you're in healthcare, HIPAA compliance is mandatory for any component that processes PHI. GDPR applies to any EU customer data, with specific requirements around right to erasure (which means your vector store needs delete capabilities, not just append) and right to explanation (which means your audit logs need to capture the full retrieval-to-response chain) [14].

Audit logging across the pipeline. Log every retrieval query, every context injection, every LLM invocation, and every response delivery. Include timestamps, requesting agent identity, customer data accessed, redaction actions applied, and response content. These logs serve double duty: compliance evidence and debugging data.

Secure data streams. When using external LLM providers, your context injection layer is the last line of defense against data exposure. All data transmitted to external models should be minimized (only what's necessary), redacted (PII stripped), encrypted in transit, and logged [15]. Consider whether your most sensitive use cases require self-hosted models to avoid external transmission entirely.

Regular audit cycles. Quarterly reviews of access control configurations, data retention policies, redaction rule effectiveness, and compliance posture. Annual penetration testing of the RAG pipeline end-to-end. Compliance doesn't end at deployment.

Measure ROI and Continuously Refine

RAG systems are investments. They need to show returns, and those returns need to be measured in business terms that matter beyond the AI team.

Track these business KPIs:

Resolution time is the most visible metric. How much faster are customer issues resolved with RAG-assisted agents versus without? LinkedIn's 28.6% improvement is a useful benchmark [2]. DoorDash's system handles hundreds of thousands of daily queries at 2.5-second response times [1]. Start by measuring your current baseline, then track the delta.

Containment rate tells you whether the system is actually resolving interactions or just responding to them. What percentage of customer queries are fully handled without human escalation? Higher containment directly reduces staffing costs, and it's the metric your operations team will care about most.

The one that often surprises leadership is cost per interaction. Compare the total cost of a RAG-assisted interaction (compute, API calls, infrastructure) against a human-handled interaction. Enterprise RAG implementations have documented 300-500% ROI in year one through productivity gains [6]. One European bank saved EUR 20 million over three years with an AI implementation that achieved ROI in just two months [6].

Finally, customer satisfaction. CSAT and NPS scores for RAG-assisted interactions versus human-only interactions. If RAG-assisted interactions score lower, your system is fast but not good enough. If they score comparable or higher, you have a scaling story worth telling.

Optimize iteratively. RAG performance is never "done." The cycle is straightforward: analyze retrieval metrics, identify precision or recall gaps, test a change, measure impact on both retrieval metrics and business KPIs, keep or revert.

Common optimization levers include chunk size and overlap adjustments, embedding model fine-tuning on domain-specific data, re-ranking threshold tuning, query preprocessing improvements, and knowledge graph enrichment. Each change should be evaluated against your SLAs from Step 1. An optimization that improves precision by 3% but adds 200ms of latency might be wrong for a real-time support use case and right for a batch analytics use case.

Token efficiency matters as costs scale. Re-ranking helps here by letting you retrieve broadly but inject narrowly, sending only the top re-ranked chunks to the model rather than every candidate. Fewer tokens per query, same answer quality, lower inference bills.

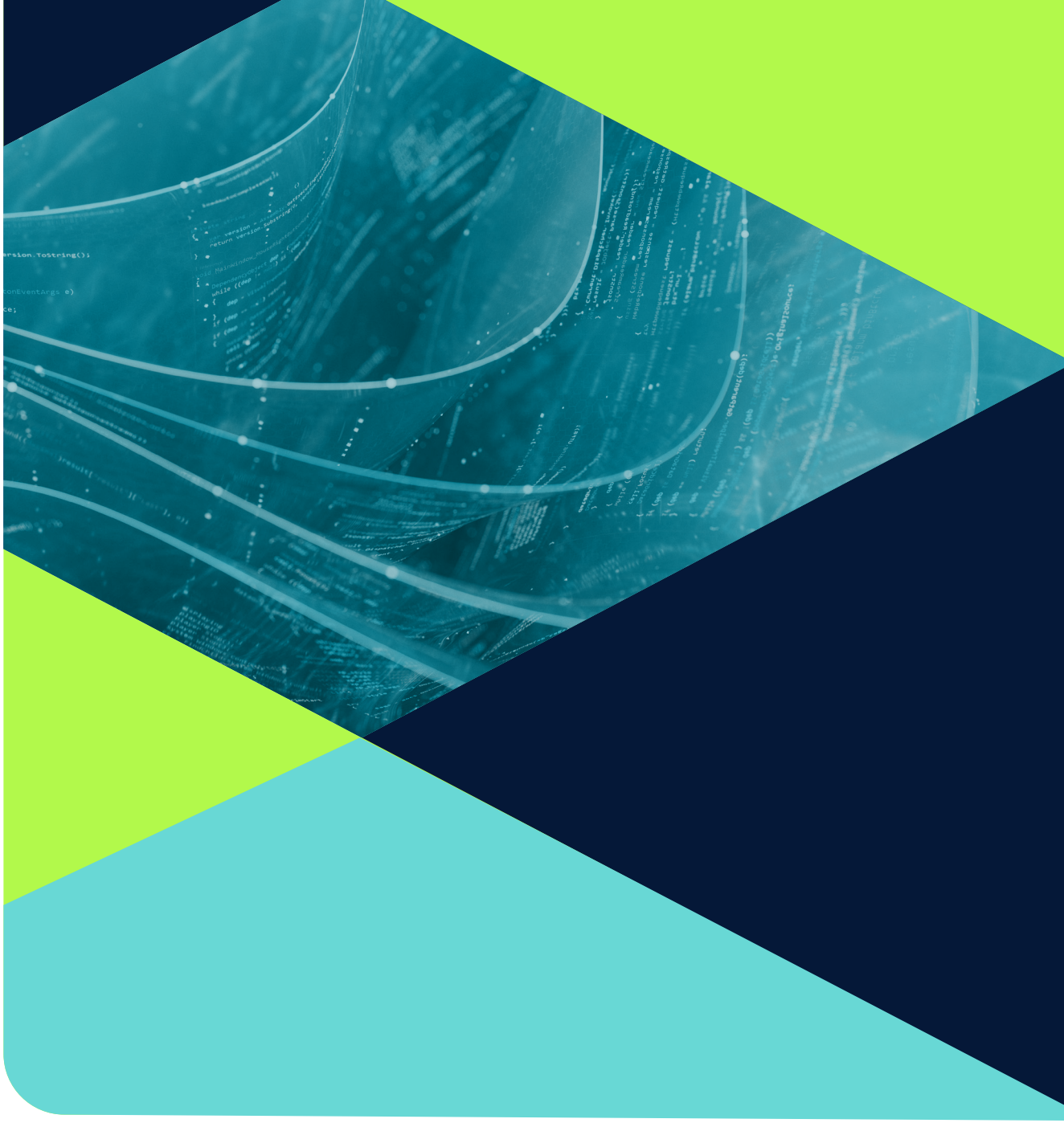
What Separates the Teams That Ship from the Teams That Stall

The technology for RAG with live customer data is production-ready. Vector databases handle millions of embeddings at sub-50ms latencies [4]. Streaming pipelines can move a customer event from source system to retrievable index in seconds [13]. Evaluation frameworks like RAGAS catch hallucinations before they reach users [7]. The hard part was never the models.

The teams that ship customer-facing RAG successfully share three traits. They define SLAs before choosing tools. They normalize data sources before writing retrieval logic. And they measure business outcomes from day one, not after the first quarter review.

The teams that stall do the opposite. They start with the vector database, work backward to the data, and hope the business metrics follow.

Seventy-three percent of RAG implementations are happening in large organizations [3]. The question is no longer whether enterprises will build these systems. It's whether they'll build them on live data or on yesterday's snapshots. The ones that choose live data will serve customers who feel understood in real time. The ones that don't will keep wondering why their AI feels a step behind.



Frequently Asked Questions

What is retrieval-augmented generation and why does live customer data matter?

Retrieval-augmented generation grounds LLM outputs in external data at query time rather than relying solely on the model's training data. When that external data is live customer information (current profiles, recent interactions, real-time behavioral signals), the model's responses reflect the customer's actual situation rather than a stale snapshot. This reduces hallucinations and makes AI-assisted interactions more accurate and relevant. Production deployments have documented 300-500% ROI in year one [6], largely because live data makes the difference between knowing what a customer did yesterday and knowing what they're doing right now.

How do I choose the right retrieval method?

Match the method to the query type. Use vector (semantic) search when customers ask questions in natural language with varied phrasing. Use keyword/BM25 search when exact term matching matters (policy numbers, SKUs, specific product names). Use hybrid retrieval, which combines both approaches, when your use cases include both types of queries. Most production enterprise systems end up with hybrid retrieval because real customer interactions are unpredictable.

What are best practices for maintaining data freshness and low latency?

Implement change data capture (CDC) to stream database changes to your retrieval pipeline in real time [13]. Use incremental indexing so only changed records are re-embedded and upserted, rather than rebuilding the entire index [9]. Profile your pipeline to identify latency bottlenecks. A real-time CDP like Tealium can handle the ingestion and unification side, feeding clean, current customer data to your vector store through streaming pipelines rather than batch ETL [10] [11].

How does structured context injection protect customer privacy?

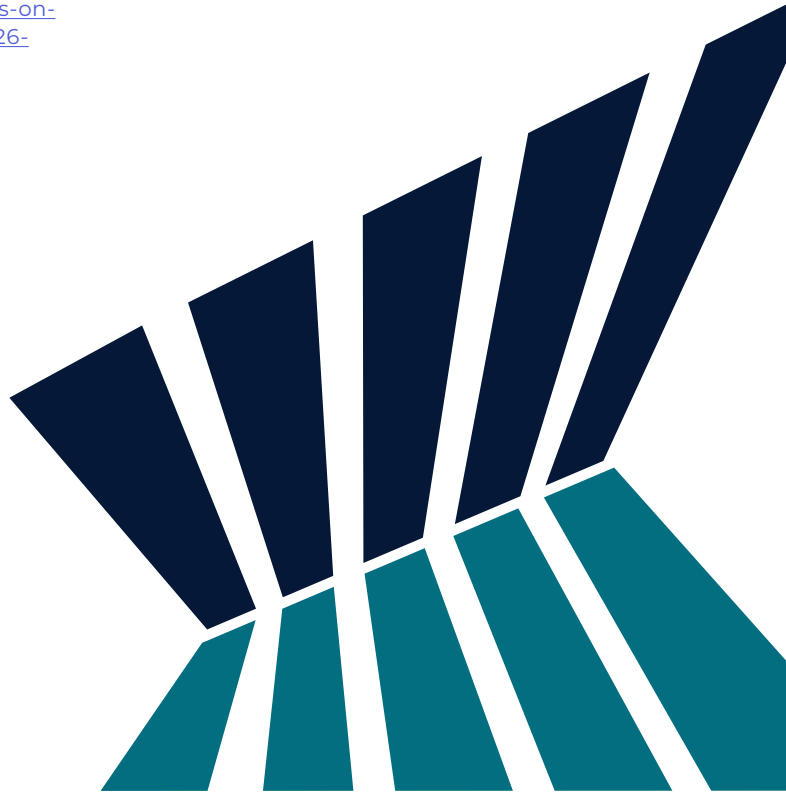
Structured context injection limits what data reaches the LLM to only the fields necessary for the response, validated against an approved schema [14]. Before any customer data enters a prompt, field-level redaction strips PII that isn't required, access controls verify the requesting agent's permissions, consent status is checked, and the entire injection event is logged for audit purposes. The Model Context Protocol (MCP) standardizes this process, providing schema definitions, access controls, and audit trails through a governed interface [10] [11].

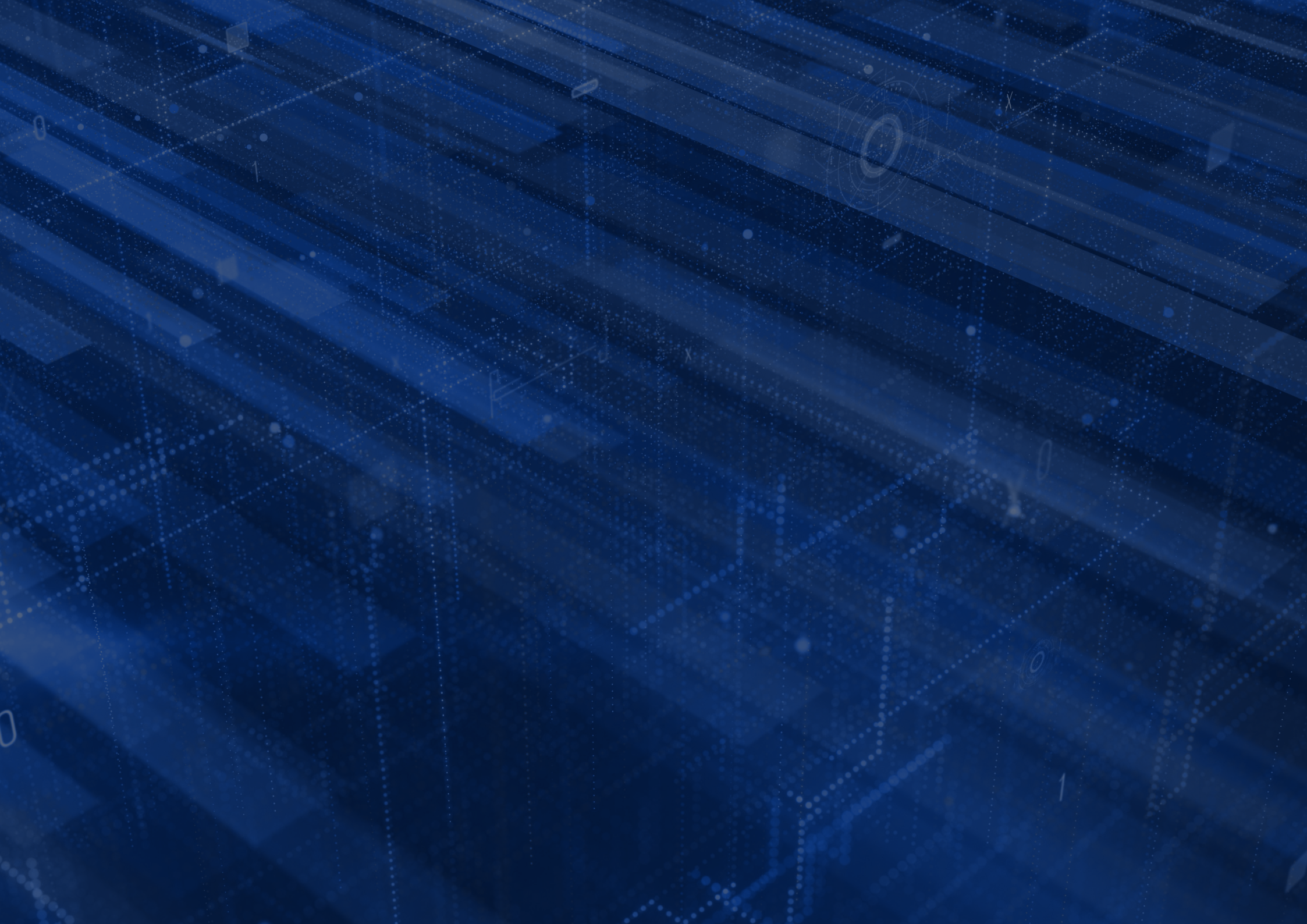
How should I measure success and optimize over time?

Measure at three levels. Retrieval quality (Precision@k, Recall@k, MRR) tells you whether the right context is being found. Generation quality (faithfulness, relevance, hallucination rate via frameworks like RAGAS [7]) tells you whether the model is using that context correctly. Business metrics (resolution time, containment rate, cost per interaction, CSAT) tell you whether any of it matters to the organization. Use automated LLM judges for continuous evaluation at volume [1], supplemented by periodic human review for edge cases.

REFERENCES

1. DoorDash and Amazon Bedrock RAG case study. AWS Solutions Case Studies.
   https://aws.amazon.com/solutions/case-studies/doordash-bedrock-case-study/
2. LinkedIn RAG-knowledge graph implementation. Evidently AI.
   https://www.evidentlyai.com/blog/rag-examples
3. RAG evaluation benchmarks and adoption metrics. Label Your Data.
   https://labelyourdata.com/articles/llm-fine-tuning/rag-evaluation
4. Vector database latency benchmarks. AI Multiple Research.
   https://research.aimultiple.com/vector-database-for-rag/
5. RAG monitoring tools latency overhead. AI Multiple Research.
   https://research.aimultiple.com/rag-monitoring/
6. Enterprise RAG ROI and cost savings. Squirro.
   https://squirro.com/squirro-blog/state-of-rag-genai
7. RAGAS evaluation framework.
   https://docs.ragas.io/en/latest/concepts/metrics/
8. GraphRAG research survey.
   https://arxiv.org/abs/2408.08921
9. LiveVectorLake architecture.
   https://arxiv.org/html/2601.05270
10. Tealium MCP integration.
    https://tealium.com/developer-center/bridging-ai-and-customer-data-platforms-with-mcp/
11. Tealium Moments API MCP server.
    https://docs.tealium.com/server-side/moments-api/managed-mcp-server/
12. RAG hallucination benchmarking (RAGTruth). Cleanlab.
    https://cleanlab.ai/blog/rag-tlm-hallucination-benchmarking/
13. Real-time streaming for RAG. AWS Architecture.
    https://docs.aws.amazon.com/architecture-diagrams/latest/exploring-real-time-streaming-for-retrieval-augmented-generation/
14. RAG compliance and data privacy. Tonic.ai.
    https://www.tonic.ai/blog/ensuring-data-compliance-in-ai-chatbots-and-rag-systems
15. Confidential RAG pipelines. OpenMetal.
    https://openmetal.io/resources/blog/how-to-build-a-confidential-rag-pipeline-that-guarantees-data-privacy/
16. Microsoft GraphRAG.
    https://microsoft.github.io/graphrag/
17. Top RAG Frameworks on GitHub (January 2026). Medium / Florin Elchis.
    https://florinelchis.medium.com/top-10-rag-frameworks-on-github-by-stars-january-2026-e6edff1e0d91





About Tealium

Tealium helps companies collect, govern, and enrich their customer data in real-time to power AI initiatives and delight customers in the moments that matter. Tealium's turnkey integration ecosystem supports more than 1,300 built-in connections from the world's most prominent technology experts. Tealium's solutions include a real-time customer data platform (CDP) with intelligent AI data streaming, tag management, and an API hub. Tealium's data collection, management, and activation capabilities enable enterprises to accelerate operating performance, enhance customer experiences, drive better outcomes, and support global data compliance. More than 850 leading businesses globally trust Tealium to power their customer data strategies.

For more information, visit

tealium.com